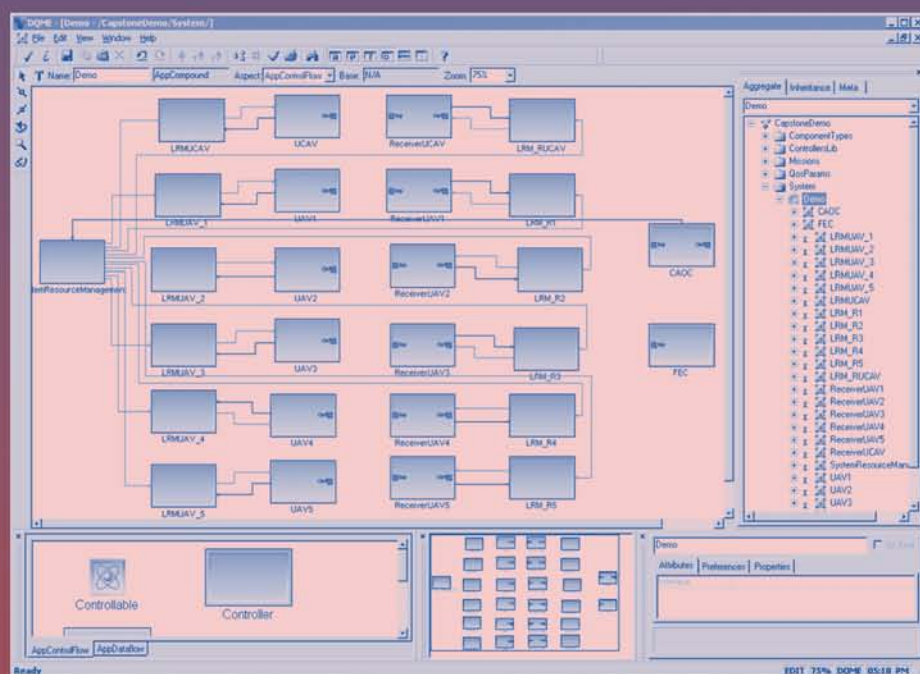


Handbook of Real-Time and Embedded Systems



Edited by
Insup Lee
Joseph Y-T. Leung
Sang H. Son

Handbook of Real-Time and Embedded Systems

CHAPMAN & HALL/CRC

COMPUTER and INFORMATION SCIENCE SERIES

Series Editor: Sartaj Sahni

PUBLISHED TITLES

ADVERSARIAL REASONING: COMPUTATIONAL APPROACHES TO READING THE OPPONENT'S MIND
Alexander Kott and William M. McEneaney

DISTRIBUTED SENSOR NETWORKS
S. Sitharama Iyengar and Richard R. Brooks

DISTRIBUTED SYSTEMS: AN ALGORITHMIC APPROACH
Sukumar Ghosh

FUNDAMENTALS OF NATURAL COMPUTING: BASIC CONCEPTS, ALGORITHMS, AND APPLICATIONS
Leandro Nunes de Castro

HANDBOOK OF ALGORITHMS FOR WIRELESS NETWORKING AND MOBILE COMPUTING
Azzedine Boukerche

HANDBOOK OF APPROXIMATION ALGORITHMS AND METAHEURISTICS
Teofilo F. Gonzalez

HANDBOOK OF BIOINSPIRED ALGORITHMS AND APPLICATIONS
Stephan Olariu and Albert Y. Zomaya

HANDBOOK OF COMPUTATIONAL MOLECULAR BIOLOGY
Srinivas Aluru

HANDBOOK OF DATA STRUCTURES AND APPLICATIONS
Dinesh P. Mehta and Sartaj Sahni

HANDBOOK OF DYNAMIC SYSTEM MODELING
Paul A. Fishwick

HANDBOOK OF REAL-TIME AND EMBEDDED SYSTEMS
Insup Lee, Joseph Y.-T. Leung, and Sang H. Son

HANDBOOK OF SCHEDULING: ALGORITHMS, MODELS, AND PERFORMANCE ANALYSIS
Joseph Y.-T. Leung

THE PRACTICAL HANDBOOK OF INTERNET COMPUTING
Munindar P. Singh

SCALABLE AND SECURE INTERNET SERVICES AND ARCHITECTURE
Cheng-Zhong Xu

SPECULATIVE EXECUTION IN HIGH PERFORMANCE COMPUTER ARCHITECTURES
David Kaeli and Pen-Chung Yew

Handbook of Real-Time and Embedded Systems

Edited by

Insup Lee

University of Pennsylvania
Philadelphia, U.S.A.

Joseph Y-T. Leung

New Jersey Institute of Technology
Newark, U.S.A.

Sang H. Son

University of Virginia
Charlottesville, U.S.A.



Chapman & Hall/CRC

Taylor & Francis Group

Boca Raton London New York

Chapman & Hall/CRC is an imprint of the
Taylor & Francis Group, an **informa** business

Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2008 by Taylor & Francis Group, LLC
Chapman & Hall/CRC is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-10: 1-58488-678-1 (Hardcover)
International Standard Book Number-13: 978-1-58488-678-5 (Hardcover)

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com ([http://www.copyright.com/](http://www.copyright.com)) or contact the Copyright Clearance Center, Inc. (CCC) 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Lee, Insup.

Handbook of real-time and embedded systems / Insup Lee, Joseph Y-T. Leung and Sang H. Son.
p. cm. -- (Chapman & Hall/CRC computer & information science series)

Includes bibliographical references and index.

ISBN-13: 978-1-58488-678-5 (alk. paper)

ISBN-10: 1-58488-678-1 (alk. paper)

1. Real-time data processing. 2. Embedded computer systems--Programming. I. Leung, Joseph Y-T.
II. Son, Sang H. III. Title. IV. Series.

QA76.54.L43 2007

004'.33--dc22

2007002062

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Dedication

To Kisung, David, and Paul.
To my wife Maria.
To Inhye, Daniel, and Yena.

Contents

Preface	xv
Editors	xvii
Contributors	xix

1	Introduction and Overview	<i>Insup Lee, Joseph Y-T. Leung and Sang Hyuk Son</i>	
			1-1
1.1	Introduction		1-1
1.2	Overview		1-2

I Real-Time Scheduling and Resource Management

2	Real-Time Scheduling and Resource Management		
	<i>Giorgio C. Buttazzo</i>		2-1
2.1	Introduction		2-1
2.2	Periodic Task Handling		2-3
2.3	Handling Aperiodic Tasks		2-7
2.4	Handling Shared Resources		2-9
2.5	Overload Management		2-12
2.6	Conclusions		2-14
3	Schedulability Analysis of Multiprocessor Sporadic Task Systems		
	<i>Theodore P. Baker and Sanjoy K. Baruah</i>		3-1
3.1	Introduction		3-1
3.2	Definitions and Models		3-2
3.3	Dynamic Priority Scheduling		3-4
3.4	Fixed Job-Priority Scheduling		3-5
3.5	Fixed Task-Priority Scheduling		3-10
3.6	Relaxations of the Sporadic Model		3-13
3.7	Conclusion		3-15
4	Rate-Based Resource Allocation Methods	<i>Kevin Jeffay</i>	4-1
4.1	Introduction		4-1
4.2	Traditional Static Priority Scheduling		4-3
4.3	A Taxonomy of Rate-Based Allocation Models		4-4
4.4	Using Rate-Based Scheduling		4-9

4.5	Hybrid Rate-Based Scheduling	4-12
4.6	Summary and Conclusions	4-13
5	Compositional Real-Time Schedulability Analysis	
	<i>Insik Shin and Insup Lee</i>	5-1
5.1	Introduction	5-1
5.2	Compositional Real-Time Scheduling Framework	5-3
5.3	Workload Models	5-6
5.4	Resource Models	5-7
5.5	Schedulability Analysis	5-9
5.6	Schedulable Workload Utilization Bounds	5-12
5.7	Extension	5-16
5.8	Conclusions	5-18
6	Power-Aware Resource Management Techniques for Low-Power Embedded Systems	
	<i>Jihong Kim and Tajana Simunic Rosing</i>	6-1
6.1	Introduction	6-1
6.2	Dynamic Voltage Scaling	6-2
6.3	Dynamic Power Management	6-8
6.4	Conclusions	6-12
7	Imprecise Computation Model: Total Weighted Error and Maximum Weighted Error	
	<i>Joseph Y-T. Leung</i>	7-1
7.1	Introduction	7-1
7.2	Total Weighted Error	7-3
7.3	Maximum Weighted Error	7-9
7.4	Concluding Remarks	7-13
8	Imprecise Computation Model: Bicriteria and Other Related Problems	
	<i>Joseph Y-T. Leung</i>	8-1
8.1	Introduction	8-1
8.2	Total <i>w</i> -Weighted Error with Constraints	8-2
8.3	Maximum <i>w'</i> -Weighted Error with Constraints	8-3
8.4	0/1-Constraints	8-6
8.5	Conclusions	8-10
9	Stochastic Analysis of Priority-Driven Periodic Real-Time Systems	
	<i>José Luis Díaz, Kanghee Kim, José María López,</i> <i>Lucia Lo Bello, Daniel F. García, Chang-Gun Lee,</i> <i>Sang Lyul Min and Orazio Mirabella</i>	9-1
9.1	Introduction	9-1
9.2	Related Work	9-2
9.3	System Model	9-3
9.4	Stochastic Analysis Framework	9-4
9.5	Steady-State Backlog Analysis	9-12
9.6	Computational Complexity	9-15
9.7	Experimental Results	9-17
9.8	Conclusions and Future Work	9-22

II Programming Languages, Paradigms, and Analysis for Real-Time and Embedded Systems

10	Temporal Control in Real-Time Systems: Languages and Systems	
	<i>Sebastian Fischmeister and Insup Lee</i>	10-1
10.1	Introduction	10-1
10.2	The Model	10-3
10.3	The Example: A Stopwatch	10-4
10.4	Implicit Temporal Control	10-6
10.5	Programming with Temporal Control	10-7
10.6	Comparison and Conclusions	10-18
11	The Evolution of Real-Time Programming	
	<i>Christoph M. Kirsch and Raja Sengupta</i>	11-1
11.1	Introduction	11-1
11.2	The Computing Abstractions of Control Engineering	11-3
11.3	Physical-Execution-Time Programming	11-6
11.4	Bounded-Execution-Time Programming	11-7
11.5	Zero-Execution-Time Programming	11-10
11.6	Logical-Execution-Time Programming	11-12
11.7	Networked Real-Time Systems	11-14
12	Real-Time Java <i>Andy Wellings and Alan Burns</i>	12-1
12.1	Introduction	12-1
12.2	Background on the RTSJ	12-2
12.3	Scheduling Objects and Scheduling	12-3
12.4	Resource Sharing and Synchronization	12-9
12.5	Time Values and Clocks	12-11
12.6	Memory Management	12-13
12.7	Conclusions	12-17
13	Programming Execution-Time Servers and Supporting EDF Scheduling in Ada 2005 <i>Alan Burns and Andy Wellings</i>	13-1
13.1	Introduction	13-1
13.2	The Ada 95 Version of the Language	13-3
13.3	New Ada 2005 Features	13-3
13.4	Programming Execution-Time Servers	13-8
13.5	Support for Deadlines	13-14
13.6	Baker's Preemption Level Protocol for Protected Objects	13-15
13.7	Supporting EDF Scheduling in Ada	13-15
13.8	Mixed Dispatching Systems	13-19
13.9	Conclusion	13-20
13.10	Postscript—Ada and Java	13-20
14	Synchronous Programming <i>Paul Caspi, Pascal Raymond and Stavros Tripakis</i>	14-1
14.1	Introduction	14-1
14.2	From Practice	14-2
14.3	To Theory	14-3
14.4	Some Languages and Compilers	14-5

14.5	Back to Practice	14-14
14.6	Conclusions and Perspectives	14-19

III Operating Systems and Middleware for Real-Time and Embedded Systems

15	QoS-Enabled Component Middleware for Distributed Real-Time and Embedded Systems <i>Gan Deng, Douglas C. Schmidt, Christopher D. Gill and Nanbor Wang</i>	15-1
15.1	Introduction	15-1
15.2	R&D Challenges for DRE Systems	15-2
15.3	Comparison of Middleware Paradigms	15-2
15.4	Achieving QoS-Enabled Component Middleware: CIAO, DAnCE, and CoSMIC	15-5
15.5	Applications of CIAO, DAnCE, and CoSMIC	15-9
15.6	Related Work	15-13
15.7	Concluding Remarks	15-14
16	Safe and Structured Use of Interrupts in Real-Time and Embedded Software <i>John Regehr</i>	16-1
16.1	Introduction	16-1
16.2	Interrupt Definitions and Semantics	16-2
16.3	Problems in Interrupt-Driven Software	16-4
16.4	Guidelines for Interrupt-Driven Embedded Software	16-9
16.5	Conclusions	16-12
17	QoS Support and an Analytic Study for USB 1.x/2.0 Devices <i>Chih-Yuan Huang, Shi-Wu Lo, Tei-Wei Kuo and Ai-Chun Pang</i>	17-1
17.1	Introduction	17-1
17.2	QoS Guarantees for USB Subsystems	17-6
17.3	Summary	17-18
18	Reference Middleware Architecture for Real-Time and Embedded Systems: A Case for Networked Service Robots <i>Saehwa Kim and Seongsoo Hong</i>	18-1
18.1	Introduction	18-1
18.2	Robot Middleware Requirements	18-2
18.3	Reference Robot Middleware Architecture	18-4
18.4	Future Challenges of Robot Middleware	18-10
18.5	Conclusions	18-12

IV Real-Time Communications/Sensor Networks

19	Online QoS Adaptation with the Flexible Time-Triggered (FTT) Communication Paradigm <i>Luis Almeida, Paulo Pedreiras, Joaquim Ferreira, Mário Calha, José Alberto Fonseca, Ricardo Marau, Valter Silva and Ernesto Martins</i>	19-1
19.1	Introduction	19-1
19.2	Toward Operational Flexibility	19-2

19.3	The Flexible Time-Triggered Paradigm	19-5
19.4	The Synchronous Messaging System	19-8
19.5	The Asynchronous Messaging System	19-14
19.6	Case Study: A Mobile Robot Control System	19-17
19.7	Conclusions	19-19
20	Wireless Sensor Networks <i>John A. Stankovic</i>	20-1
20.1	Introduction	20-1
20.2	MAC	20-2
20.3	Routing	20-2
20.4	Node Localization	20-4
20.5	Clock Synchronization	20-5
20.6	Power Management	20-5
20.7	Applications and Systems	20-6
20.8	Conclusions	20-9
21	Messaging in Sensor Networks: Addressing Wireless Communications and Application Diversity <i>Hongwei Zhang, Anish Arora, Prasun Sinha and Loren J. Rittle</i>	21-1
21.1	Introduction	21-1
21.2	SMA: An Architecture for Sensornet Messaging	21-2
21.3	Data-Driven Link Estimation and Routing	21-6
21.4	Related Work	21-17
21.5	Concluding Remarks	21-18
22	Real-Time Communication for Embedded Wireless Networks <i>Marco Caccamo and Tarek Abdelzaher</i>	22-1
22.1	Introduction	22-1
22.2	Basic Concepts for Predictable Wireless Communication	22-2
22.3	Robust and Implicit Earliest Deadline First	22-3
22.4	Higher-Level Real-Time Protocols for Sensor Networks	22-9
22.5	Real-Time Capacity of Wireless Networks	22-11
22.6	Concluding Remarks	22-13
23	Programming and Virtualization of Distributed Multitasking Sensor Networks <i>Azer Bestavros and Michael J. Ocean</i>	23-1
23.1	Introduction	23-1
23.2	The SNAFU Programming Language	23-4
23.3	Sensorium Task Execution Plan	23-7
23.4	The Sensorium Service Dispatcher	23-9
23.5	Sensorium Execution Environments	23-12
23.6	Putting It All Together	23-16
23.7	Related Work	23-18
23.8	Conclusion	23-19

V Real-Time Database/Data Services

24	Data-Intensive Services for Real-Time Systems <i>Krithi Ramamritham, Lisa Cingiser DiPippo and Sang Hyuk Son</i>	24-1
24.1	Introduction	24-1

24.2	Data Freshness and Timing Properties	24-4
24.3	Transaction Processing	24-6
24.4	Quality of Service in Real-Time Data Services	24-12
24.5	Data Services in Sensor Networks	24-17
24.6	Mobile Real-Time Databases	24-18
24.7	Dissemination of Dynamic Web Data	24-20
24.8	Conclusion	24-21
25	Real-Time Data Distribution <i>Angela Uvarov Frolov,</i> <i>Lisa Cingiser DiPippo and Victor Fay-Wolfe</i>	25-1
25.1	Introduction to Real-Time Data Distribution	25-1
25.2	Real-Time Data Distribution Problem Space	25-1
25.3	Approaches to Real-Time Data Distribution	25-5
25.4	Conclusion	25-16
26	Temporal Consistency Maintenance for Real-Time Update Transactions <i>Ming Xiong and Krithi Ramamritham</i>	26-1
26.1	Introduction	26-1
26.2	More–Less Using EDF	26-3
26.3	More–Less Using <i>Deadline Monotonic</i>	26-7
26.4	Deferrable Scheduling	26-8
26.5	Conclusions	26-17
27	Salvaging Resources by Discarding Irreconcilably Conflicting Transactions in Firm Real-Time Database Systems <i>Victor C. S. Lee, Joseph Kee-Yin Ng and Ka Man Ho</i>	27-1
27.1	Introduction	27-1
27.2	Related Work	27-3
27.3	A New Priority Cognizant CC Algorithm	27-5
27.4	Experiments	27-6
27.5	Results	27-8
27.6	Conclusion	27-13
28	Application-Tailored Databases for Real-Time Systems <i>Aleksandra Tešanović and Jörgen Hansson</i>	28-1
28.1	Introduction	28-1
28.2	Dimensions of Tailorability	28-2
28.3	Tailorable Real-Time and Embedded Database Systems	28-3
28.4	The COMET Approach	28-6
28.5	Summary	28-16
29	DeeDS NG: Architecture, Design, and Sample Application Scenario <i>Sten F. Andler, Marcus Brohede, Sanny Gustavsson and Gunnar Mathiason</i>	29-1
29.1	Introduction	29-1
29.2	Active Research Problems	29-4
29.3	DeeDS NG	29-5
29.4	Related Work	29-16
29.5	Summary	29-17

VI Formalisms, Methods, and Tools

30	State Space Abstractions for Time Petri Nets	
	<i>Bernard Berthomieu and François Vernadat</i>	30-1
30.1	Introduction	30-1
30.2	Time Petri Nets and Their State Space	30-2
30.3	State Space Abstractions Preserving Markings and Traces	30-5
30.4	State Space Abstractions Preserving States and Traces	30-8
30.5	Abstractions Preserving States and Branching Properties	30-13
30.6	Computing Experiments	30-15
30.7	Conclusion and Further Issues	30-16
31	Process-Algebraic Analysis of Timing and Schedulability	
	<i>Properties Anna Philippou and Oleg Sokolsky</i>	31-1
31.1	Introduction	31-1
31.2	Modeling of Time-Sensitive Systems	31-4
31.3	Modeling of Resource-Sensitive Systems	31-9
31.4	Conclusions	31-19
32	Modular Hierarchies of Models for Embedded Systems	<i>Manfred Broy</i>
32.1	Motivation	32-1
32.2	Comprehensive System Modeling Theory	32-3
32.3	Structuring Interfaces	32-10
32.4	Refinement	32-15
32.5	Composition and Combination	32-18
32.6	Modeling Time	32-20
32.7	Perspective, Related Work, Summary, and Outlook	32-23
33	Metamodeling Languages and Metaprogrammable Tools	
	<i>Matthew Emerson, Sandeep Neema and Janos Sztipanovits</i>	33-1
33.1	Introduction	33-1
33.2	Modeling Tool Architectures and Metaprogrammability	33-3
33.3	A Comparison of Metamodeling Languages	33-9
33.4	Relating Metamodeling Languages and Metaprogrammable Tools	33-14
33.5	Conclusion	33-16
34	Hardware/Software Codesign	<i>Wayne Wolf</i>
34.1	Introduction	34-1
34.2	Hardware/Software Partitioning Algorithms	34-2
34.3	Cosynthesis Algorithms	34-4
34.4	CPU Customization	34-5
34.5	Codesign and System Design	34-6
34.6	Summary	34-7
35	Execution Time Analysis for Embedded Real-Time Systems	
	<i>Andreas Ermedahl and Jakob Engblom</i>	35-1
35.1	Introduction	35-1
35.2	Software Behavior	35-4
35.3	Hardware Timing	35-5
35.4	Timing by Measurements	35-8

35.5	Timing by Static Analysis	35-10
35.6	Hybrid Analysis Techniques	35-15
35.7	Tools for WCET Analysis	35-15
35.8	Industrial Experience with WCET Analysis Tools	35-15
35.9	Summary	35-17

VII Experiences with Real-Time and Embedded Systems

36	Dynamic QoS Management in Distributed Real-Time Embedded Systems <i>Joseph P. Loyall and Richard E. Schantz</i>	36-1
36.1	Introduction	36-1
36.2	Issues in Providing QoS Management in DRE Systems	36-2
36.3	Solutions for Providing QoS Management in DRE Systems	36-5
36.4	Case Studies of Providing QoS Management	36-12
36.5	Conclusions	36-30
37	Embedding Mobility in Multimedia Systems and Applications <i>Heonshik Shin</i>	37-1
37.1	Introduction	37-1
37.2	Challenges for Mobile Computing with Multimedia	37-2
37.3	System-Layer Approaches	37-4
37.4	Application-Layer Approaches	37-9
37.5	Conclusions	37-14
38	Embedded Systems and Software Technology in the Automotive Domain <i>Manfred Broy</i>	38-1
38.1	Introduction	38-1
38.2	The History	38-2
38.3	State of Practice Today	38-2
38.4	The Domain Profile	38-3
38.5	The Future	38-4
38.6	Practical Challenges	38-6
38.7	Research Challenges	38-11
38.8	Comprehensive Research Agenda	38-16
38.9	Conclusion	38-18
39	Real-Time Data Services for Automotive Applications <i>Gurulingesh Raravi, Krithi Ramamritham and Neera Sharma</i>	39-1
39.1	Introduction	39-1
39.2	Real-Time Data Issues in Automotive Applications	39-2
39.3	Adaptive Cruise Control: An Overview	39-5
39.4	Our Goals and Our Approach	39-7
39.5	Specifics of the Dual Mode System	39-8
39.6	Specifics of the Real-Time Data Repository	39-10
39.7	Robotic Vehicle Control: Experimental Setup	39-12
39.8	Results and Observations	39-13
39.9	Related Work	39-17
39.10	Conclusions and Further Work	39-18

Index	I-1
-------------	-----

Preface

In the last two decades, we have witnessed an explosive growth of real-time and embedded systems being used in our daily life. A real-time system is required to complete its work and deliver its services on a timely basis. In other words, real-time systems have stringent timing requirements that they must meet. Examples of real-time systems include digital control, command and control, signal processing, and telecommunication systems. Every day these systems provide us with important services. When we drive, they control the engine and brakes of our car and regulate traffic lights. When we fly, they schedule and monitor the takeoff and landing of our plane, make it fly, maintain its flight path, and keep it out of harm's way. When we are sick, they monitor and regulate our blood pressure and heartbeats. When we are well, they entertain us with electronic games and joy rides. When we invest, they provide us with up-to-date stock quotes.

Real-time and embedded systems are gaining more and more importance in our society. Recognizing the importance of these systems, the National Science Foundation has recently established a research program dedicated to embedded systems. The European Union (EU), European countries, and Asian countries have also established many research programs in real-time and embedded systems. Therefore, we can anticipate many important and exciting results being developed in this area.

This book is intended to provide a comprehensive coverage of the most *advanced* and *timely* topics in real-time and embedded systems. A major goal is to bring together researchers in academic institutions and industry so that cross-fertilization can be facilitated. The authors are chosen from both types of institutions.

We would like to thank Sartaj Sahni for inviting us to edit this handbook. We are grateful to all the authors and coauthors who took time from their busy schedules to contribute to this handbook. Without their efforts, this handbook would not have been possible. Theresa Delforn at CRC has done a superb job in managing the project.

This work was supported in part by the National Science Foundation (NSF) under Grants DMI-0300156, DMI-0556010, CCR-0329609, CNS-0614886, CNS-0509327, and CNS-0509143; and by the Army Research Office (ARO) under DAAD19-01-1-0473 and W911NF-05-1-0182. Findings contained herein are not necessarily those of NSF and ARO.

Editors

Insup Lee is the Cecilia Fidler Moore Professor of Computer and Information Science. He received the BS degree in mathematics from the University of North Carolina, Chapel Hill, in 1977 and the PhD degree in computer science from the University of Wisconsin, Madison, in 1983. He joined the Penn faculty in the Department of Computer and Information Science in 1983 and was CSE Undergraduate Curriculum Chair during 1994–1997. He holds a secondary appointment in the Department of Electrical and Systems Engineering.

His research interests include real-time systems, embedded and hybrid systems, cyber physical systems, formal methods and tools, medical device systems, wireless sensor networks, and software engineering. The major theme of his research activity has been to assure and improve the correctness, safety, and timeliness of real-time and embedded systems.

He was chair of IEEE Computer Society Technical Committee on Real-Time Systems during 2003–2004. He has served on numerous program committees and chaired many international conferences and workshops, including RTSS, RTCSA, ISORC, CONCUR, EMSOFT, RV, and HCMDSS. He has also served on various steering and advisory committees of technical societies, including ACM SIGBED, IEEE TCRTS, Runtime Verification, ATVA, and Technical Advisory Group (TAG) on Networking and Information Technology of the President’s Council of Advisors on Science and Technology (PCAST). He has served on the editorial boards of several scientific journals, including *IEEE Transactions on Computers*, *Formal Methods in System Design*, and *Real-Time Systems Journal*. He is an IEEE Fellow and an IEEE CS Distinguished Visitor Speaker.

Joseph Y-T. Leung, PhD, is distinguished professor of computer science in New Jersey Institute of Technology. He received his BA in mathematics from Southern Illinois University at Carbondale and his PhD in computer science from the Pennsylvania State University. Since receiving his PhD, he has taught at Virginia Tech, Northwestern University, University of Texas at Dallas, University of Nebraska at Lincoln, and New Jersey Institute of Technology. He has been the chairman at University of Nebraska at Lincoln and New Jersey Institute of Technology.

Dr. Leung is a member of ACM and a senior member of IEEE. His research interests include scheduling theory, real-time systems, computational complexity, discrete optimization, and operating systems. His research has been supported by NSF, ONR, FAA, and Texas Instruments.

Sang Hyuk Son is a professor at the Department of Computer Science of University of Virginia. He received the BS degree in electronics engineering from Seoul National University, the MS degree from KAIST, and the PhD degree in computer science from University of Maryland, College Park. He has been a visiting professor at KAIST, City University of Hong Kong, Ecole Centrale de Lille in France, and Linköping University in Sweden.

His current research interests include real-time computing, data services, QoS management, wireless sensor networks, and information security. He has served as an associate editor of *IEEE Transactions*

on *Parallel and Distributed Systems*, and is currently serving as an associate editor for *Real-Time Systems Journal* and *Journal of Business Performance Management*. He has been on the executive board of the IEEE TC on Real-Time Systems, and served as the program chair or general chair of several real-time systems and sensor networks conferences, including IEEE Real-Time Systems Symposium. He received the Outstanding Contribution Award at the IEEE Conference on Embedded and Real-Time Computing Systems and Applications in 2004.

Contributors

Tarek Abdelzaher

Department of Computer Science
University of Illinois at
Urbana-Champaign
Urbana, Illinois

Luis Almeida

Departamento de Electrónica,
Telecomunicações e Informática
Universidade de Aveiro, Campo
Universitário
Aveiro, Portugal

Sten F. Andler

School of Humanities and
Informatics
University of Skövde
Skövde, Sweden

Anish Arora

Department of Computer Science
and Engineering
Ohio State University
Columbus, Ohio

Theodore P. Baker

Department of Computer Science
Florida State University
Tallahassee, Florida

Sanjoy K. Baruah

Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina

Lucia Lo Bello

Dipartimento di Ingegneria
Informatica e delle
Telecomunicazioni
Università di Catania
Catania, Italy

Bernard Berthomieu

Laboratoire d'Architecture et
d'Analyse des Systèmes du CNRS
Toulouse, France

Azer Bestavros

Computer Science Department
Boston University
Boston, Massachusetts

Marcus Brohede

School of Humanities and
Informatics
University of Skövde
Skövde, Sweden

Manfred Broy

Software and Systems Engineering
Institut für Informatik, Technische
Universität München
München, Germany

Alan Burns

Department of Computer Science
University of York
York, United Kingdom

Giorgio C. Buttazzo

RETIS Lab
Scuola Superiore Sant'Anna
Pisa, Italy

Marco Caccamo

Department of Computer Science
University of Illinois at
Urbana-Champaign
Urbana, Illinois

Mário Calha

Departamento de Informática
Faculdade de Ciências
Universidade de Lisboa
Lisboa, Portugal

Paul Caspi

Verimag Laboratory
(CNRS) Gières, France

Gan Deng

EECS Department
Vanderbilt University
Nashville, Tennessee

José Luis Díaz

Departamento de Informática
Universidad de Oviedo
Gijón, Spain

Lisa Cingiser DiPippo

Department of Computer Science
University of Rhode Island
Kingston, Rhode Island

Matthew Emerson

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee

Jakob Engblom

Virtutech AB
Stockholm, Sweden

Andreas Ermedahl

Department of Computer
Science and Electronics
Mälardalen University
Västerås, Sweden

Victor Fay-Wolfe

Department of Computer Science
University of Rhode Island
Kingston, Rhode Island

Joaquim Ferreira

Escola Superior de Tecnologia
Instituto Politécnico de
Castelo Branco
Castelo Branco, Portugal

Sebastian Fischmeister

Department of Computer and
Information Science
University of Pennsylvania
Philadelphia, Pennsylvania

José Alberto Fonseca

Departamento de Electrónica,
Telecomunicações e Informática
Universidade de Aveiro, Campo
Universitário
Aveiro, Portugal

Angela Uvarov Frolov

Department of Computer Science
University of Rhode Island
Kingston, Rhode Island

Daniel F. García

Departamento de Informática
Universidad de Oviedo
Gijón, Spain

Christopher D. Gill

CSE Department
Washington University
St. Louis, Missouri

Sanny Gustavsson

School of Humanities and
Informatics
University of Skövde
Skövde, Sweden

Jörgen Hansson

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania

Ka Man Ho

Department of Computer Science
City University of Hong Kong
Kowloon, Hong Kong

Seongsoo Hong

School of Electrical Engineering
and Computer Science
Seoul National University
Seoul, Korea

Chih-Yuan Huang

Department of Computer Science
and Information Engineering
Graduate Institute of Networking
and Multimedia
National Taiwan University
Taipei, Taiwan

Kevin Jeffay

Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina

Jihong Kim

School of Computer Science and
Engineering
Seoul National University
Seoul, Korea

Kanghee Kim

School of Computer Science and
Engineering
Seoul National University
Seoul, Korea

Saehwa Kim

School of Electrical Engineering
and Computer Science
Seoul National University
Seoul, Korea

Christoph M. Kirsch

Department of Computer Sciences
University of Salzburg
Salzburg, Austria

Tei-Wei Kuo

Department of Computer Science
and Information Engineering
Graduate Institute of Networking
and Multimedia
National Taiwan University
Taipei, Taiwan

Chang-Gun Lee

Department of Electrical
Engineering
Ohio State University
Columbus, Ohio

Insup Lee

Department of Computer and
Information Science
University of Pennsylvania
Philadelphia, Pennsylvania

Victor C. S. Lee

Department of Computer Science
City University of Hong Kong
Kowloon, Hong Kong

Joseph Y-T. Leung

Department of Computer Science
New Jersey Institute of Technology
Newark, New Jersey

Shi-Wu Lo

Department of Computer Science
and Information Engineering
National Chung-Cheng University
Chia-Yi, Taiwan

José María López

Departamento de Informática
Universidad de Oviedo
Gijón, Spain

Joseph P. Loyall

BBN Technologies
Cambridge, Massachusetts

Ricardo Marau

Departamento de Electrónica,
Telecomunicações e Informática
Universidade de Aveiro, Campo
Universitário
Aveiro, Portugal

Ernesto Martins

Departamento de Electrónica,
Telecomunicações e Informática
Universidade de Aveiro, Campo
Universitário
Aveiro, Portugal

Gunnar Mathiason

School of Humanities and
Informatics
University of Skövde
Skövde, Sweden

Sang Lyul Min

School of Computer Science and
Engineering
Seoul National University
Seoul, Korea

Orazio Mirabella

Dipartimento di Ingegneria
Informatica e delle
Telecomunicazioni
Università di Catania
Catania, Italy

Sandeep Neema

Institute for Software Integrated
Systems
Vanderbilt University
Nashville, Tennessee

Joseph Kee-Yin Ng

Department of Computer Science
Hong Kong Baptist University
Kowloon Tong, Hong Kong

Michael J. Ocean

Computer Science Department
Boston University
Boston, Massachusetts

Ai-Chun Pang

Department of Computer Science
and Information Engineering
Graduate Institute of Networking
and Multimedia
National Taiwan University
Taipei, Taiwan

Paulo Pedreiras

Departamento de Electrónica,
Telecomunicações e Informática
Universidade de Aveiro, Campo
Universitário
Aveiro, Portugal

Anna Philippou

Department of Computer Science
University of Cyprus
Nicosia, Cyprus

Krithi Ramamritham

Department of Computer
Science and Engineering
Indian Institute of
Technology
Powai, Mumbai, India

Gurulingesh Ravari

Department of Information
Technology
Indian Institute of Technology
Powai, Mumbai, India

Pascal Raymond

Verimag Laboratory
(CNRS) Gières, France

John Regehr

School of Computing
University of Utah
Salt Lake City, Utah

Loren J. Rittle

Pervasive Platforms and
Architectures Lab
Motorola Labs
Schaumburg, Illinois

Tajana Simunic Rosing

Department of Computer Science
and Engineering
University of California
San Diego, California

Richard E. Schantz

BBN Technologies
Cambridge, Massachusetts

Douglas C. Schmidt

EECS Department
Vanderbilt University
Nashville, Tennessee

Raja Sengupta

Department of Civil Engineering
University of California
Berkeley, California

Neera Sharma

Department of Computer Science
and Engineering
Indian Institute of Technology
Powai, Mumbai, India

Heonshik Shin

School of Computer Science
and Engineering
Seoul National University
Seoul, Korea

Insik Shin

Department of Computer and
Information Science
University of Pennsylvania
Philadelphia, Pennsylvania

Valter Silva

Escola Superior de Tecnologia e
Gestão de Águeda
Universidade de Aveiro
Águeda, Portugal

Prasun Sinha

Department of Computer Science
and Engineering
Ohio State University
Columbus, Ohio

Oleg Sokolsky

Department of Computer and
Information Science
University of Pennsylvania
Philadelphia, Pennsylvania

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia

John A. Stankovic

Department of Computer Science
University of Virginia
Charlottesville, Virginia

Janos Sztipanovits

Institute for Software Integrated
Systems
Vanderbilt University
Nashville, Tennessee

Aleksandra Tešanović

Philips Research Laboratories
Eindhoven, The Netherlands

Stavros Tripakis

CNRS/Verimag and Cadence
Berkeley Labs
Berkeley, California

François Vernadat

Laboratoire d'Architecture et
d'Analyse des Systèmes du CNRS
Toulouse, France

Nanbor Wang

Tech-X Corporation
Boulder, Colorado

Andy Wellings

Department of Computer Science
University of York
York, United Kingdom

Wayne Wolf

Department of Electrical Engineering
Princeton University
Princeton, New Jersey

Ming Xiong

Bell Labs
Alcatel-Lucent
Murray Hill, New Jersey

Hongwei Zhang

Department of Computer Science
Wayne State University
Detroit, Michigan

1

Introduction and Overview

Insup Lee

University of Pennsylvania

Joseph Y-T. Leung

New Jersey Institute of Technology

Sang Hyuk Son

University of Virginia

1.1	Introduction	1-1
1.2	Overview	1-2

1.1 Introduction

In the last two decades, we have witnessed an explosive growth of real-time and embedded systems being used in our daily life. A real-time system is required to complete its work and deliver its services on a timely basis. In other words, real-time systems have stringent timing requirements that must be met. Examples of real-time systems include digital control, command and control, signal processing, and telecommunication systems. Every day these systems provide us with important services. When we drive, they control the engine and brakes of our car and also regulate traffic lights. When we fly, they schedule and monitor the takeoff and landing of our plane, make it fly, maintain its flight path, and keep it out of harm's way. When we are sick, they monitor and regulate our blood pressure and heartbeats. When we are well, they entertain us with electronic games and joy rides. When we invest, they provide us with up-to-date stock quotes. Unlike PCs and workstations that run non-real-time applications, such as our editor and network browser, the computers and networks that run real-time applications are often hidden from our view. When real-time systems work correctly and well, they make us forget their existence.

Real-time and embedded systems are gaining more and more importance in our society. Recognizing the importance of these systems, the National Science Foundation has recently established a research program dedicated to embedded systems. The European Union (EU), European countries, and Asian countries also have established many research programs in real-time and embedded systems. Therefore, we can anticipate many important and exciting results being developed in this area.

This book is intended to provide a comprehensive coverage of the most *advanced* and *timely* topics in real-time and embedded systems. A major goal is to bring together researchers in academic institutions and industry so that cross-fertilization can be facilitated. The authors are chosen from both types of institutions.

This is an advanced book in real-time and embedded systems. It is intended for an audience with some knowledge of real-time and embedded systems. The book has chapters that cover both theory and practice. It is suitable for faculty and graduate students in academic institutions, researchers in industrial laboratories, as well as practitioners who need to implement real-time and embedded systems.

1.2 Overview

This book has seven major parts, each of which has several chapters. They are (I) Real-Time Scheduling and Resource Management; (II) Programming Languages, Paradigms, and Analysis for Real-Time and Embedded Systems; (III) Operating Systems and Middleware for Real-Time and Embedded Systems; (IV) Real-Time Communications/Sensor Networks; (V) Real-Time Database/Data Services; (VI) Formalisms, Methods, and Tools; and (VII) Experiences with Real-Time and Embedded Systems.

Part I covers real-time scheduling and resource management issues. Today's embedded systems are becoming more complex than ever and are required to operate under stringent performance and resource constraints. To achieve a predictable behavior and provide an offline guarantee of the application performance, deterministic and analyzable algorithms are required to manage the available resources, especially involving scheduling, intertask communication, synchronization, and interrupt handling. Chapter 2 illustrates some problems that may arise in real-time concurrent applications and some solutions that can be adopted in the kernel to overcome these problems.

Chapter 3 considers the scheduling of systems of independent sporadic tasks to meet hard deadlines upon platforms comprised of several identical processors. Although this field is very new, a large number of interesting and important results have recently been obtained. The authors focus on presenting the results within the context of what seems to be a natural classification scheme.

In Chapter 4, the author argues that while static priority scheduling has worked well for systems where there exists a natural strict priority relationship between tasks, it can be difficult to use it to satisfy the performance requirements of systems when the priority relationships between tasks are either not strict or not obvious. Rate-based resource allocation is a more flexible form of real-time resource allocation that can be used to design systems in which performance requirements are better described in terms of required processing rates rather than priority relationships. In a rate-based system, a task is guaranteed to make progress according to a well-defined rate specification. The author summarizes the theory and practice of rate-based resource allocation and demonstrates how rate-based methods can provide a framework for the natural specification and realization of timing constraints in real-time and embedded systems.

A hierarchical scheduling framework has been introduced to support hierarchical resource sharing among applications under different scheduling services. The hierarchical scheduling framework can be generally represented as a tree of nodes, where each node represents an application with its own scheduler for scheduling internal workloads, and resources are allocated from a parent node to its children nodes. In Chapter 5, the authors present an analysis of compositional real-time schedulability in a hierarchical scheduling framework.

Energy consumption has become one of the most important design constraints for modern embedded systems, especially for mobile embedded systems that operate with a limited energy source such as batteries. For these systems, the design process is characterized by a trade-off between a need for high performance and low-power consumption, emphasizing high performance to meeting the performance constraints while minimizing the power consumption. Chapter 6 discusses power-aware resource management techniques for low-power embedded systems.

Meeting deadline constraints is of paramount importance in real-time systems. Sometimes it is not possible to schedule all the tasks to meet their deadlines. In situations like this, it is often more desirable to complete some portions of every task, rather than giving up completely the processing of some tasks. The Imprecise Computation Model was introduced to allow for the trade-off of the quality of computations in favor of meeting the deadline constraints. In this model, a task is logically decomposed into two subtasks, mandatory and optional. The mandatory subtask of each task is required to be completed by its deadline, while the optional subtask can be left unfinished. If a task has an unfinished optional subtask, it incurs an error equal to the execution time of its unfinished portion. If the importance of the tasks are not identical, we can assign weights to each task, and the resulting task system is called a weighted task system. Chapter 7 describes algorithms for minimizing the total weighted error and the maximum weighted error. The algorithms are for single processor, parallel and identical processor, and uniform processor environments.

Chapter 8 continues with the discussion of bicriteria scheduling of imprecise computation tasks. The author gives algorithms for (1) minimizing the total weighted error, subject to the constraint that the maximum weighted error is minimum, and (2) minimizing the maximum weighted error, subject to the constraint that the total weighted error is minimum. As well, the author discusses the problem of scheduling imprecise computation tasks with 0/1 constraints; i.e., either the optional subtask is fully executed or it is not executed at all. Two performance measures are studied: (1) the total error and (2) the number of imprecisely scheduled tasks (i.e., the number of tasks where the optional subtask is not executed at all). The author proposes approximation algorithms and derive worst-case bounds.

Most schedulability analysis methods for priority-driven systems are to provide a deterministic guarantee, assuming that every job in a task requires its worst-case execution time (WCET). Although such deterministic timing guarantee is needed in hard real-time systems, it is considered too stringent for soft real-time applications, which require a probabilistic guarantee that the deadline miss ratio of a task is below a given threshold. Chapter 9 describes a stochastic analysis framework to compute the response time distributions of tasks for general priority-driven periodic real-time systems. The response time distributions make it possible to determine the deadline miss probability of individual tasks, even for systems with a maximum utilization factor greater than 1. In this framework, the system is modeled as a Markov chain and the stationary backlog distribution is computed by solving the Markov matrix. Chapter 9 also shows that the framework can be uniformly applied to both fixed-priority systems, such as rate monotonic (RM) and deadline monotonic (DM), and dynamic-priority systems, such as earliest deadline first (EDF), by proving the backlog dependency relations between all the jobs in a hyperperiod.

Part II is devoted to programming languages, paradigms, and analysis for real-time and embedded systems. The timing aspect of a behavior is fundamentally important for the correctness of real-time systems. There are several programming languages and systems that support the specification of temporal constraints on a program. However, the syntax and semantics, the granularity, and the supported kinds of temporal constraints differ from system to system.

Chapter 10 introduces the types of temporal constraints and describes different approaches to temporal control as supported by PEARL, Temporal Scopes, the ARTS kernel, Esterel, Real-time Java, and Giotto. These different systems and language constructs are illustrated using a task cycle model that defines the start, release, suspend, resume, and completion stages of computation of a task. The different programming methodologies are contrasted using the same example of a stopwatch task.

Chapter 11 presents a survey on different programming paradigms for real-time systems. The focus of the chapter is to compare the different methodologies with respect to their handling of timing and composability, ranging from low-level physical-time programming over bounded time programming to high-level real-time programming that incorporates abstract notions of time such as synchronous reactive programming and logical execution time programming. Specifically, this chapter describes four models: physical execution time, bounded execution time, zero execution time, and logical execution time programming. The chapter also explains how the evolution of real-time programming is enabling new control software engineering.

Since its inception in the early 1990s, there is little doubt that Java has been a great success. However, the language does have serious weaknesses both in its overall model of concurrency and in its support for real-time systems. Consequently, it was initially treated with disdain by much of the real-time community. Nevertheless, the perceived advantages of Java over languages such as C and C++ resulted in several attempts to extend the language so that it is more appropriate for a wide range of real-time systems. Chapter 12 discusses the extension.

Over the last two decades or more, a considerable volume of literature has been produced which addresses the issues and challenges of scheduling. The focus of much of this research has been on how to effectively support a collection of distinct applications each with a mixture of periodic and nonperiodic, and hard and soft activities. Unfortunately, system implementors have not, in general, taken up these results. There are a number of reasons for this. Chapter 13 discusses several of these reasons.

Synchronous programming is based on the notion of zero execution time and has been used in embedded software, without being always given clear and thorough explanations. Synchronous programming can be found on the widely adopted practice based on synchrony hypothesis, which is to abstract from implementation details by assuming that the program is sufficiently fast to keep up with its real-time environment. This separation of concerns between design and implementation is in full accordance with the model-based development, which advocates the use of high-level models with ideal implementation-independent semantics. Chapter 14 provides a comprehensive introduction to synchronous programming and its use in practice. The chapter describes the theory, the programming languages Esterel and Lustre, and the compilation and scheduling issues of implementation.

Part III covers operating systems and middleware for real-time and embedded systems. Component middleware technologies are becoming increasingly important in developing distributed real-time and embedded systems. Designing, implementing, and evaluating diverse distributed real-time and embedded systems bring several challenges, including quality-of-service (QoS) support, component integration, and configuration management. Chapter 15 discusses these issues and presents an approach to component integration and configuration capabilities.

Interrupt handling is indispensable in developing embedded and real-time systems. Interrupts use hardware support to reduce the latency and overhead of event detection. While they are essential, they also have inherent drawbacks. They are relatively nonportable across compilers and hardware platforms, and they are vulnerable to a variety of software errors that are difficult to track down since they manifest only rarely. In Chapter 16, the author presents a technical description of potential problems of interrupt handling in embedded systems, and provides a set of design rules for interrupt-driven software.

With the recent advance of operating systems and hardware technologies, resource management strategies in embedded systems are receiving more attention from system developers. Since the definition of QoS is different when different types of resources are considered, different enforcement mechanisms for different components are needed. While I/O subsystems have become one of the major players in providing QoS guarantees for real-time and embedded applications, the traditional concepts of resource management may be no longer sufficient for them. In Chapter 17, the authors present a novel approach to provide QoS guarantees for universal serial bus (USB) subsystems, one of the most widely used method for attaching and accessing I/O devices.

While middleware technology has been successfully utilized in the enterprise computing domain, its adoption in commercial real-time and embedded systems is slow due to their extra nonfunctional requirements such as real-time guarantee, resource limitation, and fault tolerance. Chapter 18 presents the reference middleware architecture, called the robot software communications architecture (RSCA), to address these requirements for robot systems. The RSCA illustrates how typical real-time and embedded middleware for robot systems can be organized. It consists of a real-time operating system, a communication middleware, and a deployment middleware, which collectively form a hierarchical structure that enables device abstraction, component modeling, dynamic reconfigurability, resource frugality, and real-time QoS capabilities. The chapter also addresses future challenges arising from new trends related to power-aware high-performance computing, effective data streaming support, and domain-specific abstractions.

Part IV is devoted to real-time communication and sensor networks. For distributed real-time and embedded systems, networking and communication in real-time are essential to achieve timeliness. Recently, providing real-time properties over wireless communication receives lot of attention to support ever-increasing mobile units and sensor devices. As more wireless sensor networks are deployed and used as an integral part of ubiquitous and embedded computing applications, the need for reliable, efficient, and scalable real-time communication becomes critical.

The flexible time-triggered (FTT) paradigm has been developed to support operational flexibility in distributed embedded systems to facilitate maintenance and dynamic reconfiguration and to support continuous adaptation as required for dynamic QoS management while maximizing efficiency in resource

utilization. The FTT paradigm combines flexibility with timeliness, supporting prompt changes to the current communication requirements while guaranteeing the continued timely behavior of the system. Although the FTT paradigm still follows the time-triggered communication model, the FTT paradigm also supports event-triggered traffic but with temporal isolation with respect to time-triggered one. This combination is achieved using a dual-phase cyclic communication framework that supports two separate communication subsystems: Synchronous Messaging System for the time-triggered traffic and Asynchronous Messaging System for the event-triggered one. Chapter 19 focuses on the traffic scheduling issues related with the FTT paradigm with specific contributions to scheduling of the nonpreemptive blocking-free model and to the response-time analysis for the dual-phase cyclic communication.

A wireless sensor network is a collection of nodes organized into a cooperative network. Each node consists of processing units, memory, radio frequency transceiver, power source, and various sensors and actuators. They typically self-organize after being deployed in an *ad hoc* fashion. It is not unrealistic that within the next decade, the world will be covered with wireless sensor networks with real-time access to them via Internet. Chapter 20 presents an overview of some of the key areas and research work in wireless sensor networks. The author uses examples of recent work to portray the state of the art and show how such solutions differ from solutions found in other distributed systems.

In wireless sensor networks, nodes coordinate with one another to accomplish the mission such as event detection, data collection, and target tracking. Since nodes are spatially spread out, message passing is one of the most important functions for coordination. For message passing services, consideration of routing and transport issues is important. While message servicing has been studied extensively in existing networks, such as the Internet, it is still an important issue in wireless sensor networks due to the complex dynamics of wireless communication and resource constraints. The authors of Chapter 21 present an architecture that accommodates and adapts to diverse application traffic patterns and QoS requirements. The architecture also supports diverse in-network processing methods.

For automated real-time monitoring and control, more computer systems have been embedded into physical environments. This trend is expected to continue to improve and secure our quality of life in many areas. For such embedded applications, end-to-end real-time performance is critical to ensure emergency response within bounded delay. Timeliness is a property tightly related to the management of shared resources (e.g., wireless medium). Chapter 22 discusses the challenges in the medium access control (MAC) layer, and provides an overview of a light-weight real-time communication architecture, a delay sensitive network-layer protocol, and a notion of network capacity to quantify the ability of the network to transmit information in time.

In the near future, it is expected that general-purpose and well-provisioned sensor networks that are embedded in physical spaces will be available, whose use will be shared among users of that space for independent and possibly conflicting missions. This view is in contrast to the common view of an embedded sensor network as a special-purpose infrastructure that serves a well-defined special mission. The usefulness of such general-purpose sensor networks will not be measured by how highly optimized its various protocols are, but rather how flexible and extensible it is in supporting a wide range of applications. In Chapter 23, the authors overview and present a first-generation implementation of SNBENCH—a programming environment and associated run-time system that support the entire life cycle of programming sensing-oriented applications.

Part V covers real-time database and data services. Databases have become an integral part of many computer systems, ranging from complex real-time and embedded applications that control network traffic and plant operations to general-purpose computing systems that run audio and video applications. It is interesting to note that a system using real-time data, such as sensor data, does not in itself constitute a real-time database system. In a real-time database system, timing constraints are associated with transactions, and data are valid for specific time intervals. In more recent years, the research that has been performed in the area of real-time database systems has started to be applied and extended in real-time and embedded applications, under the more general term of real-time data services. Chapter 24 describes several key issues of real-time databases and data services that are different from traditional database functionalities.

It also provides a survey of the highlights of advanced features in real-time data services in emerging applications, such as sensor networks, mobile data services, and dynamic web data distribution.

For many real-time and embedded applications, sharing data among their various distributed components is critical. Some of these applications require that data be available within certain time bounds and temporally consistent. Real-time data distribution is the transfer of data from one source to one or more destinations within a deterministic time frame. In Chapter 25, the authors describe the range of the problems associated with real-time data distribution. They also discuss several existing approaches, and how each approach addresses certain aspects of the problem space.

For proper real-time data services, a real-time data object (e.g., the speed or position of a vehicle, or the temperature in the engine) should be temporally consistent if its values reflect the current status of the corresponding entity in the environment. This is typically achieved by associating the value with a temporal validity interval. The sensor transactions that sample the latest status of the entities need to periodically refresh the values of real-time data objects before their temporal validity intervals expire. Chapter 26 discusses the approaches to ensuring that real-time data objects remain temporally valid all the time.

Generally, there are two groups of approaches to concurrency control in real-time database systems: lock-based and optimistic approaches. In optimistic concurrency control (OCC), data conflicts are detected during the validation phase of a transaction. There are several conflict resolution strategies, such as OCC with broadcast commit (OCC-BC) and OCC with time interval (OCC-TI). Regardless of the strategies selected by the system, the rationale behind the decision is to increase the number of transactions meeting deadlines. Since most of the real-time and embedded systems are time and resource constrained, it is essential for concurrency control algorithms to minimize the wastage of resources. In Chapter 27, the authors present a method to salvage resources by discarding irreconcilably conflicting transactions.

The increasing number of real-time and embedded applications and their intrinsic needs to handle large amounts of data bring new challenges to system designers. The real-time database research has shown the merits of integrating database support in such applications, since a data-centric view enables simpler capturing of real-world data requirements, without losing semantics under a typical task-centric approach. In Chapter 28, the authors identify the challenges of application-tailored databases regarding desirable capabilities and their impact on the architecture. They also present the COMET platform and show how component-based software engineering combined with aspect-oriented methodology can benefit tailoring the database software architecture.

Several emerging applications of real-time and embedded systems deal with security and emergency management. They often require resource predictability, high availability, scalability, data consistency, and support for timely recovery. Chapter 29 presents DeeDS NG which is being developed with a goal to support those emerging applications. The DeeDS NG architecture is designed to support several key features, including eventual consistency, scalability with virtual full replication, and distributed fault-tolerant real-time simulation. Using the wildfire scenario as an example of an emerging application in emergency management, the authors illustrate the requirements and the research issues being addressed in DeeDS NG.

Part VI is devoted to formalisms, methods, and tools. The first two chapters present formalisms, timed Petri nets and real-time process algebra. Time Petri nets (TPN) extend Petri nets with temporal intervals associated with transitions, specifying firing delay ranges for the transitions. Timed Petri nets have been widely used for the specification and verification of systems in which time constraints are essential for their correctness, e.g., communication protocols, hardware components, and real-time systems. As with many formal models for real-time systems, the state spaces of TPN are typically infinite. Thus, model checking of TPN depends on finite abstractions of their state spaces that preserve some classes of useful properties. Chapter 30 overviews a number of state space abstractions for TPN that preserve richer classes of properties, such as state reachability properties and branching time temporal logic properties.

Chapter 31 presents an overview of how timing information can be embedded in process-algebraic frame works. The chapter begins by discussing design approaches that have been adopted in different formalisms to model time and time passage, and how the resulting mechanisms interact with one

another and with standard untimed process-algebraic operators. It then gives an overview of ACSR, a timed process algebra developed for modeling and reasoning about timed, resource-constrained systems. ACSR adopts the notion of a resource as a first-class entity, and it replaces maximal progress, employed by other timed process algebras, by the notion of resource-constrained progress. ACSR associates resource usage with time passage, and implements appropriate semantic rules to ensure that progress in the system is enforced as far as possible while simultaneous usage of a resource by distinct processes is excluded. In addition, ACSR employs the notion of priorities to arbitrate access to resources by competing processes. The chapter illustrates the use of ACSR for the schedulability analysis of a realistic real-time system.

Embedded software is increasingly distributed onto networks and structured into logical components that interact asynchronously. It is connected to sensors, actuators, human machine interfaces, and networks. Modeling has been and will be a key activity in software and systems engineering of real-time and embedded systems. Chapter 32 studies mathematical models of composed software systems and their properties, identifies and describes various basic views, and shows how they are related to composed systems. The chapter considers, in particular, models of data, states, interfaces, functionality, hierarchical composed systems, and processes as well as the concepts of abstraction, refinement, composition, and modularity. In addition, the chapter introduces features, services, and logical architectures, which are essential for modeling multifunctional embedded systems in terms of functionality and architecture of components.

The convergence of control, systems, and software engineering to model-based development of embedded systems is one of the most profound trends in technology today. Control engineers build on computing and communication technology to design robust, adaptive, and distributed control systems for operating plants. Systems engineers integrate large-scale systems of networked embedded subsystems. Software engineers implement software to satisfy requirements that are simultaneously physical and computational. Since the use of models on different levels of abstraction has been a fundamental approach in control, systems, and software engineering, it is essential to have model-based tools that can support domain-specific models specific to embedded application domains. Chapter 33 introduces and motivates metaprogramming in model-based design, which is to automatically configure and customize a class of generic modeling, model management, transformation, and analysis tools for use in a domain-specific context. It describes different metamodeling languages and metaprogramming tools.

Hardware/software codesign is a set of methodologies and algorithms for the simultaneous design of the hardware and software components of an embedded system. As more nonfunctional requirements, e.g., performance, size, power, and cost, are imposed on the end product, the design becomes harder. Hardware/software codesign allows more degrees of freedom to consider with more flexible design processes to meet complex constraints and the system requirements. Recently, it has become a major technique for the design of embedded computing systems. Chapter 34 provides an introduction to the hardware/software codesign problem and surveys existing hardware/software partitioning algorithms, cosynthesis algorithms, and CPU customization technique. The chapter also explains the basic codesign methodology.

Reliable timing estimates are important when designing, modeling, and verifying real-time and embedded systems. Execution time analysis is the problem of obtaining information about the execution time of a program or parts of a program. There are many challenges in determining the WCET of a program due to the complexity of real-time software and variability in inputs as well as advanced computer architecture features. Chapter 35 describes a variety of methods that are available to perform timing and the WCET analysis, ranging from manual measurements to automated static analysis. The chapter also describes tools for the WCET analysis and industrial experience in using these tools.

Part VII covers experiences with real-time and embedded systems. Digital hardware and software has become the driving force of innovation in many areas of technology and, in particular, in embedded systems. The center of advances in technological innovations lie in the applications of embedded systems.

Real-time embedded systems are increasingly part of the larger distributed embedded systems, such as military combat or command and control systems, emergency response systems, and telecommunications.

Middleware is being applied to these types of applications because of its ability to abstract issues of distribution, heterogeneity, and programming language from the design of systems. More and more of these systems are networked systems of systems, with heterogeneous platforms and networks, and QoS management must be adaptive due to inherent dynamic conditions associated with these systems. Chapter 36 describes issues in developing middleware for dynamic, adaptive QoS management in distributed embedded systems. The chapter also describes an experimental middleware platform with dynamic QoS management and three detailed case studies to illustrate how the QoS management techniques can be applied in practice.

Chapter 37 describes the mobile multimedia system by presenting both system and application layers of the software stack. First, at the system level, the chapter explains how the memory performance can be characterized for embedded multimedia applications. To cope with severe resource constraints, the chapter describes an approach to selectively offload computing tasks to fixed servers in the grid. Another approach presented is to balance the requirement for computational resources against the QoS. For the application layer, the chapter concentrates on MPEG-4 applications. The chapter presents the techniques of saving energy using requantization and a subdecoder as well as the effect of the wireless channel on video broadcasts with different scheduling algorithms. The chapter shows that the layered structure of fine-grained scalable MPEG video can be exploited to deliver pictures of better quality through efficient error correction.

Embedded hardware/software systems control functions in automobiles, support and assist the driver, and are essential in realizing new features for driver assistance, information, and entertainment. Systems and software engineering for embedded systems in automobiles is one of the great scientific, methodological, and technical challenges for the automotive industry. In recent years, the amount of data that needs to be sensed, stored, and processed has increased significantly in an automobile with the rapid growth of electronics and control software applications. The number of microprocessors required to implement these applications has also gone up significantly. These applications deal with safety-enhancing vehicle functions and hence with critical data having stringent requirements on freshness and involving deadline bound computations. Chapter 38 identifies the main problems, research challenges, and approaches to deal with them with emphasis on real-time issues, architecture, model-based system, and software development as well as requirements engineering. Chapter 39 studies one specific critical application, namely, adaptive cruise control (ACC). The chapter describes the data- and task-centric approaches in providing real-time data services while designing and implementing this application.

Real-Time Scheduling and Resource Management

2

Real-Time Scheduling and Resource Management

Giorgio C. Buttazzo
Scuola Superiore Sant'Anna

2.1	Introduction	2-1
	Models and Terminology	
2.2	Periodic Task Handling	2-3
	Timeline Scheduling • Fixed-Priority Scheduling • Earliest Deadline First	
2.3	Handling Aperiodic Tasks	2-7
2.4	Handling Shared Resources	2-9
	Priority Inheritance Protocol • Priority Ceiling Protocol • Schedulability Analysis	
2.5	Overload Management	2-12
	Resource Reservation • Period Adaptation	
2.6	Conclusions	2-14

2.1 Introduction

A real-time control system is a system in which the resulting performance depends not only on the correctness of the single control actions but also on the time at which the actions are produced [30]. Real-time applications span a wide range of domains including industrial plants control, automotive, flight control systems, monitoring systems, multimedia systems, virtual reality, interactive games, consumer electronics, industrial automation, robotics, space missions, and telecommunications. In these systems, a late action might cause a wrong behavior (e.g., system instability) that could even lead to a critical system failure. Hence, the main difference between a real-time task and a non-real-time task is that a real-time task must complete within a given deadline, which is the maximum time allowed for a computational process to finish its execution.

The operating system is the major architectural component responsible for ensuring a timely execution of all the tasks having some timing requirements. In the presence of several concurrent activities running on a single processor, the objective of a real-time kernel is to ensure that each activity completes its execution within its deadline. Notice that this is very different than minimizing the average response times of a set of tasks.

Depending on the consequences caused by a missed deadline, real-time activities can be classified into hard and soft tasks. A real-time task is said to be hard if missing a deadline may have catastrophic consequences on the controlled system, and is said to be soft if missing a deadline causes a performance degradation but does not jeopardize the correct system behavior. An operating system able to manage hard tasks is called a hard real-time system [11,31]. In a control application, typical hard tasks include sensory

data acquisition, detection of critical conditions, motor actuation, and action planning. Typical soft tasks include user command interpretation, keyboard input, message visualization, system status representation, and graphical activities. In general, hard real-time systems have to handle both hard and soft activities.

In spite of the large range of application domains, most of today's real-time control software is still designed using *ad hoc* techniques and heuristic approaches. Very often, control applications with stringent time constraints are implemented by writing large portions of code in assembly language, programming timers, writing low-level drivers for device handling, and manipulating task and interrupt priorities. Although the code produced by these techniques can be optimized to run very efficiently, this approach has several disadvantages. First of all, the implementation of large and complex applications in assembly language is much more difficult and time consuming than using high-level programming. Moreover, the efficiency of the code strongly depends on the programmer's ability. In addition, assembly code optimization makes a program more difficult to comprehend, complicating software maintenance. Finally, without the support of specific tools and methodologies for code and schedulability analysis, the verification of time constraints becomes practically impossible.

The major consequence of this state of practice is that the resulting control software can be highly unpredictable. If all critical time constraints cannot be verified *a priori* and the operating system does not include specific features for handling real-time tasks, the system apparently works well for a period of time, but may collapse in certain rare, but possible, situations. The consequences of a failure can sometimes be catastrophic and may injure people or cause serious damage to the environment. A trustworthy guarantee of system behavior under all possible operating conditions can only be achieved by adopting appropriate design methodologies and kernel mechanisms specifically developed for handling explicit timing constraints.

The most important property of a real-time system is not high speed, but predictability. In a predictable system, we should be able to determine in advance whether all the computational activities can be completed within their timing constraints. The deterministic behavior of a system typically depends on several factors ranging from the hardware architecture to the operating system up to the programming language used to write the application. Architectural features that have major influence on task execution include interrupts, direct memory access (DMA), cache, and prefetching mechanisms. Although such features improve the average performance of the processor, they introduce a nondeterministic behavior in process execution, prolonging the worst-case response times. Other factors that significantly affect task execution are due to the internal mechanisms used in the operating system, such as the scheduling algorithm, the synchronization mechanisms, the memory management policy, and the method used to handle I/O devices.

2.1.1 Models and Terminology

To analyze the timing behavior of a real-time system, all software activities running in the processor are modeled as a set of n real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_n\}$, where each task τ_i is a sequence of instructions that, in the absence of other activities, is cyclicly executed on different input data. Hence, a task τ_i can be considered as an infinite sequence of instances, or jobs, $\tau_{i,j}$ ($j = 1, 2, \dots$), each having a computation time $c_{i,j}$, a release time $r_{i,j}$, and an absolute deadline $d_{i,j}$. For simplicity, all jobs of the same task $\tau_{i,j}$ are assumed to have the same worst-case execution time (WCET) C_i and the same relative deadline D_i , which is the interval of time, from the job release, within which the job should complete its execution.

In addition, the following timing parameters are typically defined on real-time tasks:

$s_{i,j}$ denotes the start time of job $\tau_{i,j}$, that is, the time at which its first instruction is executed;

$f_{i,j}$ denotes the finishing time of job $\tau_{i,j}$, that is, the time at which the job completes its execution;

$R_{i,j}$ denotes the response time of job $\tau_{i,j}$, that is, the difference between the finishing time and the release time ($R_{i,j} = f_{i,j} - r_{i,j}$);

R_i denotes the maximum response time of task τ_i , that is, $R_i = \max_j R_{i,j}$.

A task is said to be *periodic* if all its jobs are released one after the other with a regular interval T_i called the task period. If the first job $\tau_{i,1}$ is released at time $r_{i,1} = \Phi_i$ (also called the task phase), the generic job $\tau_{i,k}$ is characterized by the following release times and deadlines:

$$\begin{cases} r_{i,k} = \Phi_i + (k-1)T_i \\ d_{i,k} = r_{i,k} + D_i \end{cases}$$

If the jobs are released in a nonregular fashion, the task is said to be *aperiodic*. Aperiodic tasks in which consecutive jobs are separated by a minimum interarrival time are called *sporadic*.

In real-time applications, timing constraints are usually specified on task execution, activation, or termination to enforce some performance requirements. In addition, other types of constraints can be defined on tasks, as precedence relations (for respecting some execution ordering) or synchronization points (for waiting for events or accessing mutually exclusive resources).

A schedule is said to be *feasible* if all tasks complete their execution under a set of specified constraints. A task set is said to be *schedulable* if there exists a feasible schedule for it.

Unfortunately, the problem of verifying the feasibility of a schedule in its general form has been proved to be NP-complete [15], and hence computationally intractable. However, the complexity of the feasibility analysis can be reduced for specific types of tasks and under proper (still significant) hypotheses.

In the rest of this chapter, a number of methods are presented to verify the schedulability of a task set under different assumptions. In particular, Section 2.2 treats the problem of scheduling and analyzing a set of periodic tasks; Section 2.3 addresses the issue of aperiodic service; Section 2.4 analyzes the effect of resource contention; Section 2.5 proposes some methods for handling overload conditions; and Section 2.6 concludes the chapter by presenting some open research problems.

2.2 Periodic Task Handling

Most of the control activities, such as signal acquisition, filtering, sensory data processing, action planning, and actuator control, are typically implemented as periodic tasks activated at specific rates imposed by the application requirements. When a set \mathcal{T} of n periodic tasks has to be concurrently executed on the same processor, the problem is to verify whether all tasks can complete their execution within their timing constraints. In the rest of this section, we consider the analysis for a classical cyclic scheduling approach, a fixed priority-based scheduler, and a dynamic priority algorithm based on absolute deadlines.

2.2.1 Timeline Scheduling

One of the most commonly used approaches to schedule a set of periodic tasks on a single processor consists in dividing the timeline into slots of equal length and statically allocating tasks into slots to respect the constraints imposed by the application requirements. A timer synchronizes the activation of the tasks at the beginning of each slot. The length of the slot, called the minor cycle (T_{\min}), is set equal to the greatest common divisor of the periods, and the schedule has to be constructed until the least common multiple of all the periods, called the major cycle (T_{maj}) or the hyperperiod. Note that, since the schedule repeats itself every major cycle, the schedule has to be constructed only in the first N slots, where $N = T_{\text{maj}}/T_{\min}$. For such a reason, this method is also known as a *cyclic executive*.

To verify the feasibility of the schedule, it is sufficient to check whether the sum of the computation times in each slot is less than or equal to T_{\min} . If $h(i, j)$ is a binary function, equal to 1 if τ_i is allocated in slot j , and equal to 0 otherwise, the task set is schedulable if and only if

$$\forall j = 1 \dots N, \quad \sum_{i=1}^n h(i, j)C_i < T_{\min}$$

To illustrate this method, consider the following example in which three tasks, τ_1 , τ_2 , and τ_3 , with worst-case computation times $C_1 = 10$, $C_2 = 8$, and $C_3 = 5$, have to be periodically executed on a processor

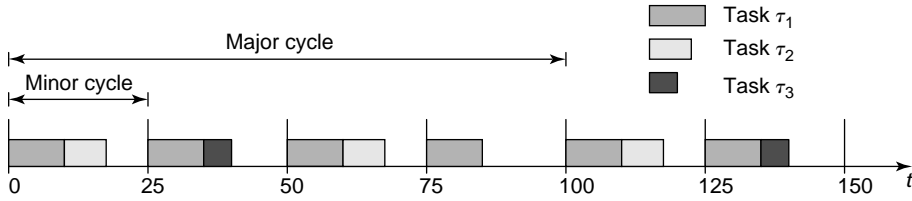


FIGURE 2.1 Example of timeline scheduling.

with periods $T_1 = 25$, $T_2 = 50$, and $T_3 = 100$ ms. From the task periods, the minor cycle results to be $T_{\min} = 25$ ms, whereas the major cycle results to be $T_{\text{maj}} = 100$ ms. To meet the required rates, it is easy to see that τ_1 needs to be executed in each slot ($T_1/T_{\min} = 1$), τ_2 every two slots ($T_2/T_{\min} = 2$), and τ_3 every four slots ($T_3/T_{\min} = 4$). A possible scheduling solution for this task set is illustrated in Figure 2.1.

Notice that the schedule produced by this method is feasible because

$$\begin{cases} C_1 + C_2 = 18 \leq 25 \text{ ms} \\ C_1 + C_3 = 15 \leq 25 \text{ ms} \end{cases}$$

The major relevant advantage of timeline scheduling is its simplicity. In fact, the method can be easily implemented by coding the major cycle into a big loop containing the task calls as they appear in the schedule. Then, a synchronization primitive has to be inserted before the task calls relative to each minor cycle and a timer must be programmed to generate an interrupt with a period equal to T_{\min} , so triggering the execution of the tasks allocated in each slot. Since the task sequence is not generated by a scheduling algorithm, but directly coded in a program, there are no context switches, so the runtime overhead is very low. Moreover, the sequence of tasks in the schedule is always the same and is not affected by jitter. In spite of these advantages, however, timeline scheduling also has some problems. For example, it is very fragile during overload conditions. In fact, if a task does not terminate at the minor cycle boundary and it is not aborted, it can cause a domino effect on the other tasks, breaking the entire schedule (timeline break). In contrast, if the failing task is aborted, it may leave a shared resource in an inconsistent state, so jeopardizing the correct system behavior.

Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase in its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch. Considering the previous example, if task τ_2 is updated to τ'_2 and the code is changed so that $C_1 + C'_2 > 25$ ms, then task τ'_2 must be divided into two or more pieces to be allocated in the available intervals in the timeline. Changing a task period may cause even more radical changes in the schedule. For example, if the period of task τ_2 changes from 50 to 40 ms, the previous schedule is not valid anymore, because the new minor cycle is equal to 10 ms and the new major cycle is equal to 200 ms, so a new schedule has to be constructed in $N = 20$ slots. Finally, another limitation of the timeline scheduling is that it is difficult to handle aperiodic activities efficiently without changing the task sequence. The problems outlined above can be solved by using priority-based scheduling algorithms.

2.2.2 Fixed-Priority Scheduling

The most common priority-based method for scheduling a set of periodic tasks is the rate-monotonic (RM) algorithm, which assigns each task a priority directly proportional to its activation frequency, so that tasks with shorter period have higher priority. Since a period is usually kept constant for a task, the RM algorithm implements a fixed-priority assignment in the sense that task priorities are decided at task creation and remain unchanged for the entire application run. RM is typically preemptive, although it can also be used in a nonpreemptive mode. In 1973, Liu and Layland [22] showed that RM is optimal among

all static scheduling algorithms in the sense that if a task set is not schedulable by RM, then the task set cannot be feasibly scheduled by any other fixed priority assignment. Another important result proved by the same authors is that a set $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ of n periodic tasks is schedulable by RM if

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \quad (2.1)$$

The quantity $U = \sum_{i=1}^n \frac{C_i}{T_i}$ represents the processor utilization factor and denotes the fraction of time used by the processor to execute the entire task set. The right-hand term in Equation 2.1 decreases with n and, for large n , it tends to the following limit value:

$$\lim_{n \rightarrow \infty} n(2^{1/n} - 1) = \ln 2 \simeq 0.69 \quad (2.2)$$

The Liu and Layland test gives only a sufficient condition for the schedulability of a task set under the RM algorithm, meaning that, if Equation 2.1 is satisfied, then the task set is certainly schedulable, but if Equation 2.1 is not satisfied nothing can be said unless $U > 1$.

The schedulability of a task set under RM can also be checked using the hyperbolic test [6], according to which a task set is schedulable by RM if

$$\prod_{i=1}^n \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \quad (2.3)$$

Although only sufficient, the hyperbolic test is more precise than the Liu and Layland one in the sense that it is able to discover a higher number of schedulable task sets. Their difference can be better appreciated by representing the corresponding feasibility regions in the utilization space, denoted as the U -space. Here, the Liu and Layland bound (LL-bound) for RM is represented by an n -dimensional plane, which intersects each axis in $U_{\text{lub}}(n) = n(2^{1/n} - 1)$. All points below such a plane represent periodic task sets that are feasible by RM. The hyperbolic bound (H-bound) expressed by Equation 2.3 is represented by an n -dimensional hyperbolic surface tangent to the RM plane and intersecting the axes for $U_i = 1$. The hyperplane intersecting each axes in $U_i = 1$, denoted as the earliest deadline first (EDF)-bound, represents the limit of the feasibility region, above which any task set cannot be scheduled by any algorithm. Figure 2.2 illustrates such bounds for $n = 2$. From the plots, it is clear that the feasibility region below the H-bound is larger than that below the LL-bound, and the gain is given by the dark gray area. It is worth noting that such gain (in terms of schedulability) increases as a function of n and tends to $\sqrt{2}$ for n tending to infinity.

A necessary and sufficient schedulability test for RM is possible but at the cost of a higher computational complexity. Several pseudopolynomial time exact tests have been proposed in the real-time literature following different approaches [3,5,17,19]. For example, the method proposed by Audsley et al. [3], known as the response time analysis, consists in computing the worst-case response time R_i of each periodic task and then verifying that it does not exceed its relative deadline D_i . The worst-case response time of a task is derived by summing its computation time and the interference caused by tasks with higher priority:

$$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \quad (2.4)$$

where $hp(i)$ denotes the set of tasks having priority higher than τ_i and $\lceil x \rceil$ the ceiling of a rational number, that is, the smallest integer greater than or equal to x . The equation above can be solved by an iterative approach, starting with $R_i(0) = C_i$ and terminating when $R_i(s) = R_i(s - 1)$. If $R_i(s) > D_i$ for some task, the iteration is stopped and the task set is declared unschedulable by RM.

All exact tests are more general than those based on the utilization because they also apply to tasks with relative deadlines less than or equal to periods. In this case, however, the scheduling algorithm that

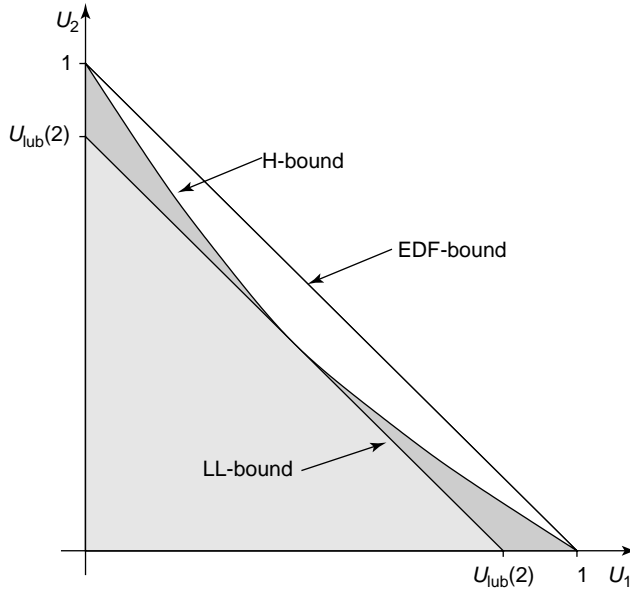


FIGURE 2.2 Schedulability bounds for RM and EDF in the utilization space.

achieves the best performance in terms of schedulability is the one that assigns priorities to tasks based on their relative deadlines, known as deadline monotonic (DM) [21]. According to DM, at each instant the processor is assigned to the task with the shortest relative deadline. In priority-based kernels, this is equivalent to assigning each task a priority P_i inversely proportional to its relative deadline. Since D_i is fixed for each task, DM is classified as a fixed-priority scheduling algorithm.

The major problem of fixed-priority scheduling is that, to achieve a feasible schedule, the processor cannot be fully utilized, except for the specific case in which the tasks have harmonic period relations (i.e., for any pair of tasks, one of the periods must be the multiple of the other). In the worst case, the maximum processor utilization that guarantees feasibility is about 0.69, as given by Equation 2.2. This problem can be overcome by dynamic priority scheduling schemes.

2.2.3 Earliest Deadline First

The most common dynamic priority scheme for real-time scheduling is the EDF algorithm, which orders the ready tasks based on their absolute deadline. According to EDF, a task receives the highest priority if its deadline is the earliest among those of the ready tasks. Since the absolute deadline changes from job to job in the same task, EDF is considered a dynamic priority algorithm. The EDF algorithm is typically preemptive in the sense that a newly arrived task preempts the running task if its absolute deadline is shorter. However, it can also be used in a nonpreemptive fashion.

EDF is more general than RM, since it can be used to schedule both periodic and aperiodic task sets, because the selection of a task is based on the value of its absolute deadline, which can be defined for both types of tasks. In 1973, Liu and Layland [22] proved that a set $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ of n periodic tasks is schedulable by EDF if and only if

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad (2.5)$$

Later, Dertouzos [14] showed that EDF is optimal among all online algorithms, meaning that if a task set is not schedulable by EDF, then it cannot be scheduled by any other algorithm. Note that Equation 2.5

provides a necessary and sufficient condition to verify the feasibility of the schedule. Thus, if it is not satisfied, no algorithm can produce a feasible schedule for that task set.

The dynamic priority assignment allows EDF to exploit the full CPU capacity, reaching up to 100% of processor utilization. When the task set has a utilization factor less than one, the residual fraction can be efficiently used to handle aperiodic requests activated by external events. In general, compared with fixed-priority schemes, EDF is superior in many aspects [7], and also generates a lower number of context switches, thus causing less runtime overhead. Finally, using a suitable kernel mechanism for time representation [13], EDF can be effectively implemented even in small microprocessors [8] for increasing system utilization and achieving a timely execution of periodic and aperiodic tasks.

Under EDF, the schedulability analysis for periodic task sets with deadlines less than periods is based on the processor demand criterion [4]. According to this method, a task set is schedulable by EDF if and only if, in every interval of length L , the overall computational demand is no greater than the available processing time, that is, if and only if $U < 1$ and

$$\forall L > 0, \quad \sum_{i=1}^n \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \leq L \quad (2.6)$$

where $\lfloor x \rfloor$ denotes the floor of a rational number, that is, the highest integer less than or equal to x . Notice that, in practice, the number of points in which the test has to be performed can be limited to the set of absolute deadlines not exceeding $t_{\max} = \min\{L^*, H\}$, where H is the hyperperiod and

$$L^* = \max \left\{ D_1, \dots, D_n, \frac{\sum_{i=1}^n (T_i - D_i) C_i / T_i}{1 - U} \right\} \quad (2.7)$$

2.3 Handling Aperiodic Tasks

Although in a real-time system most acquisition and control tasks are periodic, there exist computational activities that must be executed only at the occurrence of external events (typically signaled through interrupts), which may arrive at irregular intervals of time. When the system must handle aperiodic requests of computation, we have to balance two conflicting interests: on the one hand, we would like to serve an event as soon as possible to improve system responsiveness; on the other, we do not want to jeopardize the schedulability of periodic tasks. If aperiodic activities are less critical than periodic tasks, then the objective of a scheduling algorithm should be to minimize their response time, while guaranteeing that all periodic tasks (although being delayed by the aperiodic service) complete their executions within their deadlines. If some aperiodic task has a hard deadline, we should try to guarantee its timely completion offline. Such a guarantee can only be done by assuming that aperiodic requests, although arriving at irregular intervals, do not exceed a maximum given frequency, that is, they are separated by a minimum interarrival time. An aperiodic task characterized by a minimum interarrival time is called a sporadic task. Let us consider an example in which an aperiodic job J_a of 3 units of time must be scheduled by RM along with two periodic tasks, having computation times $C_1 = 1$, $C_2 = 3$ and periods $T_1 = 4$, $T_2 = 6$, respectively. As shown in Figure 2.3, if the aperiodic request is serviced immediately (i.e., with a priority higher than that assigned to periodic tasks), then task τ_2 will miss its deadline.

The simplest technique for managing aperiodic activities while preserving the guarantee for periodic tasks is to schedule them in background. This means that an aperiodic task executes only when the processor is not busy with periodic tasks. The disadvantage of this solution is that, if the computational load due to periodic tasks is high, the residual time left for aperiodic execution can be insufficient for satisfying their timing constraints. Considering the same task set as before, Figure 2.4 illustrates how job J_a is handled by a background service.

The response time of aperiodic tasks can be improved by handling them through an aperiodic server dedicated to their execution. As any other periodic task, a server is characterized by a period T_s and

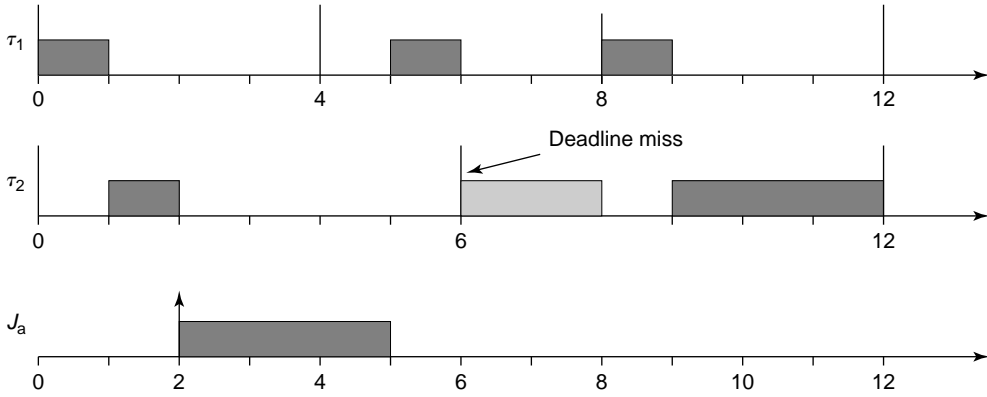


FIGURE 2.3 Immediate service of an aperiodic task. Periodic tasks are scheduled by RM.

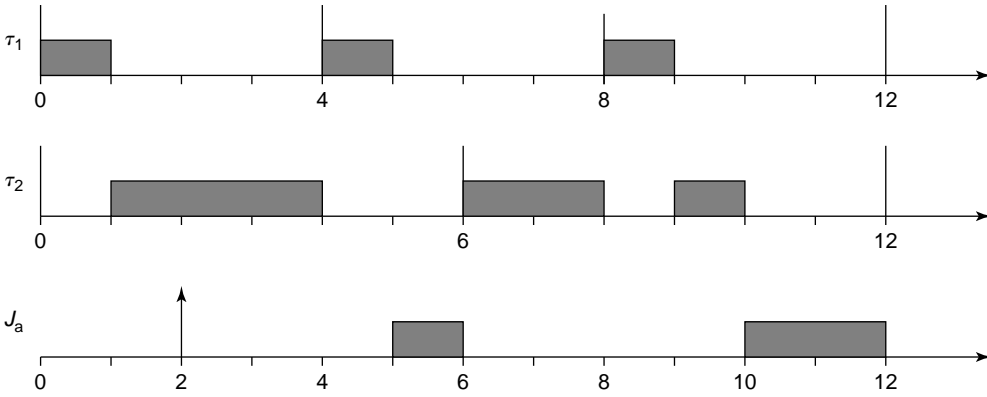


FIGURE 2.4 Background service of an aperiodic task. Periodic tasks are scheduled by RM.

an execution time C_s , called the server capacity (or budget). In general, the server is scheduled using the algorithm adopted for the periodic tasks and, once activated, it starts serving the pending aperiodic requests within the limit of its current capacity. The order of service of the aperiodic requests is independent of the scheduling algorithm used for the periodic tasks, and it can be a function of the arrival time, computation time, or deadline. During the past years, several aperiodic service algorithms have been proposed in the real-time literature, differing in performance and complexity. Among the fixed-priority algorithms, we mention the polling server and the deferrable server [20,32], the sporadic server [27], and the slack stealer [18]. Among those servers using dynamic priorities (which are more efficient on the average) we recall the dynamic sporadic server [16,28], the total bandwidth server [29], the tunable bandwidth server [10], and the constant bandwidth server (CBS) [1]. To clarify the idea behind an aperiodic server, Figure 2.5 illustrates the schedule produced, under EDF, by a dynamic deferrable server with capacity $C_s = 1$ and period $T_s = 4$. We note that, when the absolute deadline of the server is equal to the one of a periodic task, the priority is given to the server to enhance aperiodic responsiveness. We also observe that the same task set would not be schedulable under a fixed-priority system.

Although the response time achieved by a server is less than that achieved through the background service, it is not the minimum possible. The minimum response time can be obtained with an optimal server (TB*), which assigns each aperiodic request the earliest possible deadline that still produces a feasible EDF schedule [10]. The schedule generated by the optimal TB* algorithm is illustrated in Figure 2.6, where the minimum response time for job J_a is equal to 5 units of time (obtained by assigning the job a deadline $d_a = 7$). As for all the efficient solutions, better performance is achieved at the price of a larger

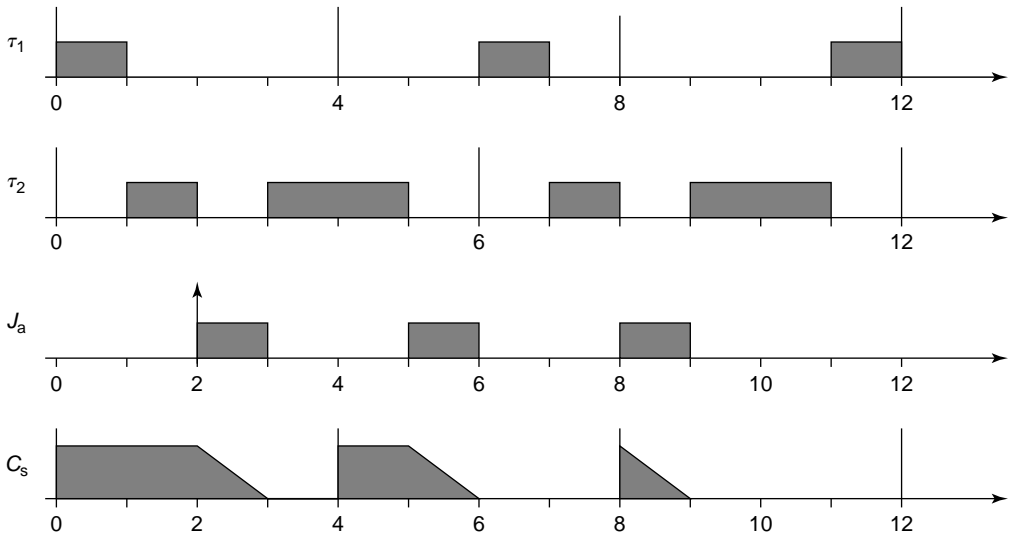


FIGURE 2.5 Aperiodic service performed by a dynamic deferrable server. Periodic tasks, including the server, are scheduled by EDF. C_s is the remaining budget available for J_a .

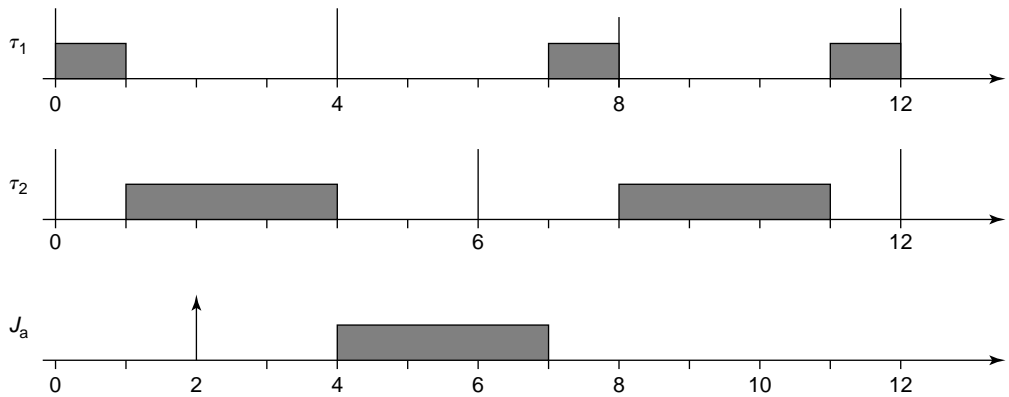


FIGURE 2.6 Optimal aperiodic service under EDF.

runtime overhead (due to the complexity of computing the minimum deadline). However, adopting a variant of the algorithm, called the tunable bandwidth server [10], overhead cost and performance can be balanced to select the best service method for a given real-time system. An overview of the most common aperiodic service algorithms (both under fixed and dynamic priorities) can be found in Ref. 11.

2.4 Handling Shared Resources

When two or more tasks interact through shared resources (e.g., shared memory buffers), the direct use of classical synchronization mechanisms, such as semaphores or monitors, can cause a phenomenon known as priority inversion: a high-priority task can be blocked by a low-priority task for an unbounded interval of time. Such a blocking condition can create serious problems in safety critical real-time systems, since

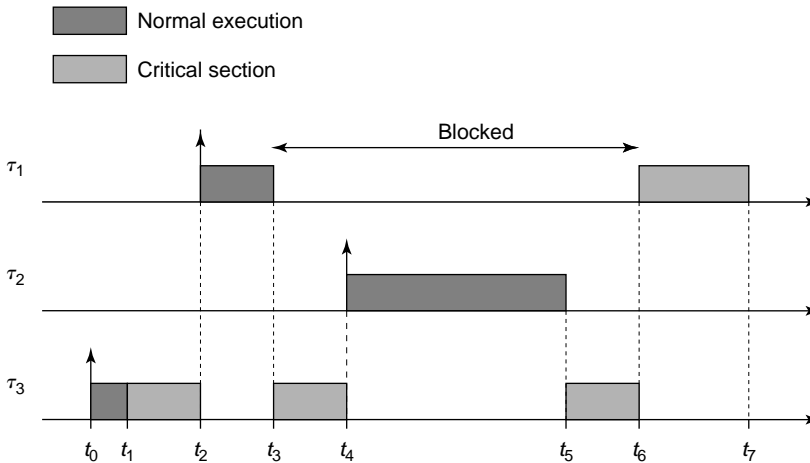


FIGURE 2.7 Example of priority inversion.

it can cause deadlines to be missed. For example, consider three tasks, τ_1 , τ_2 , and τ_3 , having decreasing priority (τ_1 is the task with highest priority), and assume that τ_1 and τ_3 share a data structure protected by a binary semaphore S . As shown in Figure 2.7, suppose that at time t_1 task τ_3 enters its critical section, holding semaphore S . During the execution of τ_3 at time t_2 , assume τ_1 becomes ready and preempts τ_3 .

At time t_3 , when τ_1 tries to access the shared resource, it is blocked on semaphore S , since the resource is used by τ_3 . Since τ_1 is the highest-priority task, we would expect it to be blocked for an interval no longer than the time needed by τ_3 to complete its critical section. Unfortunately, however, the maximum blocking time for τ_1 can become much larger. In fact, task τ_3 , while holding the resource, can be preempted by medium-priority tasks (such as τ_2), which will prolong the blocking interval of τ_1 for their entire execution! The situation illustrated in Figure 2.7 can be avoided by simply preventing preemption inside critical sections. This solution, however, is appropriate only for very short critical sections, because it could introduce unnecessary delays in high-priority tasks. For example, a low-priority task inside a long critical section could prevent the execution of high-priority tasks even though they do not share any resource. A more efficient solution is to regulate the access to shared resources through the use of specific concurrency control protocols [24] designed to limit the priority inversion phenomenon.

2.4.1 Priority Inheritance Protocol

An elegant solution to the priority inversion phenomenon caused by mutual exclusion is offered by the Priority Inheritance Protocol [26]. Here, the problem is solved by dynamically modifying the priorities of tasks that cause a blocking condition. In particular, when a task τ_a blocks on a shared resource, it transmits its priority to the task τ_b that is holding the resource. In this way, τ_b will execute its critical section with the priority of task τ_a . In general, τ_b inherits the highest priority among the tasks it blocks. Moreover, priority inheritance is transitive, thus if task τ_c blocks τ_b , which in turn blocks τ_a , then τ_c will inherit the priority of τ_a through τ_b .

Figure 2.8 illustrates how the schedule shown in Figure 2.7 is changed when resources are accessed using the Priority Inheritance Protocol. Until time t_3 the system evolution is the same as the one shown in Figure 2.7. At time t_3 , the high-priority task τ_1 blocks after attempting to enter the resource held by τ_3 (direct blocking). In this case, however, the protocol imposes that τ_3 inherits the maximum priority among the tasks blocked on that resource, thus it continues the execution of its critical section at the priority of τ_1 . Under these conditions, at time t_4 , task τ_2 is not able to preempt τ_3 , hence it blocks until the resource is released (push-through blocking).

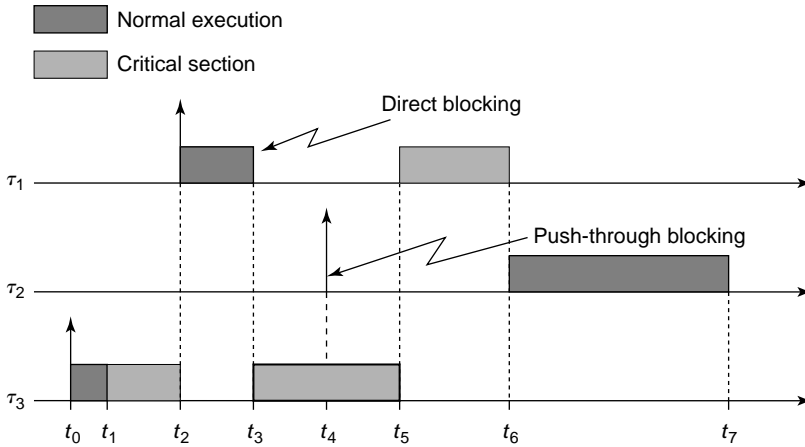


FIGURE 2.8 Schedule produced using priority inheritance on the task set of Figure 2.7.

In other words, although τ_2 has a nominal priority greater than τ_3 it cannot execute because τ_3 inherited the priority of τ_1 . At time t_5 , τ_3 exits its critical section, releases the semaphore, and recovers its nominal priority. As a consequence, τ_1 can proceed until its completion, which occurs at time t_6 . Only then τ_2 can start executing.

The Priority Inheritance Protocol has the following property [26]:

Given a task τ_i , let l_i be the number of tasks with lower priority sharing a resource with a task with priority higher than or equal to τ_i , and let r_i be the number of resources that could block τ_i . Then, τ_i can be blocked for at most the duration of $\min(l_i, r_i)$ critical sections.

Although the Priority Inheritance Protocol limits the priority inversion phenomenon, the maximum blocking time for high-priority tasks can still be significant due to possible chained blocking conditions. Moreover, deadlock can occur if semaphores are not properly used in nested critical sections.

2.4.2 Priority Ceiling Protocol

The Priority Ceiling Protocol [26] provides a better solution for the priority inversion phenomenon, also avoiding chained blocking and deadlock conditions. The basic idea behind this protocol is to ensure that, whenever a task τ enters a critical section, its priority is the highest among those that can be inherited from all the lower-priority tasks that are currently suspended in a critical section. If this condition is not satisfied, τ is blocked and the task that is blocking τ inherits τ 's priority. This idea is implemented by assigning each semaphore a priority ceiling equal to the highest priority of the tasks using that semaphore. Then, a task τ is allowed to enter a critical section only if its priority is strictly greater than all priority ceilings of the semaphores held by the other tasks. As for the Priority Inheritance Protocol, the inheritance mechanism is transitive. The Priority Ceiling Protocol, besides avoiding chained blocking and deadlocks, has the property that each task can be blocked for at most the duration of a single critical section.

2.4.3 Schedulability Analysis

The importance of the protocols for accessing shared resources in a real-time system derives from the fact that they can bound the maximum blocking time experienced by a task. This is essential for analyzing the schedulability of a set of real-time tasks interacting through shared buffers or any other nonpreemptable resource, for example, a communication port or bus. To verify the schedulability of task τ_i using the processor utilization approach, we need to consider the utilization factor of task τ_i , the interference caused by the higher-priority tasks, and the blocking time caused by lower-priority tasks. If B_i is the maximum

blocking time that can be experienced by task τ_i , then the sum of the utilization factors due to these three causes cannot exceed the least upper bound of the scheduling algorithm, that is

$$\forall i, \quad 1 \leq i \leq n, \quad \sum_{k \in hp(i)} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1) \quad (2.8)$$

where $hp(i)$ denotes the set of tasks with priority higher than τ_i . The same test is valid for both the protocols described above, the only difference being the amount of blocking that each task may experience.

2.5 Overload Management

This section deals with the problem of scheduling real-time tasks in overload conditions; that is, in those critical situations in which the computational demand requested by the task set exceeds the time available on the processor, and hence not all tasks can complete within their deadlines.

A transient overload condition can occur for the simultaneous arrival of asynchronous events, or because some execution time exceeds the value for which it has been guaranteed. When a task executes more than expected, it is said to overrun. In a periodic task system, the overload can become permanent after the activation of a new periodic task (if $U > 1$), or after a task increases its activation rate to react to some change in the environment. In such a situation, computational activities start to accumulate in the system's queues (which tend to become longer and longer if the overload persists), and tasks' response times tend to increase indefinitely. In the following sections, we present two effective methods for dealing with transient and permanent overload conditions.

2.5.1 Resource Reservation

Resource reservation is a general technique used in real-time systems for limiting the effects of overruns in tasks with variable computation times. According to this method, each task is assigned a fraction of the available resources just enough to satisfy its timing constraints. The kernel, however, must prevent each task to consume more than the allocated amount to protect the other tasks in the systems (temporal protection). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation independent of the behavior of the other tasks.

A simple and effective mechanism for implementing temporal protection in a real-time system is to reserve, for each task τ_i , a specified amount Q_i of CPU time in every interval P_i . Some authors [25] tend to distinguish between *hard* and *soft* reservations, where a hard reservation allows the reserved task to execute *at most* for Q_i units of time every P_i , whereas a soft reservation guarantees that the task executes *at least* for Q_i time units every P_i , allowing it to execute more if there is some idle time available.

A resource reservation technique for fixed-priority scheduling was first presented in Ref. 23. According to this method, a task τ_i is first assigned a pair (Q_i, P_i) (denoted as a *CPU capacity reserve*) and then it is enabled to execute as a real-time task for Q_i units of time every interval of length P_i . When the task consumes its reserved quantum Q_i , it is blocked until the next period, if the reservation is hard, or it is scheduled in background as a nonreal-time task, if the reservation is soft. If the task is not finished, it is assigned another time quantum Q_i at the beginning of the next period and it is scheduled as a real-time task until the budget expires, and so on. In this way, a task is *reshaped* so that it behaves like a periodic real-time task with known parameters (Q_i, P_i) and can be properly scheduled by a classical real-time scheduler.

Under EDF, temporal protection can be efficiently implemented by handling each task through a dedicated CBS [1,2]. The behavior of the server is tuned by two parameters (Q_i, P_i) , where Q_i is the *server maximum budget* and P_i is the *server period*. The ratio $U_i = Q_i/P_i$ is denoted as the *server bandwidth*. At each instant, two state variables are maintained for each server: the server deadline d_i and the actual server budget q_i . Each job handled by a server is scheduled using the current server deadline and whenever the

server executes a job, the budget q_i is decreased by the same amount. At the beginning $d_i = q_i = 0$. Since a job is not activated while the previous one is active, the CBS algorithm can be formally defined as follows:

1. When a job $\tau_{i,j}$ arrives, if $q_i \geq (d_i - r_{i,j})U_i$, it is assigned a server deadline $d_i = r_{i,j} + P_i$ and q_i is recharged at the maximum value Q_i , otherwise the job is served with the current deadline using the current budget.
2. When $q_i = 0$, the server budget is recharged at the maximum value Q_i and the server deadline is postponed at $d_i = d_i + P_i$. Notice that there are no finite intervals of time in which the budget is equal to zero.

As shown in Ref. 2, if a task τ_i is handled by a CBS with bandwidth U_i it will never demand more than U_i independent of the actual execution time of its jobs. As a consequence, possible overruns occurring in the served task do not create extra interference in the other tasks, but only delay τ_i .

Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system performance becomes quite dependent on a correct bandwidth allocation. In fact, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. In contrast, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

2.5.2 Period Adaptation

If a permanent overload occurs in a periodic task set, the load can be reduced by enlarging task periods to suitable values, so that the total workload can be kept below a desired threshold. The possibility of varying tasks' rates increases the flexibility of the system in handling overload conditions, providing a more general admission control mechanism. For example, whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request.

An effective method to change task periods as a function of the desired workload is the elastic framework [9,12], according to which each task is considered as flexible as a spring, whose utilization can be modified by changing its period within a specified range. The advantage of the elastic model with respect to the other methods proposed in the literature is that a new period configuration can easily be determined online as a function of the elastic coefficients, which can be set to reflect tasks' importance. Once elastic coefficients are defined based on some design criterion, periods can be quickly computed online depending on the current workload and the desired load level.

More specifically, each task is characterized by four parameters: a worst-case computation time C_i , a minimum period $T_{i,\min}$ (considered as a nominal period), a maximum period $T_{i,\max}$, and an elastic coefficient E_i . The elastic coefficient specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration: the greater the E_i , the more elastic the task. Hence, an elastic task is denoted by

$$\tau_i(C_i, T_{i,\min}, T_{i,\max}, E_i)$$

The actual period of task τ_i is denoted by T_i and is constrained to be in the range $[T_{i,\min}, T_{i,\max}]$. Moreover, $U_{i,\max} = C_i/T_{i,\min}$ and $U_{i,\min} = C_i/T_{i,\max}$ denote the maximum and minimum utilization of τ_i , whereas $U_{\max} = \sum_{i=1}^n U_{i,\max}$ and $U_{\min} = \sum_{i=1}^n U_{i,\min}$ denote the maximum and minimum utilization of the task set.

Assuming tasks are scheduled by the EDF algorithm [22], if $U_{\max} \leq 1$, all tasks can be activated at their minimum period $T_{i,\min}$, otherwise the elastic algorithm is used to adapt their periods to T_i such that $\sum \frac{C_i}{T_i} = U_d \leq 1$, where U_d is some desired utilization factor. This can be done as in a linear spring system, where springs are compressed by a force F (depending on their elasticity) up to a desired total length. The concept is illustrated in Figure 2.9. It can be easily shown (see Ref. 9 for details) that a solution always exists if $U_{\min} \leq U_d$.

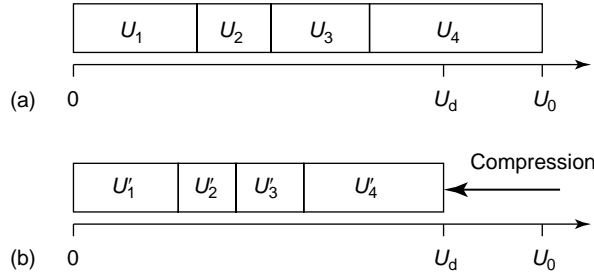


FIGURE 2.9 Compressing the utilizations of a set of elastic tasks.

As shown in Ref. 9, in the absence of period constraints (i.e., if $T_{\max} = \infty$), the utilization U_i of each compressed task can be computed as follows:

$$\forall i, \quad U_i = U_{i_{\max}} - (U_{\max} - U_d) \frac{E_i}{E_{\text{tot}}} \quad (2.9)$$

where

$$E_{\text{tot}} = \sum_{i=1}^n E_i \quad (2.10)$$

In the presence of period constraints, the compression algorithm becomes iterative with complexity $O(n^2)$, where n is the number of tasks. The same algorithm can be used to reduce the periods when the overload is over, so adapting task rates to the current load condition to better exploit the computational resources.

2.6 Conclusions

This chapter surveyed some kernel methodologies aimed at enhancing the efficiency and the predictability of real-time control applications. In particular, some scheduling algorithms and analysis techniques have been presented for periodic and aperiodic task sets illustrating that dynamic priority scheduling schemes achieve better resource exploitation with respect to fixed-priority algorithms.

It has been shown that, when tasks interact through shared resources, the use of critical sections may cause a priority inversion phenomenon, where high-priority tasks can be blocked by low-priority tasks for an unbounded interval of time. Two concurrency control protocols (namely the Priority Inheritance and Priority Ceiling protocols) have been described to avoid this problem. Each method allows to bound the maximum blocking time for each task and can be analyzed offline to verify the feasibility of the schedule within the timing constraints imposed by the application.

Finally, some overload management techniques have been described to keep the system workload below a desired threshold and deal with dangerous peak load situations that could degrade system performance. In the presence of soft real-time activities with extremely variable computation requirements (as those running in multimedia systems), *resource reservation* is an effective methodology for limiting the effects of execution overruns and protecting the critical activities from an unbounded interference. Moreover, it allows to guarantee a task in isolation independent of the behavior of the other tasks. Implementing resource reservation, however, requires a specific support from the kernel, which has to provide a tracing mechanism to monitor the actual execution of each job. To prevent permanent overload conditions caused by excessive periodic load, elastic scheduling provides a simple and effective way to shrink task utilizations up to a desired load.

Next generation embedded systems are required to work in dynamic environments where the characteristics of the computational load cannot always be predicted in advance. Still timely responses to

events have to be provided within precise timing constraints to guarantee a desired level of performance. The combination of real-time features in tasks with dynamic behavior, together with cost and resource constraints, creates new problems to be addressed in the design of such systems at different architecture levels. The classical worst-case design approach, typically adopted in hard real-time systems to guarantee timely responses in all possible scenarios, is no longer acceptable in highly dynamic environments, because it would waste resources and prohibitively increase the cost. Instead of allocating resources for the worst case, smarter techniques are needed to sense the current state of the environment and react as a consequence. This means that, to cope with dynamic environments, a real-time system must be *adaptive*, that is, it must be able to adjust its internal strategies in response to a change in the environment to keep the system performance at a desired level or, if this is not possible, degrade it in a controlled fashion.

References

1. L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
2. L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," *Real-Time Systems*, Vol. 27, No. 2, pp. 123–167, July 2004.
3. N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," *Software Engineering Journal*, Vol. 8, No. 5, pp. 284–292, September 1993.
4. S. K. Baruah, R. R. Howell, and L. E. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, Vol. 2, No. 4, pp. 301–304, 1990.
5. E. Bini and G. Buttazzo, "Schedulability Analysis of Periodic Fixed Priority Systems," *IEEE Transactions on Computers*, Vol. 53, No. 11, pp. 1462–1473, November 2004.
6. E. Bini, G. C. Buttazzo, and G. M. Buttazzo, "Rate Monotonic Analysis: The Hyperbolic Bound," *IEEE Transactions on Computers*, Vol. 52, No. 7, pp. 933–942, July 2003.
7. G. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems*, Vol. 28, pp. 1–22, 2005.
8. G. Buttazzo and P. Gai, "Efficient EDF Implementation for Small Embedded Systems," *Proceedings of the 2nd Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2006)*, Dresden, Germany, July 2006.
9. G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Transactions on Computers*, Vol. 51, No. 3, pp. 289–302, March 2002.
10. G. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments," *IEEE Transactions on Computers*, Vol. 48, No. 10, pp. 1035–1052, October 1999.
11. G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Boston, MA, 1997.
12. G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control," *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 286–295, December 1998.
13. A. Carlini and G. Buttazzo, "An Efficient Time Representation for Real-Time Embedded Systems," *Proceedings of the ACM Symposium on Applied Computing (SAC 2003)*, Melbourne, FL, pp. 705–712, March 9–12, 2003.
14. M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Information Processing*, Vol. 74, North-Holland Publishing Company, pp. 807–813, 1974.
15. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.
16. T. M. Ghazalie and T. P. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *Real-Time Systems*, Vol. 9, No. 1, pp. 31–67, 1995.

17. M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, Vol. 29, No. 5, pp. 390–395, 1986.
18. J. P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.
19. J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.
20. J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261–270, 1987.
21. J. Leung and J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, Vol. 2, No. 4, pp. 237–250, 1982.
22. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 40–61, 1973.
23. C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, May 1994.
24. R. Rajkumar, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Boston, MA, 1991.
25. R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, San José, CA, January 1998.
26. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175–1185, 1990.
27. B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time System," *Journal of Real-Time Systems*, Vol. 1, pp. 27–60, June 1989.
28. M. Spuri and G. C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling," *Proceedings of IEEE Real-Time System Symposium*, San Juan, Puerto Rico, December 1994.
29. M. Spuri and G. C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, Vol. 10, No. 2, pp. 179–210, 1996.
30. J. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, Vol. 21, No. 10, pp. 10–19, October 1988.
31. J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, Vol. 28, No. 6, pp. 16–25, June 1995.
32. J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Transactions on Computers*, Vol. 44, No. 1, pp. 73–91, January 1995.

3

Schedulability Analysis of Multiprocessor Sporadic Task Systems

3.1	Introduction	3-1
3.2	Definitions and Models	3-2
	Sporadic Task System • A Classification of Scheduling Algorithms	
3.3	Dynamic Priority Scheduling	3-4
	Partitioned Scheduling • Global Scheduling	
3.4	Fixed Job-Priority Scheduling	3-5
	Partitioned Scheduling • Global Scheduling	
3.5	Fixed Task-Priority Scheduling	3-10
	Partitioned Scheduling • Global Scheduling	
3.6	Relaxations of the Sporadic Model	3-13
	Sleeping within a Job • Task Interdependencies • Nonpreemptability • Scheduling and Task-Switching Overhead • Changes in Task System • Aperiodic Tasks	
3.7	Conclusion	3-15

Theodore P. Baker*

Florida State University

Sanjoy K. Baruah

University of North Carolina

3.1 Introduction

In this chapter, we consider the scheduling of systems of independent sporadic tasks to meet hard deadlines upon platforms comprised of several identical processors. Although this field is very new, a large number of interesting and important results have recently been obtained. We focus on presenting the results within the context of what seems to be a natural classification scheme (described in Section 3.2.2). Space considerations rule out the possibility of providing proofs of these results; instead, we provide references to primary sources where possible.

The organization of this chapter is as follows. In Section 3.2, we formally define the sporadic task model and other important concepts, and describe the classification scheme for multiprocessor scheduling algorithms that we have adopted. In each of Sections 3.3 through 3.5, we discuss results concerning one class of scheduling algorithms. The focus of Sections 3.2 through 3.5 is primarily theoretical; in Section 3.6, we briefly discuss some possible extensions to these results that may enhance their practical applicability.

*This material is based on the work supported in part by the National Science Foundation under Grant No. 0509131 and a DURIP grant from the Army Research Office.

3.2 Definitions and Models

The sporadic task model may be viewed as a set of timing constraints imposed on a more general model. A general *task* is an abstraction of a sequential thread of control, which executes an algorithm that may continue indefinitely, alternating between states where it is competing for processor time and states where it is waiting for an event. (Waiting for an event means an intentional act of the task to suspend computation, as opposed to incidental blocking such as may be imposed by contention for resources.) The computation between two wait states of a task is a *job*. The time of an event that triggers a transition from waiting to competing for execution is the *release time* of a job, and the time of the next transition to a wait state is the *completion time* of the job. The amount of processor time used by the job during this interval is its *execution time*. In hard real-time systems, a deadline is specified for a job prior to which the job's completion time must occur. The *absolute deadline* of a job is its release time plus its *relative deadline*, and the *scheduling window* of a job is the interval that begins with its release time and ends at its absolute deadline. The *response time* of a job during a particular execution of the system is defined to be the amount of elapsed time between the job's release time and its completion time. A job misses its deadline if its response time exceeds its relative deadline.

3.2.1 Sporadic Task System

Intuitively, it is impossible to guarantee that a system will not miss any deadlines if there is no upper bound on the computational demand of the system. The idea behind the sporadic task model is that if there is a lower bound on the interval between triggering events of a task and an upper bound on the amount of processor time that a task requires for each triggering event, then the computational demand of each task in any time interval is bounded.

A *sporadic task system* is a set $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ of sporadic tasks. Each *sporadic task* τ_i is characterized by a triple (p_i, e_i, d_i) , where p_i is a lower bound on the separation between release times of the jobs of the task, also known as the *period* of the task; e_i an upper bound on the execution time for each job of the task, also known as the *worst-case execution time*; and d_i the *relative deadline*, which is the length of the scheduling window of each job. A sporadic task system is said to have *implicit deadlines* if $d_i = p_i$ for every task, *constrained deadlines* if $d_i \leq p_i$ for every task, and *arbitrary deadlines* if there is no such constraint.

A special case of a sporadic task system is a *periodic task system* in which the separation between release times is required to be exactly equal to the period.

The concepts of task and system *utilization* and *density* prove useful in the analysis of sporadic task systems on multiprocessors. These concepts are defined as follows:

Utilization: The utilization u_i of a task τ_i is the ratio e_i/p_i of its execution time to its period. The total utilization $u_{\text{sum}}(\tau)$ and the largest utilization $u_{\text{max}}(\tau)$ of a task system τ are defined as follows:

$$u_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i \quad u_{\text{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (u_i)$$

Constrained density: The density δ_i of a task τ_i is the ratio e_i/d_i of its execution time to its relative deadline. The total density $\delta_{\text{sum}}(\tau)$ and the largest density $\delta_{\text{max}}(\tau)$ of a task system τ are defined as follows:

$$\delta_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \delta_i \quad \delta_{\text{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (\delta_i)$$

Generalized density: The generalized density λ_i of a task τ_i is the ratio $e_i / \min(d_i, p_i)$ of its execution time to the lesser of its relative deadline and its period. The total generalized density $\lambda_{\text{sum}}(\tau)$ and the largest generalized density $\lambda_{\text{max}}(\tau)$ of a task system τ are defined as follows:

$$\lambda_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \lambda_i \quad \lambda_{\text{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (\lambda_i)$$

An additional concept that plays a critical role in the schedulability analysis of sporadic task systems is that of *demand bound function*. For any interval length t , the demand bound function $\text{DBF}(\tau_i, t)$ of a sporadic task τ_i bounds the maximum cumulative execution requirement by jobs of τ_i that both arrive in, and have deadlines within, any interval of length t . It has been shown [12] that

$$\text{DBF}(\tau_i, t) = \max \left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1 \right) e_i \right) \quad (3.1)$$

For any sporadic task system τ , a load parameter may be defined as follows:

$$\text{load}(\tau) \stackrel{\text{def}}{=} \max_{t > 0} \left(\frac{\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t)}{t} \right) \quad (3.2)$$

3.2.2 A Classification of Scheduling Algorithms

Scheduling is the allocation of processor time to jobs. An m -processor *schedule* for a set of jobs is a partial mapping of time instants and processors to jobs. It specifies the job, if any, that is scheduled on each processor at each time instant. We require that a schedule not assign more than one processor to a job at each instant, and not assign a processor to a job before the job's release time or after the job completes.

A given schedule is *feasible* for a given job set if it provides sufficient time for each job to complete within its scheduling window, that is, if the response time of each job is less than or equal to its relative deadline. A given job set is *feasible* if there exists a feasible schedule for it. In practice, feasibility does not mean much unless there is an algorithm to compute a feasible schedule for it. A job set is *schedulable* by a given scheduling algorithm if the algorithm produces a feasible schedule.

The above definitions generalize from jobs to tasks in the natural way, subject to the constraint that the jobs of each task must be scheduled serially, in order of release time and with no overlap. A sporadic task system is feasible if there is a feasible schedule for every set of jobs that is consistent with the period, deadline, and worst-case execution time constraints of the task system, and it is schedulable by a given algorithm if the algorithm finds a feasible schedule for every such set of jobs.

A *schedulability test* for a given scheduling algorithm is an algorithm that takes as input a description of a task system and provides as output an answer to whether the system is schedulable by the given scheduling algorithm. A schedulability test is *exact* if it correctly identifies all schedulable and unschedulable task systems. It is *sufficient* if it correctly identifies all unschedulable task systems, but may misidentify some schedulable systems as being unschedulable.

For any scheduling algorithm to be useful for hard-deadline real-time applications it must have at least a sufficient schedulability test that can verify that a given job set is schedulable. The quality of the scheduling algorithm and the schedulability test are inseparable, since there is no practical difference between a job set that is not schedulable and one that cannot be proven to be schedulable.

3.2.2.1 A Classification Scheme

Given a task system and a set of processors, it is possible to divide the processors into clusters and assign each task to a cluster. A scheduling algorithm is then applied locally in each cluster, to the jobs generated by the tasks that are assigned to the cluster. Each of the multiprocessor schedulability tests reported here focus on one of the following two extremes of clustering:

1. *Partitioned scheduling*: Each processor is a cluster of size one. Historically, this has been the first step in extending single-processor analysis techniques to the multiprocessor domain.
2. *Global scheduling*: There is only one cluster containing all the processors.

Observe that partitioned scheduling may be considered as a two-step process: (i) the tasks are partitioned among the processors prior to runtime; and (ii) the tasks assigned to each processor are scheduled individually during runtime, using some local (uniprocessor) scheduling algorithm.

Runtime scheduling algorithms are typically implemented as follows: at each time instant, a *priority* is assigned to each job that has been released but not yet completed, and the available processors are allocated to the highest-priority jobs. Depending upon the restrictions that are placed upon the manner in which priorities may be assigned to jobs, we distinguish in this chapter between three classes of scheduling algorithms:

1. *Fixed task-priority* (FTP) scheduling: All jobs generated by a task are assigned the same priority.
2. *Fixed job-priority* (FJP) scheduling: Different jobs of the same task may have different priorities. However, the priority of each job may not change between its release time and its completion time.
3. *Dynamic priority* (DP) scheduling: Priorities of jobs may change between their release times and their completion times.

It is evident from the definitions that FJP scheduling is a generalization of FTP scheduling, and DP scheduling is a generalization of FJP scheduling.

3.2.2.2 Predictable Scheduling Algorithms

At first glance, proving that a given task system is schedulable by a given algorithm seems very difficult. The sporadic constraint only requires that the release times of any two successive jobs of a task τ_i be separated by at least p_i time units and that the execution time of every job be in the range $[0, e_i]$. Therefore, there are infinitely many sets of jobs, finite and infinite, that may be generated by any given sporadic task set.

As observed above, partitioned scheduling algorithms essentially reduce a multiprocessor scheduling problem to a series of uniprocessor scheduling algorithms (one to each processor). Results from uniprocessor scheduling theory [29] may hence be used to identify the “worst-case” sequence of jobs that can be generated by a given sporadic task system, such that the task system is schedulable if and only if this worst-case sequence of jobs is successfully scheduled.

Ha and Liu [25,26] found a way to reduce the complexity of schedulability testing for global multiprocessor scheduling, through the concept of *predictable* scheduling algorithms. Consider any two sets \mathcal{J} and \mathcal{J}' of jobs that only differ in the execution times of the jobs, with the execution times of jobs in \mathcal{J}' being less than or equal to the execution times of the corresponding jobs in \mathcal{J} . A scheduling algorithm is defined to be predictable if, in scheduling such \mathcal{J} and \mathcal{J}' (separately), the completion time of each job in \mathcal{J}' is always no later than the completion time of the corresponding job in \mathcal{J} . That is, with a predictable scheduling algorithm it is sufficient, for the purpose of bounding the worst-case response time of a task or proving schedulability of a task set, to look just at the jobs of each task whose actual execution times are equal to the task’s worst-case execution time.

Not every scheduling algorithm is predictable in this sense. There are well-known examples of scheduling anomalies in which shortening the execution time of one or more jobs in a task set that is schedulable by a given algorithm can make the task set unschedulable. In particular, such anomalies have been observed for nonpreemptive priority-driven scheduling algorithms on multiprocessors. Ha and Liu proved that all preemptable FTP and FJP scheduling algorithms are predictable. As a consequence, in considering such scheduling algorithms we need only consider the case where each job’s execution requirement is equal to the worst-case execution requirement of its generating task.

3.3 Dynamic Priority Scheduling

In DP scheduling, no restrictions are placed upon the manner in which priorities may be assigned to jobs; in particular, the priority of a job may be changed arbitrarily often between its release and completion times.

3.3.1 Partitioned Scheduling

Upon uniprocessor platforms, it has been shown [20,29] that the earliest deadline first scheduling algorithm (uniprocessor EDF) is an optimal algorithm for scheduling sporadic task systems upon uniprocessors in the sense that any feasible task system is also uniprocessor EDF-schedulable.

As a consequence of this optimality of uniprocessor EDF, observe that any task system that is schedulable by some partitioned DP scheduling algorithm can also be scheduled by partitioning the tasks among the processors in exactly the same manner, and subsequently scheduling each partition by EDF. Since EDF is a FJP scheduling algorithm, it follows that *DP scheduling is no more powerful than FJP scheduling with respect to the partitioned scheduling of sporadic task systems*. FJP partitioned scheduling of sporadic task systems is discussed in detail in Section 3.4.1; the entire discussion there holds for DP scheduling as well.

3.3.2 Global Scheduling

3.3.2.1 Implicit-Deadline Systems

For global scheduling of implicit-deadline sporadic task systems, the following result is easily proved:

Test 1 *Any implicit-deadline sporadic task system τ satisfying*

$$u_{\text{sum}}(\tau) \leq m \quad \text{and} \quad u_{\text{max}}(\tau) \leq 1$$

is schedulable upon a platform comprised of m unit-capacity processors by a DP scheduling algorithm.

To see why this holds, observe that a “processor sharing” schedule in which each job of each task τ_i is assigned a fraction u_i of a processor between its release time and its deadline, would meet all deadlines—such a processor-sharing schedule may subsequently be converted into one in which each job executes on zero or one processor at each time instant by means of the technique of Coffman and Denning [18, Chapter 3].

Such an algorithm is however clearly inefficient, and unlikely to be implemented in practice. Instead, powerful and efficient algorithms based on the technique of *pfair scheduling* [11] may be used to perform optimal global scheduling of implicit-deadline sporadic task systems upon multiprocessor platforms—see Ref. 1 for a survey.

Notice that task systems τ with $u_{\text{sum}}(\tau) > m$ or $u_{\text{max}}(\tau) > 1$ cannot possibly be feasible upon a platform comprised of m unit-capacity processors; hence, Test 1 represents an exact test for global DP-schedulability.

3.3.2.2 Constrained- and Arbitrary-Deadline Systems

To our knowledge, not much is known regarding the global DP scheduling of sporadic task systems with deadline distinct from periods. It can be shown that Test 1 is trivially extended to such systems by replacing the utilization (u) parameters by corresponding generalized density (λ) parameters:

Test 2 *Any sporadic task system τ satisfying*

$$\lambda_{\text{sum}}(\tau) \leq m \quad \text{and} \quad \lambda_{\text{max}}(\tau) \leq 1$$

is schedulable upon a platform comprised of m unit-capacity processors by a DP scheduling algorithm.

Unlike Test 1 above, this test is sufficient but not necessary for DP-schedulability.

3.4 Fixed Job-Priority Scheduling

3.4.1 Partitioned Scheduling

As stated in Section 3.3.1 above, the additional power of dynamic priority (DP) assignment provides no benefits with respect to partitioned scheduling; consequently, all the results in this section are valid for DP partitioned scheduling as well.

3.4.1.1 Implicit-Deadline Systems

Liu and Layland [29] proved that a periodic task system τ with implicit deadlines is schedulable on a single processor by EDF if and only if $u_{\text{sum}}(\tau) \leq 1$. They also showed that EDF scheduling is optimal for scheduling such task systems on a single processor in the sense that a task system is feasible if and only if it is schedulable by EDF. The proofs extend directly to the case of sporadic task systems with implicit deadlines.

Lopez et al. [34,35] applied this single-processor utilization bound test to partitioned multiprocessor scheduling with a first-fit-decreasing-utilization (FFDU) bin packing algorithm and obtained the following schedulability test. Although originally stated in terms of periodic tasks, the result generalizes trivially to the sporadic model.

Test 3 (FFDU EDF Utilization) *Any implicit-deadline sporadic task system τ is schedulable by some partitioned FJP algorithm (specifically, FFDU partitioning and EDF local scheduling) upon a platform of m unit-capacity processors, if*

$$u_{\text{sum}}(\tau) < \frac{m\beta_{\text{EDF}}(\tau) + 1}{\beta_{\text{EDF}}(\tau) + 1}$$

where $\beta_{\text{EDF}}(\tau) = \lfloor 1/u_{\text{max}}(\tau) \rfloor$.

If $u_{\text{max}}(\tau) = 1$ then $\beta_{\text{EDF}}(\tau) = 1$, and the above condition reduces to the following simpler test:

Test 4 (Simple FFDU EDF Utilization) *Any implicit-deadline sporadic task system τ is schedulable by some partitioned FJP algorithm (specifically, FFDU partitioning and EDF local scheduling) upon a platform of m unit-capacity processors, if $u_{\text{sum}}(\tau) < (m + 1)/2$.*

3.4.1.2 Constrained- and Arbitrary-Deadline Systems

Baruah and Fisher [8] proposed several different algorithms for partitioning constrained- and arbitrary-deadline sporadic task systems among the processors of a multiprocessor platform. In first-fit decreasing density (FFDD) partitioning, the tasks are considered in nonincreasing order of their generalized density parameters; when considering a task, it is assigned to any processor upon which the total density assigned thus far does not exceed the computing capacity of the processor. For such a partitioning scheme, they proved the following:

Test 5 (FFDU EDF) *Any arbitrary-deadline sporadic task system τ is schedulable by a partitioned FJP algorithm (specifically, FFDD partitioning and EDF local scheduling) upon a platform comprised of m unit-capacity processors, provided*

$$\lambda_{\text{sum}}(\tau) \leq \begin{cases} m - (m - 1)\lambda_{\text{max}}(\tau) & \text{if } \lambda_{\text{max}}(\tau) \leq \frac{1}{2} \\ \frac{m}{2} + \lambda_{\text{max}}(\tau) & \text{if } \lambda_{\text{max}}(\tau) \geq \frac{1}{2} \end{cases} \quad (3.3)$$

Baruah and Fisher [8] also proposed a somewhat more sophisticated partitioning scheme that runs in time polynomial (quadratic) in the number of tasks in the system. This scheme—first-fit-increasing-deadline (FFID)—considers tasks in increasing order of their relative deadline parameters. In considering a task, it is assigned to any processor upon which it would meet all deadlines using EDF as the local scheduling algorithm where the test for local EDF-schedulability is done by computing an approximation

to the DBFs of the tasks. This local EDF-schedulability test is sufficient rather than exact; with this test for local EDF-schedulability, they derived the following test:

Test 6 (FFID EDF) *Any arbitrary-deadline sporadic task system τ is schedulable by a partitioned FJP algorithm (specifically, FFID partitioning and EDF local scheduling) upon a platform comprised of m unit-capacity processors, provided*

$$m \geq \frac{2\text{load}(\tau) - \delta_{\max}(\tau)}{1 - \delta_{\max}(\tau)} + \frac{u_{\text{sum}}(\tau) - u_{\max}(\tau)}{1 - u_{\max}(\tau)}$$

For constrained-deadline systems, the test becomes somewhat simpler [9]:

Test 7 (FFID EDF; Constrained deadlines) *Any constrained-deadline sporadic task system τ is schedulable by a partitioned FJP algorithm (specifically, FFID partitioning and EDF local scheduling) upon a platform comprised of m unit-capacity processors, if*

$$\text{load}(\tau) \leq \frac{1}{2}(m - (m - 1)\delta_{\max}(\tau))$$

3.4.2 Global Scheduling

3.4.2.1 Implicit-Deadline Systems

Dhall and Liu [21] showed that the uniprocessor EDF utilization bound test does not extend directly to global multiprocessor scheduling. In fact, they showed that there are implicit-deadline sporadic task systems τ with $u_{\text{sum}}(\tau) = 1 + \epsilon$ for arbitrarily small positive ϵ , such that τ is not schedulable on m processors, for any value of m . The key to this observation is that m tasks with infinitesimal utilization but short deadlines will have higher priority than a task with a longer deadline and utilization near 1. For several years, this “Dhall phenomenon” was interpreted as proof that global EDF scheduling is unsuitable for hard-real-time scheduling on multiprocessors.

Notice, however, that the phenomenon demonstrated by Dhall and Liu depends on there being at least one task with very high utilization. This observation was exploited in Refs. 24 and 44 to obtain the following sufficient condition for global EDF-schedulability of implicit-deadline sporadic task systems:

Test 8 (EDF Utilization) *Any implicit-deadline sporadic task system τ satisfying*

$$u_{\text{sum}}(\tau) \leq m - (m - 1)u_{\max}(\tau) \tag{3.4}$$

is schedulable upon a platform comprised of m unit-capacity processors by the global EDF scheduling algorithm.

(Though written for periodic task systems, the proofs of the above test extend trivially to sporadic task systems with implicit deadlines.)

To handle systems with large u_{\max} , Srinivasan and Baruah [44] also proposed a hybrid scheduling algorithm called EDF-US[ζ], for any positive $\zeta \leq 1$. EDF-US[ζ] gives top priority to jobs of tasks with utilizations above some threshold ζ and schedules jobs of the remaining tasks according to their deadlines. In Ref. 44 it is proven that EDF-US[$m/(2m - 1)$] successfully schedules any implicit-deadline sporadic task system τ with total utilization $u_{\text{sum}}(\tau) \leq m^2/(2m - 1)$, regardless of the value of $u_{\max}(\tau)$. Baker [4] showed that this result implies that any sporadic task system τ with implicit deadlines can be scheduled by EDF-US[1/2] on m processors if $u_{\text{sum}}(\tau) \leq (m + 1)/2$.

Goossens et al. [24] suggested another approach for dealing with cases where $u_{\max}(\tau)$ is large. The algorithm they call EDF(k) assigns top priority to jobs of the $k - 1$ tasks that have the highest utilizations,

and assigns priorities according to deadline to jobs generated by all the other tasks. They showed that an implicit-deadline sporadic task system τ is schedulable on m unit-capacity processors by EDF(k) if

$$m \geq (k - 1) + \left\lceil \frac{u_{\text{sum}}(\tau) - u_k(\tau)}{1 - u_k(\tau)} \right\rceil$$

where $u_k(\tau)$ denotes the utilization of the k th task when the tasks are ordered by nondecreasing utilization.

Goossens et al. [24] also proposed an algorithm called PRI-D, which computes the minimum value m such that there exists a value $k < n$ that satisfies condition (3.4.2.1) above, and then schedules the task system on m processors using EDF(k). Baker [4] suggested a variant on this idea, called EDF(k_{\min}) here, in which the value m is fixed, k is chosen to be the minimum value that satisfies this condition, and the task system is then scheduled by EDF(k). EDF(k_{\min}) can schedule any sporadic task system τ with implicit deadlines on m processors if $u_{\text{sum}}(\tau) \leq (m + 1)/2$. This is the same utilization bound as EDF-US[1/2], but the domain of task systems that are schedulable by EDF(k_{\min}) is a superset of those schedulable by EDF-US[1/2].

Test 9 (EDF Hybrid Utilization) *Any implicit-deadline sporadic task system τ satisfying*

$$u_{\text{sum}}(\tau) \leq (m + 1)/2$$

is schedulable upon a platform comprised of m unit-capacity processors by the global scheduling algorithms EDF-US[1/2] or EDF(k_{\min}).

Observe that this is the same bound as in partitioned FJP scheduling (Test 4).

The worst-case utilization bound of $(m + 1)/2$ achieved by EDF-US[1/2] and EDF(k_{\min}) is optimal. Andersson et al. [2] showed that the utilization guarantee for EDF or any other multiprocessor scheduling algorithm that assigns a fixed priority to each job—whether partitioned or global—cannot be higher than $(m + 1)/2$ on an m -processor platform. (This result was stated for periodic task systems with implicit deadlines, but the proof applies also to the sporadic case.) Consider a set of $m + 1$ identical sporadic tasks with $p_i = d_i = 2x$ and $e_i = x + 1$, and consider the case when all the tasks release jobs simultaneously. It is clear that such a job set cannot be scheduled successfully on m processors. By making x sufficiently large the total utilization can be arbitrarily close to $(m + 1)/2$.

3.4.2.2 Constrained- and Arbitrary-Deadline Systems

Bertogna et al. [13] showed that the results presented above for implicit-deadline systems hold for constrained-deadline systems as well, when the utilization parameters in the tests ($u_{\text{sum}}(\tau)$, $u_{\text{max}}(\tau)$, and $u_k(\tau)$) are replaced by corresponding density parameters. Further examination of the proof reveals that it also generalizes to arbitrary deadlines. Hence,

Test 10 *Any sporadic task system τ satisfying*

$$\lambda_{\text{sum}}(\tau) \leq m - (m - 1)\lambda_{\text{max}}(\tau)$$

is schedulable upon a platform comprised of m unit-capacity processors by a fixed-priority global scheduling algorithm.

Once again, this test is sufficient but not necessary for global FJP-schedulability. Baker [3,4] determined the schedulability of some task systems more precisely by finding a separate density-based upper bound $\beta_k^\lambda(i)$ on the contribution of each task τ_i to the load of an interval leading up to a missed deadline of a task τ_k , and testing the schedulability of each task individually.

Test 11 (Baker EDF) *An arbitrary-deadline sporadic task system τ is schedulable on m processors using global EDF scheduling if, for every task τ_k , there exists $\lambda \in \{\lambda_k\} \cup \{u_i \mid u_i \geq \lambda_k, \tau_i \in \tau\}$, such that*

$$\sum_{\tau_i \in \tau} \min(\beta_k^\lambda(i), 1) \leq m(1 - \lambda) + \lambda$$

where

$$\beta_k^\lambda(i) \stackrel{\text{def}}{=} \begin{cases} u_i(1 + \max(0, \frac{T_i - d_i}{d_k})) & \text{if } u_i \leq \lambda \\ u_i(1 + \frac{p_i}{d_k}) - \lambda \frac{d_i}{d_k} & \text{if } u_i > \lambda \text{ and } d_i \leq p_i \\ u_i(1 + \frac{p_i}{d_k}) & \text{if } u_i > \lambda \text{ and } d_i > p_i \end{cases}$$

This test is able to verify some schedulable task systems that cannot be verified by the generalized EDF utilization test (Test 10), and the generalized EDF density bound test is able to verify some task systems that cannot be verified using the Baker test. The computational complexity of the Baker test is $\Theta(n^3)$ (where n is the number of sporadic tasks in the system) while Test 10 has an $\mathcal{O}(n^2)$ computational complexity.

A direct application of the DBF to global scheduling has not yet been obtained. However, Bertogna et al. [13] have taken a step in that direction by devising a schedulability test that uses DBF for the interval of length p_i preceding a missed deadline of some job of a task τ_k , based on exact computation of the demand of a competing task over that interval.

Test 12 (BCL EDF) *Any constrained-deadline sporadic task system is schedulable on m processors using preemptive EDF scheduling if for each task τ_k one of the following is true:*

$$\sum_{i \neq k} \min(\beta_k(i), 1 - \lambda_k) < m(1 - \lambda_k) \quad (3.5)$$

$$\sum_{i \neq k} \min(\beta_k(i), 1 - \lambda_k) = m(1 - \lambda_k) \quad \text{and} \quad \exists i \neq k : 0 < \beta_k(i) \leq 1 - \lambda_k \quad (3.6)$$

where

$$\beta_k(i) \stackrel{\text{def}}{=} \frac{\text{DBF}(\tau_i, e_k) + \min(e_i, \max(0, d_k - \text{DBF}(\tau_i, e_k)p_i/e_i))}{d_k}$$

This test is incomparable with both Tests 10 and 11 above in the sense that there are task systems determined to be schedulable by each test for which the other two tests fail to determine schedulability.

An alternative approach to global multiprocessor EDF schedulability analysis of sporadic task systems is based on the following result, paraphrased from Ref. 37:

Theorem 3.1 (from Ref. 37)

Any independent collection of jobs that is feasible upon a multiprocessor platform comprised of m processors each of computing capacity $m/(2m - 1)$ is global EDF-schedulable upon a multiprocessor platform comprised of m unit-capacity processors.

Theorem 3.1 above may be used to transform any test demonstrating feasibility of a sporadic task system into a global FJP schedulability test. For instance, recall that Test 7 deemed any constrained-deadline sporadic task system satisfying the following condition to be schedulable by a partitioned FJP algorithm upon m unit-capacity processors:

$$\text{load}(\tau) \leq \frac{m - (m - 1)\delta_{\max}(\tau)}{2}$$

Any τ satisfying the above condition is clearly feasible on m unit-capacity processors as well. This fact, in conjunction with Theorem 3.1 above, yields the following sufficient condition for multiprocessor global EDF schedulability:

Test 13 *Any constrained-deadline sporadic task system τ is global EDF-schedulable upon a platform comprised of m unit-capacity processors, provided*

$$\text{load}(\tau) \leq \frac{\frac{m^2}{2m-1} - (m-1)\delta_{\max}(\tau)}{2}$$

3.5 Fixed Task-Priority Scheduling

Partisans of EDF scheduling [16] have argued that FTP scheduling should be forgotten, since EDF scheduling has clear advantages. On single processors, the optimality of EDF ([20,29]) implies that it successfully schedules all task systems that are successfully scheduled by any FTP algorithm. Even on multiprocessors it is trivially true that FJP scheduling performs no worse than FTP scheduling since all FTP algorithms are also FJP algorithms by definition, while the converse is not true. EDF also allows each application thread to express its own deadline requirement directly, rather than indirectly through priority as in FTP-scheduled systems. In contrast, making the right priority assignments in an FTP system can be very difficult to do, and the priority assignment is fragile since a change to any thread may require adjusting the priorities of all other threads. However, there are still many loyal users of FTP scheduling, who cite its own advantages. FTP is arguably simpler than EDF to implement. It is supported by nearly every operating system, including those that support the POSIX real-time application program interface [27]. With FTP it may be simpler to design a system for overload tolerance, since the lowest-priority tasks will be impacted first. Therefore, FTP scheduling is still important and is of more immediate importance to some users than EDF.

3.5.1 Partitioned Scheduling

3.5.1.1 Implicit-Deadline Systems

Liu and Layland [29] analyzed FTP as well as EDF scheduling upon preemptive uniprocessors. They showed that ordering task priorities according to period parameters is optimal for implicit-deadline periodic task systems in the sense that if a task system is schedulable under any FTP scheme, then the task system is schedulable if tasks are ordered according to period parameters, with smaller-period tasks being assigned greater priority. This priority order is called rate monotonic (RM). The proofs extend directly to the case of sporadic tasks.

Test 14 (Uniprocessor RM Utilization) *Any implicit-deadline sporadic task system τ comprised of n tasks is schedulable upon a unit-capacity processor by RM, if $u_{\text{sum}}(\tau) \leq n(2^{1/n} - 1)$.*

Unlike the uniprocessor EDF schedulability test, this uniprocessor RM utilization test is sufficient but not necessary.

Oh and Baker [36] applied this uniprocessor utilization bound test to partitioned multiprocessor FTP scheduling using (FFDU) assignment of tasks to processors and RM local scheduling on each processor. They proved that any system of implicit-deadline sporadic tasks with total utilization $U < m(2^{1/2} - 1)$ is schedulable by this method on m processors. They also showed that for any $m \geq 2$ there is a task system with $U = (m+1)/(1+2^{1/(m+1)})$ that cannot be scheduled upon m processors using partitioned FTP

scheduling. Lopez et al. [31–33] refined and generalized this result, and along with other more complex schedulability tests, showed the following:

Test 15 (FFDU RM Utilization) *Any implicit-deadline sporadic task system τ comprised of n tasks is schedulable upon m unit-capacity processors by an FTP partitioning algorithm (specifically, FFDU partitioning and RM local scheduling), if*

$$u_{\text{sum}}(\tau) \leq (n-1)(2^{1/2} - 1) + (m-n+1)(2^{1/(m-n+1)} - 1)$$

3.5.1.2 Constrained- and Arbitrary-Deadline Systems

Leung [28] showed that ordering task priorities according to the relative deadline parameters (this priority assignment is called deadline monotonic [DM]) is an optimal priority assignment for the uniprocessor scheduling of constrained-deadline sporadic task systems. Hence, DM is the logical choice for use as the local scheduling algorithm in partitioned FTP schemes for the multiprocessor scheduling of constrained-deadline sporadic task systems; simulations [6] indicate that DM performs well on arbitrary-deadline sporadic task systems as well.

Fisher et al. [22] studied the partitioned FTP scheduling of sporadic constrained- and arbitrary-deadline task systems when DM scheduling is used as the local scheduler on each processor. They defined a variation on the FFID partitioning scheme described in Section 3.4.1.2, called FBB-FFD, and obtained the following schedulability tests:

Test 16 *Any arbitrary sporadic task system τ is partitioned FTP-schedulable (specifically, by FBB-FFD partitioning and DM local scheduling) upon a platform comprised of m unit-capacity processors, if*

$$m \geq \frac{\text{load}(\tau) + u_{\text{sum}}(\tau) - \delta_{\text{max}}(\tau)}{1 - \delta_{\text{max}}(\tau)} + \frac{u_{\text{sum}}(\tau) - u_{\text{max}}(\tau)}{1 - u_{\text{max}}(\tau)}$$

For constrained-deadline systems, the test becomes somewhat simpler:

Test 17 *Any constrained-deadline sporadic task system τ is partitioned FTP-schedulable (once again, by FBB-FFD partitioning and DM local scheduling) upon a platform comprised of m unit-capacity processors, if*

$$\text{load}(\tau) + u_{\text{sum}}(\tau) \leq m - (m-1)\delta_{\text{max}}(\tau)$$

3.5.2 Global Scheduling

3.5.2.1 Implicit-Deadline Systems

Andersson et al. [2] showed that any implicit-deadline periodic task system τ satisfying $u_{\text{max}}(\tau) \leq m/(3m-2)$ can be scheduled successfully on m processors using RM scheduling, provided $u_{\text{sum}}(\tau) \leq m^2/(3m-1)$.

Bertogna et al. [14] proved the following tighter result:

Test 18 (RM Utilization) *Any implicit-deadline sporadic task system τ is global FTP-schedulable (by global RM scheduling) upon a platform comprised of m unit-capacity processors, if*

$$u_{\text{sum}}(\tau) \leq \frac{m}{2}(1 - u_{\text{max}}(\tau)) + u_{\text{max}}(\tau)$$

Andersson et al. [2] proposed a hybrid scheduling algorithm called RM-US[$m/(3m-2)$], similar to EDF-US [44], that gives higher priority to tasks with utilizations above $m/(3m-2)$, and showed it is able to successfully schedule any set of independent periodic tasks with total utilization up to $m^2/(3m-1)$.

Baker [5] considered RM-US[ζ] for arbitrary values of ζ , showing that the optimum value of ζ is $1/3$ and RM-US[$1/3$] is always successful if $u_{\text{sum}} \leq m/3$.

Bertogna et al. [14] tightened the above result and proved the following general utilization bound:

Test 19 (RM Utilization) *Any implicit-deadline sporadic task system τ is global FTP-schedulable (by global RM[$1/3$] scheduling) upon a platform comprised of m unit-capacity processors, if*

$$u_{\text{sum}}(\tau) \leq \frac{m+1}{3}$$

3.5.2.2 Constrained- and Arbitrary-Deadline Systems

Baker [3] derived an FTP schedulability test based on generalized density for sporadic task systems with constrained deadlines. The analysis was refined in Ref. 5 to allow arbitrary deadlines. Baker and Cirinei [7] combined an insight from Refs. 13 and 14 with the analysis of Ref. 5, and unified the treatment of FTP and FJP scheduling. If one considers only the FTP case, the Baker and Cirinei test reduces to the following:

Test 20 (BC FTP) *Any arbitrary-deadline sporadic task system τ is global FTP-schedulable upon a platform comprised of m unit-capacity processors, if for every task τ_k , there exists $\lambda \in \{\lambda_k\} \cup \{u_i \mid u_i \geq \lambda_k, \tau_i \in \tau\}$, such that one of the following is true:*

$$\sum_{i < k} \min(\beta_k^\lambda(i), 1 - \lambda) < m(1 - \lambda)$$

$$\sum_{i < k} \min(\beta_k^\lambda(i), 1 - \lambda) = m(1 - \lambda) \quad \text{and} \quad \exists i \neq k : 0 < \beta_k^\lambda(i) \leq 1 - \lambda_k$$

where

$$\beta_k^\lambda(i) \stackrel{\text{def}}{=} \begin{cases} u_i(1 + \max(0, \frac{p_i - e_i}{d_k})) & \text{if } u_i \leq \lambda \\ u_i(1 + \max(0, \frac{d_i + p_i - e_i - \lambda d_i / u_i}{d_k})) & \text{if } u_i > \lambda \end{cases}$$

The RM-US[ζ] idea can be applied to arbitrary fixed task-priority schemes and applied to task systems with arbitrary deadlines as well as task systems with implicit deadlines. With FTP-US[ζ] one gives top priority to the k tasks with utilization above ζ , for $k < m$, and schedules the remaining tasks according to some other fixed task-priority scheme. Clearly, the top priority tasks will be schedulable, since $k < m$. Any basic FTP-schedulability test can then be applied to the $n - k$ remaining tasks, using $m - k$ processors. The test may be overly conservative, but it is safe.

The idea of EDF(k) and EDF(k_{\min}) can also be applied with FTP. That is, for FTP(k) assign top priority to jobs of the $k - 1$ tasks in τ that have highest utilizations, and assign lower priorities according to the given FTP scheme to jobs generated by all the other tasks in τ . FTP(k_{\min}) uses the least k for which FTP(k) can be verified as schedulable. The same strategy for testing schedulability using a basic FTP-schedulability test on $m - k$ processors applies.

Bertogna et al. [14] have applied to FTP scheduling the same technique they applied to FJP scheduling in Ref. 13, and obtained the following FTP analog of Test 12.

Test 21 (BCL FTP) Any constrained-deadline sporadic task system τ is global FTP-schedulable upon a platform comprised of m unit-capacity processors, if for each task τ_k one of the following is true:

$$\sum_{i < k} \min(\beta_k(i), 1 - \lambda_k) < m(1 - \lambda_k)$$

$$\sum_{i < k} \min(\beta_k(i), 1 - \lambda_k) = m(1 - \lambda_k) \quad \text{and} \quad \exists i \neq k : 0 < \beta_k(i) \leq 1 - \lambda_k$$

where

$$\beta_k(i) \stackrel{\text{def}}{=} \frac{N_{i,k}e_i + \min(e_i, \max(0, d_k - N_{i,k}p_i + d_i - e_i))}{d_k}$$

and

$$N_{i,k} = \left(\left\lfloor \frac{d_k - e_i}{p_i} \right\rfloor + 1 \right)$$

3.6 Relaxations of the Sporadic Model

The sporadic task model may seem very restrictive from the point of view of a programmer. The restrictions are imposed to simplify analysis, and cannot be removed entirely without making it impractical or impossible to determine whether a task system is schedulable. However, some of them can be relaxed using well-known tricks, if one is prepared to accept a degree of conservative overestimation in the analysis. The following are illustrative examples of how one can accommodate some features that at first seem to violate the sporadic model.

3.6.1 Sleeping within a Job

Within a single job, a task cannot execute for a while, put itself to sleep to wait for completion of an input or output operation, and then execute some more. That means there can be no deadlines that span more than one job.

It is possible to work around this restriction by modeling tasks with multistep jobs as if they were multiple independent jobs with appropriately related deadlines. For example, suppose a task repeats the following every 10 ms with relative deadline of 5 ms:

1. Execute computation A , taking up to 1 ms, then read some data from an input device.
2. Initiate an input operation, then wait for the input device to respond, where the worst-case response time of the device is 1 ms.
3. Execute computation B , taking up to 1 ms.
4. Initiate an output operation, and do not wait for the operation to complete.

This could be modeled by two independent tasks, each with period 10 ms. The first task would execute A and the initiation of the input operation, with relative deadline of 1.5 ms. The second would be released in response to the completion of the input operation (by an interrupt generated by the device) and execute B with deadline of 1.5 ms. If the first task can be completed within 1.5 ms, the I/O operation can be completed within 1 ms, and the second task can be completed within 1.5 ms, then the work of the original task can be completed within $1.5 + 1 + 1.5 = 4$ ms, satisfying the original relative deadline of 5 ms.

3.6.2 Task Interdependencies

The only precedence constraints in the sporadic task model are between jobs of the same task. There are no precedence constraints between jobs of different tasks, and a task cannot be blocked by a resource, such as a lock held by another task.

This restriction can be relaxed slightly when one is testing schedulability of a given task, by padding the worst-case execution time of the given task with the worst-case execution times of the longest possible combination of critical sections that can block it. However, care must be taken in a multiprocessor environment, since the worst-case impact of such resource blocking is greater than that in a single-processor environment. That is, on a single processor if a job is blocked the processor will at least be doing useful work executing another thread. On a multiprocessor, it is possible that $m - 1$ processors are idle because their tasks are blocked by a critical section of a job running on another processor. This problem may be reduced with partitioned scheduling, if tasks that contend for access to the same data are assigned to the same processor. However, in a global scheduling environment or where performance concerns dictate accessing shared data in parallel from multiple processors, contention for data access becomes unavoidable. It then becomes important to bound blocking time, by using a priority-aware locking protocol. Rajkumar et al. [38,39] have shown how to apply the priority ceiling protocol to multiprocessor systems with partitioned rate monotone (RM) scheduling, and Chen and Tripathi [17] have shown how to extend the technique to multiprocessor systems with partitioned EDF scheduling. Further study of this subject in the context of global scheduling is needed.

3.6.3 Nonpreemptability

The model assumes that a scheduler may preempt a task at any time, to give the processor to another task.

Short nonpreemptable sections may be accommodated by treating them as critical sections that can block every other task, and adding an appropriate allowance to the execution time of the task that is attempting to preempt. The problem of multiway blocking described above cannot occur. Even with global scheduling, the $m - 1$ jobs with highest priority will not be affected at all.

3.6.4 Scheduling and Task-Switching Overhead

The model does not take into account the execution time overheads involved in task scheduling, preemption, and migration of tasks between processors.

Overhead that is only incurred when a job starts or completes execution can be modeled by adding it to the execution time of the corresponding job, and overhead that is associated with timer interrupt service can be modeled by a separate task.

3.6.5 Changes in Task System

The model assumes that τ is fixed, including the number n of tasks and the values e_i , p_i , and d_i for each task.

This restriction can be relaxed by modeling the system as switching between a finite set of modes, with a different task system for each mode. If the task system for each mode is verified as schedulable, the only risk of missed deadlines is during switches between modes. Through careful control of the release times of tasks it is also possible to avoid missed deadlines during mode changes. At the extreme, one can delete the tasks of the old mode as their pending jobs complete, and wait to start up the tasks of the new mode until all of the tasks of the old mode have been deleted. As less extreme set of rules to ensure safe mode, changes in a single-processor FTP system was proposed in 1989 by Sha et al. [41], and corrected in 1992 by Burns et al. [15]. We do not know of any similar work for EDF scheduling or for FTP scheduling on multiprocessors.

3.6.6 Aperiodic Tasks

The model assumes that there is a nontrivial minimum interrelease time and that the execution time of each job of a task is consistent enough that it makes sense to schedule for the worst-case execution time.

This restriction can be relaxed by decoupling logical jobs from the jobs seen by the scheduler, and viewing the task as a server thread. That is, the thread of a sporadic server has a control structure like Figure 3.1. The server thread is scheduled according to one of the many budget-based preemptive bounded-allocation scheduling algorithms that are compatible with the underlying scheduling scheme.

```

SPORADIC_SERVER_TASK

1  for ever do
2      Sleep until queue is non-empty
3      while queue is not empty do
4          Remove next event from queue
5          Compute

```

FIGURE 3.1 Pseudocode for a sporadic server.

These scheduling schemes have in common that the scheduler maintains an execution time budget for the server thread, and it executes until either its queue is empty or the server's budget is exhausted. The server budget is replenished according to the particular scheduling algorithm. If the server is suspended for using up its budget, then, when the budget is replenished, the server resumes execution at the point it was suspended. If the replenishment and priority policies of the server limit the computational load to a level that can be modeled by a sporadic task, an aperiodic server may be treated as a sporadic task in the schedulability analysis. The sporadic server [42]*, deadline sporadic server [23], constant bandwidth server [19], and total bandwidth server [43] are examples of such scheduling policies. For an overview of simple aperiodic server scheduling algorithms for the FTP and EDF environments, see Ref. 40, and for more detail see Ref. 30. For specifics on how this approach can be applied to a constant bandwidth server (CBS) on a multiprocessor with EDF scheduling, see the work of Baruah et al. [10].

It appears that multiprocessor applications of aperiodic server scheduling, combined with bandwidth recovery techniques to compensate for overallocation of processing resources to guarantee hard deadline tasks, have a strong potential to exploit the advantages of global scheduling on multiprocessors.

3.7 Conclusion

We have shown how demand analysis techniques developed originally for the analysis of schedulability with priority-based preemptive scheduling on single-processor systems have been extended to apply to multiprocessor systems. There are several practical algorithmic tests for verifying the schedulability of such systems under both EDF and fixed task-priority scheduling in both partitioned and global modes. While none of the multiprocessor scheduling algorithms we have considered here is optimal, and none of the corresponding schedulability tests is tight, experiments with pseudorandomly generated task sets have shown that some of them are successful on most task sets, up to nearly full system utilization.

It is likely that further improvements can still be made in the accuracy of schedulability tests for global scheduling on multiprocessors, with fixed job-priority schemes like EDF and FTP, and that more dramatic improvements may be made as research on dynamic job-priority schemes progresses. However, the present state of knowledge is already sufficient to make effective use of multiple processors for systems with hard deadlines.

References

1. J. Anderson, P. Holman, and A. Srinivasan. Fair multiprocessor scheduling. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, Boca Raton, FL, 2003.

*The original description of the sporadic server in Ref. 42 contained a defect, that led to several variations. See Ref. 30 for details.

2. B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 193–202, London, December 2001.
3. T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proc. 24th IEEE Real-Time Systems Symposium*, pages 120–129, 2003.
4. T. P. Baker. An analysis of EDF scheduling on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):760–768, 2005.
5. T. P. Baker. An analysis of fixed-priority scheduling on a multiprocessor. *Real-Time Systems*, 32(1–2):49–71, 2005.
6. T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time. Technical Report TR-050601, Department of Computer Science, Florida State University, Tallahassee, FL, July 2005.
7. T. P. Baker and M. Cirinei. A unified analysis of global EDF and fixed-task-priority schedulability of sporadic task systems on multiprocessors. Technical Report TR-060501, Department of Computer Science, Florida State University, Tallahassee, FL, May 2006.
8. S. Baruah and N. Fisher. Partitioned multiprocessor scheduling of sporadic task systems. In *Proc. 26th IEEE Real-Time Systems Symposium*, Miami, FL, December 2005. IEEE Computer Society Press.
9. S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, 2006.
10. S. Baruah, J. Goossens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 154–163, San Jose, CA, September 2002.
11. S. K. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.
12. S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. 11th IEEE Real-Time Systems Symposium*, pages 182–190, 1990.
13. M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proc. 17th Euromicro Conference on Real-Time Systems*, pages 209–218, Palma de Mallorca, Spain, July 2005.
14. M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proc. 9th International Conference on Principles of Distributed Systems*, Pisa, Italy, December 2005.
15. A. Burns, K. Tindell, and A. J. Wellings. Mode changes in priority preemptive scheduled systems. In *Proc. 13th IEEE Real-Time Systems Symposium*, pages 100–109, Phoenix, AZ, 1992.
16. G. Buttazzo. Rate-monotonic vs. EDF: Judgement day. *Real-Time Systems*, 29(1):5–26, 2005.
17. C. M. Chen and S. K. Tripathi. Multiprocessor priority ceiling protocols. Technical Report CS-TR-3252, Department of Computer Science, University of Maryland, April 1994.
18. J. E. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1973.
19. Z. J. Deng, W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proc. 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, June 1997.
20. M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proc. IFIP Congress*, pages 807–813, 1974.
21. S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.
22. N. Fisher, S. Baruah, and T. Baker. The partitioned scheduling of sporadic tasks according to static priorities. In *Proc. EuroMicro Conference on Real-Time Systems*, pages 118–127, Dresden, Germany, July 2006. IEEE Computer Society Press.
23. T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9, pages 31–67, 1995.
24. J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2–3):187–205, 2003.

25. R. Ha. *Validating timing constraints in multiprocessor and distributed systems*. PhD thesis, University of Illinois, Department of Computer Science, Urbana-Champaign, IL, 1995.
26. R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. 14th IEEE International Conference on Distributed Computing Systems*, pages 162–171, Poznan, Poland, June 1994. IEEE Computer Society.
27. IEEE Portable Applications Standards Committee. *ISO/IEC 9945-2:2003(E), Information Technology—Portable Operating System Interface (POSIX) – Part 2: System Interfaces*. IEEE Standards Association, 2003.
28. J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, 1982.
29. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
30. J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Upper Saddle River, NJ, 2000.
31. J. M. Lopez, J. L. Diaz, and D. F. Garcia. Minimum and maximum utilization bounds for multiprocessor RM scheduling. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 67–75, Delft, Netherlands, June 2001.
32. J. M. Lopez, J. L. Diaz, and D. F. Garcia. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):642–653, 2004.
33. J. M. Lopez, J. L. Diaz, and D. F. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems: The International Journal of Time-Critical Computing*, 28(1):39–68, 2004.
34. J. M. Lopez, J. L. Diaz, M. Garcia, and D. F. Garcia. Utilization bounds for multiprocessor RM scheduling. *Real-Time Systems*, 24(1):5–28, 2003.
35. J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proc. 12th Euromicro Conference on Real-Time Systems*, pages 25–33, 2000.
36. D. I. Oh and T. P. Baker. Utilization bounds for N -processor rate monotone scheduling with stable processor assignment. *Real-Time Systems*, 15(2):183–193, 1998.
37. C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proc. 29th Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, TX, 1997. ACM.
38. R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.
39. R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. 10th International Conference on Distributed Computing*, pages 116–125, 1990.
40. L. Sha, T. Abdelzaher, K. E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, 2004.
41. L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):244–264, 1989.
42. B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.
43. M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.
44. A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84:93–98, 2002.

Rate-Based Resource Allocation Methods

4.1	Introduction	4-1
4.2	Traditional Static Priority Scheduling	4-3
	Mapping Performance Requirements to Priorities • Managing “Misbehaved” Tasks • Providing Graceful/ Uniform Degradation • Achieving Full Resource Utilization	
4.3	A Taxonomy of Rate-Based Allocation Models	4-4
	Fluid-Flow Resource Allocation: Proportional Share Scheduling • Liu and Layland Extensions: Rate-Based Execution • Server-Based Allocation: The Constant Bandwidth Server	
4.4	Using Rate-Based Scheduling	4-9
	A Sample Real-Time Workload • Rate-Based Scheduling of Operating System Layers • Workload Performance under Proportional Share Allocation • Workload Performance under Generalized Liu and Layland Allocation • Workload Performance under Server-Based Allocation	
4.5	Hybrid Rate-Based Scheduling	4-12
4.6	Summary and Conclusions	4-13

Kevin Jeffay
University of North Carolina

4.1 Introduction

Operating systems for real-time and embedded systems have long relied on static priority scheduling to ensure that timing and performance constraints are met. In static priority scheduling, tasks are assigned an integer priority value that remains fixed for the lifetime of the task. Whenever a task is made ready to run, the active task with the highest priority commences or resumes execution, preempting the currently executing task if need be. There is a rich literature that analyzes static priority scheduling and demonstrates how timing and synchronization constraints can be met using static priority scheduling (see Ref. 14 for a good summary discussion). For these and other reasons, virtually all commercial real-time operating systems, including VxWorks [27], VRTX [17], QNX [22], pSOSystem (pSOS) [21], and LynxOS [16], support static priority scheduling.

Static priority scheduling works well for systems where there exists a natural strict priority relationship between tasks. However, it can often be difficult to use static priority scheduling to satisfy the performance requirements of systems when the priority relationships between tasks are either not strict or not obvious. In general, static priority scheduling has a number of significant shortcomings including

- An inability to directly map timing or importance constraints into priority values.
- The problem of dealing with tasks whose execution time is either unknown or may vary over time.
- The problem of dealing with tasks whose execution time or execution rate deviates significantly at runtime from the behavior expected at design time.

- The problem of degrading system performance gracefully in times of overload.
- The problem of ensuring full utilization of the processor or other system resources in tightly resource constrained systems.

Rate-based resource allocation is a more flexible form of real-time resource allocation that can be used to design systems where performance requirements are better described in terms of required processing rates rather than priority relationships. In a rate-based system, a task is guaranteed to make progress according to a well-defined rate specification such as “process x samples per second,” or “process x messages per second, where each message consists of 3–5 consecutive network packets.”

This chapter summarizes the theory and practice of rate-based resource allocation and demonstrates how rate-based methods can provide a framework for the natural specification and realization of timing constraints in real-time and embedded systems. To structure the discussion, three dimensions of the basic resource allocation problem are considered:

The type of resource allocation problem. To fully explore the utility of rate-based resource allocation, three scheduling problems are considered: application-level scheduling (i.e., scheduling of user programs or tasks/threads), scheduling the execution of system calls made by applications (“top-half” operating system-level scheduling), and scheduling asynchronous events generated by devices (“bottom-half” operating system-level scheduling). This treatment is motivated by the logical structure of traditional, monolithic real-time (and general-purpose) operating systems with hardware-enforced protection boundaries. Moreover, this simple taxonomy of scheduling problems emphasizes that in real systems one must consider issues of intrakernel resource allocation and scheduling as well as application task scheduling.

The type or class of rate-based resource allocation method. We consider three broad classes of rate-based resource allocation paradigms: allocation based on a fluid-flow paradigm, allocation based on a polling or periodic server paradigm, and allocation based on a generalized Liu and Layland paradigm. For an instance of the fluid-flow paradigm, the proportional share (PS) scheduling algorithm *earliest eligible virtual deadline first* (EEVDF) [26] is considered. For an instance of the polling server paradigm, scheduling based on the *constant bandwidth server* (CBS) concept [1] is considered. Finally, for the generalized Liu and Layland paradigm a rate-based extension to the original Liu and Layland task model called *rate-based execution* (RBE) [8] is considered.

The characteristics of the workload generated. For each rate-based allocation scheme above, the likely expected real-time performance of an application is considered under a set of execution environments where the applications execute at various rates. Specifically, cases are considered wherein the applications execute at “well-behaved,” constant rates, at bursty rates, and at uncontrolled “misbehaved” rates. For well-behaved workloads, the three rate-based schemes considered (and indeed virtually any real-time scheduling scheme) can execute a given workload in real time. However, the rate-based schemes perform quite differently when applications need to be scheduled at bursty or uncontrolled rates.

The goal is to demonstrate how rate-based methods naturally solve a variety of resource allocation problems that traditional static priority scheduling methods are inherently poorly suited to. However, it will also be argued that “one size does not fit all.” One rate-based resource allocation scheme does not suffice for all scheduling problems that arise within the layers of a real system. While one can construct an execution environment wherein all of the rate-based schemes considered here perform well, for more realistic environments that are likely to be encountered in practice, the best results are likely to be achieved by employing different rate-based allocation schemes at different levels in the operating system.

The following section describes the shortcomings of static priority scheduling in more detail. Section 4.3 presents a taxonomy of rate-based resource allocation methods. Three classes of rate-based resource allocation are described and the strengths and weaknesses of each approach are discussed. The use of each class of rate-based method is demonstrated through a series of implementation experiments. This evaluation is presented in Section 4.4. Section 4.5 proposes and evaluates a hybrid scheme that combines different forms of rate-based resource allocation within different layers of the operating system and application. Section 4.6 summarizes the results and concludes with a discussion of directions for further study in rate-based resource allocation.

4.2 Traditional Static Priority Scheduling

Traditional models of real-time resource allocation are based on the concept of a discrete but recurring event, such as a periodic timer interrupt, that causes the release of a task. The task must be scheduled such that it completes execution before a well-defined deadline. For example, most real-time models of execution are based on the Liu and Layland periodic task model [15] or Mok's sporadic task model [18]. In these models, each execution of a task must complete before the next instance of the same task is released. The challenge is to assign priority values to tasks such that all releases of all tasks are guaranteed to complete execution before their respective deadlines when scheduled by a preemptive priority scheduler. Common priority assignment schemes include the *rate-monotonic* scheme [15], wherein tasks that recur at high rates have priority over tasks that recur at low rates (i.e., a task's priority is equal to its recurrence period), and the *deadline monotonic* scheme [13], wherein a task's priority is related to its response time requirement (it's deadline). In either case, static assignment of priority values to tasks leads to a number of problems.

4.2.1 Mapping Performance Requirements to Priorities

Simple timing constraints for tasks that are released in response to a single event, such as a response time requirement for a specific interrupt, can be easily specified in a static priority system. More complex constraints are frequently quite difficult to map into priority values. For example, consider a signal processing system operating on video frames that arrive over a LAN from a remote acquisition device. The performance requirement may be to process 30 frames/s, and hence it would be natural to model the processing as a periodic task and trivial to schedule using a static priority scheduler. However, it is easily the case that each video frame arrives at the processing node in a series of network packets that must be reassembled into a video frame, and it is not at all obvious how the network packet and protocol processing should be scheduled. That is, while there is a natural constraint for the application-level processing, there is no natural constraint (e.g., no unique response time requirement) for the processing of the individual events that will occur during the process of realizing the higher-level constraint. Nonetheless, in a static priority scheduler the processing of these events must be assigned a single priority value.

The problem here is that the system designer implicitly creates a response time requirement when assigning a priority value to the network processing. Since there is no natural unique value for this requirement, a conservative (e.g., short response time) value is typically chosen. This conservative assignment of priority has the effect of reserving more processing resources for the task than will actually ever be consumed and ultimately limits either the number or the complexity of tasks that can be executed on the processor.

4.2.2 Managing “Misbehaved” Tasks

To analyze any resource allocation problems, assumptions must be made about the environment in which the task executes. In particular, in virtually all real-time problems the amount of resources required for the execution of the task (e.g., the amount of processor time required to execute the task) is assumed to be known in advance. A second problem with static priority resource allocation occurs when assumptions such as these are violated and a task “misbehaves” by consuming more resources than expected. The problem is to ensure that a misbehaving task does not compromise the execution of other “well-behaved” tasks in the system.

An often-touted advantage of static priority systems is that if a task violates its execution assumptions, higher-priority tasks are not affected. While it is true that higher-priority tasks are not affected by misbehaving lower-priority tasks (unless higher-priority tasks share software resources with the lower-priority tasks), all other tasks have no protection from the misbehaving task. For example, a task that is released at a higher rate than expected can potentially block *all* lower-priority tasks indefinitely.

The issue is that static priority scheduling fundamentally provides no mechanism for isolating tasks from the ill effects of other tasks other than the same mechanism that is used to ensure response time properties.

Given that isolation concerns typically are driven by the relative importance of tasks (e.g., a noncritical task should never effect or interfere with the execution of a mission-critical task), and importance and response time are often independent concepts, attempting to manage both concerns with a single mechanism is inappropriate.

4.2.3 Providing Graceful/Uniform Degradation

Related to the task isolation problem is that of providing graceful performance degradation under transient (or persistent) overload conditions. The problem of graceful degradation can be considered a generalization of the isolation problem; a set of tasks (or the environment that generates work for the tasks) misbehaves and the processing requirements for the system as a whole increase to the point where tasks miss deadlines. In these overload situations, it is again useful to control which tasks' performance degrades and by how much.

Static priority scheduling again has only a single mechanism for managing importance and response time. If the mission-critical tasks also have the smallest response time requirements then they will have the highest priority and will continue to function. However, if this is not the case then there is no separate mechanism to control the execution of important tasks. Worse, even if the priority structure matches the importance structure, in overload conditions under static priority scheduling only one form of degraded performance is possible: high-priority tasks execute normally while the lowest-priority task executes at a diminished rate if at all. That is, since a task with priority p will *always* execute to completion before any pending task with priority less than p commences or resumes execution, it is impossible to control how tasks degrade their execution. (Note however, as frequently claimed by advocates of static priority scheduling, the performance of a system under overload conditions is predictable.)

4.2.4 Achieving Full Resource Utilization

The rate-monotonic priority assignment scheme is well known to be an optimal static priority assignment. (Optimal in the sense that if a static priority assignment exists which guarantees periodic tasks have a response time no greater than their period, then the rate-monotonic priority assignment will also provide the same guarantees.) One drawback to static priority scheduling, however, is that the achievable processor utilization is restricted. In their seminal paper, Liu and Layland showed that a set of n periodic tasks will be schedulable under a rate-monotonic priority assignment if the processor utilization of a task set does not exceed $n(2^{1/n} - 1)$ [15]. If the utilization of a task set exceeds this bound then the tasks may or may not be schedulable. (That is, this condition is a sufficient but not necessary condition for schedulability.) Moreover, Liu and Layland showed that in the limit the utilization bound approached $\ln 2$ or approximately 0.69. Thus, 69% is the least upper bound on the processor utilization that guarantees feasibility. A least upper bound here means that this is the minimum utilization of all task sets that fully utilize the processor. (A task set fully utilizes the processor if increasing the cost of any task in the set causes a task to miss a deadline.) If the utilization of the processor by a task set is less than or equal to 69% then the tasks are guaranteed to be schedulable.

Lehoczky et al. [12] formulated an exact test for schedulability under a rate-monotonic priority assignment and showed that ultimately, schedulability is not a function of processor utilization. However, nonetheless, in practice the utilization test remains the dominant test for schedulability as it is both a simple and an intuitive test. Given this, a resource allocation scheme wherein schedulability was more closely tied to processor utilization (or a similarly intuitive metric) would be highly desirable.

4.3 A Taxonomy of Rate-Based Allocation Models

The genesis of rate-based resource allocation schemes can be traced to the problem of supporting multimedia computing and other soft real-time problems. In this arena, it was observed that while one could support the needs of these applications with traditional real-time scheduling models, these models were

not the most natural ones to be applied [5,7,11,28]. Whereas Liu and Layland models typically dealt with response time guarantees for the processing of periodic/sporadic events, the requirements of multimedia applications were better modeled as aggregate, but bounded, processing rates.

From our perspective three classes of rate-based resource allocation models have evolved: *fluid-flow allocation*, *server-based allocation*, and *generalized Liu and Layland style allocation*. Fluid-flow allocation derives largely from the work on fair-share bandwidth allocation in the networking community. Algorithms such as *generalized processor sharing* (GPS) [20] and *packet-by-packet generalized processor sharing* (PGPS) [20] (better known as *weighted fair queuing* (WFQ) [4]) were concerned with allocating network bandwidth to connections (“flows”) such that for a particular definition of fairness, all connections continuously receive their fair share of the bandwidth. Since connections were assumed to be continually generating packets, fairness was expressed in terms of a guaranteed transmission rate (i.e., some number of bits per second). These allocation policies were labeled as “fluid-flow” allocation because since transmission capacity was continuously available to be allocated, analogies were drawn between conceptually allowing multiple connections to transmit packets on a link and allowing multiple “streams of fluid” to flow through a “pipe.”

These algorithms stimulated tremendous activity in both real-time CPU and network link scheduling. In the CPU scheduling realm numerous algorithms were developed, differing largely in their definition and realization of “fair allocation” [19,26,28]. Although fair/fluid allocation is in principle a distinct concept from real-time allocation, it is a powerful building block for realizing real-time services [25].

Server-based allocation derives from the problem of scheduling aperiodic tasks in a real-time system. The salient abstraction is that a “server process” is invoked periodically to service any requests for work that have arrived since the previous invocation of the server. The server typically has a “capacity” for servicing requests (usually expressed in units of CPU execution time) in any given invocation. Once this capacity is exhausted, any in-progress work is suspended until at least the next server invocation time. Numerous server algorithms have appeared in the literature, differing largely in the manner in which the server is invoked and how its capacity is allocated [1,23,24]. Server algorithms are considered to be rate-based forms of allocation as the execution of a server is not (in general) directly coupled with the arrival of a task. Moreover, server-based allocation has the effect of ensuring aperiodic processing progresses at a well-defined, uniform rate.

Finally, rate-based generalizations of the original Liu and Layland periodic task model have been developed to allow more flexibility in how a scheduler responds to events that arrive at a uniform, average rate but unconstrained, instantaneous rate. Representative examples here include the (m, k) allocation models that require only m out of every k events to be processed in real time [6], the *window-based allocation* (DWYQ) method that ensures a minimum number of events are processed in real time within sliding time windows [29], and the RBE algorithm that “reshapes” the deadlines of events that arrive at a higher than expected rate to be those that the events would have had had they arrived at a uniform rate [8].

To illustrate the utility of rate-based resource allocation, one algorithm from the literature from each class of rate-based allocation methods will be discussed in more detail. The choice is motivated by our prior work, specifically experience gained implementing and using these algorithms in production systems [9,10]. For an instance of the fluid-flow paradigm, the proportional share scheduling algorithm EEVDF [26] will be discussed. For an instance of the polling server paradigm the CBS server concept [1] will be discussed. For the generalized Liu and Layland paradigm the RBE model [8] will be discussed. Although specific algorithms are chosen for discussion, ultimately the results presented are believed to hold for each algorithm in the same class as the algorithm discussed.

4.3.1 Fluid-Flow Resource Allocation: Proportional Share Scheduling

In a PS system, each shared resource r is allocated in discrete quanta of size at most q_r . At the beginning of each time quantum a task is selected to use the resource. Once the task acquires the resource, it may use the resource for the entire time quantum, or it may release the resource before the time quantum expires. For a given resource, a *weight* is associated with each task that determines the relative *share* of the resource

that the task should receive. Let w_i denote the weight of task i , and let $A(t)$ be the set of all tasks active at time t . Define the (instantaneous) share $f_i(t)$ of task i at time t as

$$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j} \quad (4.1)$$

A share represents a fraction of the resource's capacity that is "reserved" for a task. If the resource can be allocated in arbitrarily small-sized quanta, and if the task's share remains constant during any time interval $[t_1, t_2]$, then the task is entitled to use the resource for $(t_2 - t_1)f_i(t)$ time units in the interval. As tasks are created/destroyed or blocked/released, the membership of $A(t)$ changes and hence the denominator in Equation 4.1 changes. Thus in practice, a task's share of a given resource will change over time. As the total weight of tasks in the system increases, each task's share of the resource decreases. As the total weight of tasks in the system decreases, each task's share of the resource increases. When a task's share varies over time, the service time S that task i should receive in any interval $[t_1, t_2]$ is

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(t) dt \quad (4.2)$$

time units.

Equations 4.1 and 4.2 correspond to an ideal "fluid-flow" system in which the resource can be allocated for arbitrarily small units of time. In such a system, tasks make progress at a uniform rate as if they were executing on a dedicated processor with a capacity that is $f_i(t)$ that of the actual processor. In practice, one can implement only a discrete approximation to the fluid system. When the resource is allocated in discrete time quanta, it is not possible for a task to always receive exactly the service time it is entitled to at all time intervals. The difference between the service time that a task should receive at a time t and the time it actually receives is called the service time lag (or simply lag). Let t_0^i be the time at which task i becomes active, and let $s(t_0^i, t)$ be the service time task i receives in the interval $[t_0^i, t]$. Then if task i is active in the interval $[t_0^i, t]$, its lag at time t is defined as

$$\text{lag}_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t) \quad (4.3)$$

Since the lag quantifies the allocation accuracy, it is used as the primary metric for evaluating the performance of PS scheduling algorithms. One can schedule a set of tasks in a PS system using a "virtual time" *earliest deadline first* rule such that the lag is bounded by a constant over all time intervals [26]. By using this algorithm, called EEVDF, a PS system's deviation from a system with perfectly uniform allocation is bounded and thus, as explained below, real-time execution is possible.

4.3.1.1 Scheduling to Minimize Lag

The goal in PS scheduling is to minimize the maximum possible lag. This is done by conceptually tracking the lag of tasks and at the end of each quantum, considering only those tasks whose lag is positive [26]. If a task's lag is positive then it is "behind schedule" compared to the perfect fluid system. That is, the task should have accumulated more time on the CPU than it has up to the current time. If a task's lag is positive it is considered eligible to execute. If its lag is negative, then the task has received more processor time than it should have up to the current time and it is considered ineligible to execute.

When multiple tasks are eligible in EEVDF they are scheduled earliest deadline first, where a task's deadline is equal to its estimated execution time cost divided by its share of the CPU, $f_i(t)$. This deadline represents a point in the future when the task should complete execution if it receives exactly its share of the CPU. For example, if a task's weight is such that its share of the CPU at the current time is 10% and it requires 2 ms of CPU time to complete execution, then its deadline will be 20 ms in the future. If the task actually receives 10.

PS allocation is realized through a form of weighted round-robin scheduling, wherein in each round the task with the earliest deadline is selected. In Ref. 26 it was shown that the EEVDF algorithm provides optimal (i.e., minimum possible) lag bounds.

4.3.1.2 Realizing Real-Time Execution

In principle, there is nothing “real time” about PS resource allocation. PS resource allocation is concerned solely with *uniform* allocation (often referred to in the literature as “fluid” or “fair” allocation). A PS scheduler achieves uniform allocation if it can guarantee that tasks’ lags are always bounded.

Real-time computing is achieved in a PS system by (i) ensuring that a task’s share of the CPU (and other required resources) remains constant over time and (ii) scheduling tasks such that each task’s lag is always bounded by a constant. If these two conditions hold over an interval of length t for a task i , then task i is guaranteed to receive $(f_i \times t) \pm \epsilon$ units of the resource’s capacity, where f_i is the fraction of the resource reserved for task i and ϵ the allocation error, $0 \leq \epsilon \leq \delta$, for some constant δ (for EEVDF, δ = the quantum size q) [26]. Thus, although real-time allocation is possible, it is not possible to provide hard-and-fast guarantees of adherence to application-defined timing constraints. In other words, all guarantees have an implicit and fundamental “ $\pm\epsilon$ ” term. In FreeBSD-based implementations of EEVDF, ϵ has been a tunable parameter and was most commonly set to 1 ms [10].

The deadline-based EEVDF PS scheduling algorithm ensures that each task’s lag is bounded by a constant [26] (condition (i)). To ensure a task’s share remains constant over time (condition (ii)), whenever the total weight in the system changes, a “real-time” task’s weight must be adjusted so that its initial share (as given by Equation 4.1) does not change. For example, if the total weight in the system increases (e.g., because new tasks are created), then a real-time task’s weight must increase by a proportional amount. Adjusting the weight to maintain a constant share is simply a matter of solving Equation 4.1 for w_i when $f_i(t)$ is a constant function. (Note that w_i appears in both the numerator and denominator of the right-hand side of Equation 4.1.) If the sum of the processor shares of the real-time tasks is less than 1.0 then all tasks will execute in real time (i.e., under EEVDF, real-time schedulability is a simple function of processor utilization).

4.3.2 Liu and Layland Extensions: Rate-Based Execution

The traditional Liu and Layland model of periodic real-time execution has been extended in a number of directions to be more flexible in a way in which real-time requirements were modeled and realized. For example, all of the traditional theory assumes a minimum separation in time between releases of instances of the same task. This requirement does not map well in actual systems that, for example, receive inputs over a network. For example, in a video-processing application, video frames may be transmitted across an internetwork at precise intervals but arrive at a receiver with arbitrary spacing in time because of the store-and-forward nature of most network switches. Although there is explicit structure in this problem (frames are generated at a precise rate), there is no way to capture this structure in a Liu and Layland model.

The RBE paradigm is one extension to the Liu and Layland model to address this problem. In RBE, each task is associated with three parameters (x, y, d) , which define a rate specification. In an RBE system, each task is guaranteed to process at least x events every y time units, and each event j will be processed before a relative deadline d . The actual deadline for processing of the j th event for task i is given by the following recurrence. If t_{ij} is the time of the arrival of the j th event, then the instance of task i servicing this event will complete execution before time:

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{if } 1 \leq j \leq x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \quad (4.4)$$

The deadline for the processing of an event is the larger of the release time of the job plus its relative deadline, or the deadline of the x th previous job plus the y parameter (the averaging interval) of the task. This deadline assignment function confers two important properties on RBE tasks. First, up to x consecutive jobs of a task may contend for the processor with the same deadline and second, the deadlines for processing events j and $j + x$ for task i are separated by at least y time units. Without the latter restriction, if a set

of events for a task arrives simultaneously it would be possible to saturate the processor. However, with the restriction, the time at which a task must complete its execution is not wholly dependent on its release time. This is done to bound processor demand. Under this deadline assignment function, requests for tasks that arrive at a faster rate than x arrivals every y time units have their deadlines postponed until the time they would have been assigned had they actually arrived at the rate of exactly x arrivals every y time units [8].

The RBE task model derives from the *linear bounded arrival process* (LBAP) model as defined and used in the DASH system [2]. In the LBAP model, tasks specify a desired execution rate as the number of messages to be processed per second, and the size of a buffer pool used to store bursts of messages that arrive for the task. The RBE model generalizes the LBAP model to include a more generic specification of rate and adds an independent response time (relative deadline) parameter to enable more precise real-time control of task executions.

RBE tasks can be scheduled by a simple *earliest deadline first* rule as long as the combined processor utilization of all tasks does not saturate the processor. (Hence, there are no undue limits on the achievable processor utilization.) Although nothing in the statement of the RBE model precludes static priority scheduling of tasks, it turns out that the RBE model points out a fundamental distinction between deadline-based scheduling methods and static priority-based methods. Analysis of the RBE model has shown that under deadline-based scheduling, feasibility is solely a function of the distribution of task deadlines in time and is independent of the rate at which tasks are invoked. In contrast, the opposite is true of static priority schedulers. For *any* static priority scheduler, feasibility is a function of the rate at which tasks are invoked and is independent of the deadlines of the tasks [8]. In simple terms, the feasibility of static priority schedulers is solely a function of the periodicity of tasks, while the feasibility of deadline schedulers is solely a function of the periodicity of the occurrence of a task's deadlines. Given that it is often the operating system or application that assigns deadlines to tasks, this means that the feasibility of a static priority scheduler is a function of the behavior of the external environment (i.e., arrival processes), while the feasibility of a deadline-driven scheduler is a function of the implementation of the operating system/application. This is a significant observation as one typically has more control over the implementation of the operating system than they do over the processes external to the system that generate work for the system. For this reason, deadline-based scheduling methods have a significant and fundamental advantage over static priority-based methods when there is uncertainty in the rates at which work is generated for a real-time system, such as is the case in virtually all distributed real-time systems.

4.3.3 Server-Based Allocation: The Constant Bandwidth Server

The final class of rate-based resource allocation algorithms is server algorithms. At a higher level, the CBS algorithm combines aspects of both EEVDF and RBE scheduling (although it was developed independent of both works). Like RBE, CBS is an event-based scheduler, however, like EEVDF CBS also has a notion of a quantum. Like both, it achieves rate-based allocation by a form of deadline scheduling.

In CBS, and its related cousin the *total bandwidth server* (TBS) [23,24], a portion of the processor's capacity, denoted U_S , is reserved for processing aperiodic requests of a task. When an aperiodic request arrives it is assigned a deadline and scheduled according to the *earliest deadline first* algorithm. However, while the server executes, its capacity linearly decreases. If the server's capacity for executing a single request is exhausted before the request finishes, the request is suspended until the next time the server is invoked.

A server is parameterized by two additional parameters C_S and T_S , where C_S is the execution time available for processing requests in any single-server invocation and T_S is the interinvocation period of the server ($U_S = C_S/T_S$). Effectively, if the k th aperiodic request arrives at time t_k , it will execute as a task with a deadline

$$d_k = \max(t_k + D_{k-1}) + c_k/U_S \quad (4.5)$$

where c_k is the worst-case execution time of the k th aperiodic request, d_{k-1} the deadline of the previous request from this task, and U_S the processor capacity allocated to the server for this task.

CBS resource allocation is considered a rate-based scheme because deadlines are assigned to aperiodic requests based on the rate at which the server can serve them and not, e.g., on the rate at which they are expected to arrive. Note that like EEVDF and RBE, scheduability in CBS is solely a function of processor utilization. Any real-time task set that does not saturate the processor is guaranteed to execute in real time under any of these three algorithms.

4.4 Using Rate-Based Scheduling

To see how rate-based resource allocation methods can be used to realize real-time constraints and overcome the shortcoming of static priority scheduling, the three algorithms above were used to solve various resource allocation problems that arose in FreeBSD UNIX when executing a set of interactive multimedia applications. The details of this study are reported in Ref. 9. The results are summarized here.

4.4.1 A Sample Real-Time Workload

To compare the rate-based schedulers, three simple multimedia applications were considered. These applications were chosen because they illustrate many of the essential resource allocation problems found in distributed real-time and embedded applications. The applications are

- An Internet telephone application that handles incoming 100 byte audio messages at a rate of 50 per second and computes for 1 ms on each message (requiring 5% of the CPU on average).
- A motion-JPEG video player that handles incoming 1470 byte messages at a rate of 90 per second and computes for 5 ms on each message (requiring 45% of the CPU on average).
- A file-transfer program that handles incoming 1470 byte messages at a rate of 200 per second and computes for 1 ms on each message (requiring 20% of the CPU on average).

The performance of different rate-based resource allocation schemes was considered under varying workload conditions. The goal was to evaluate how each rate-based allocation scheme performed when the rates of tasks to be scheduled varied from constant (uniform) to “bursty” (erratic instantaneous rate but constant average rate) to “misbehaved” (long-term deviation from average expected processing rate).

4.4.2 Rate-Based Scheduling of Operating System Layers

Our experiments focused on the problem of processing inbound network packets and scheduling user applications to consume these packets. Figure 4.1 illustrates the high-level architecture of the FreeBSD kernel. Briefly, in FreeBSD, packet processing occurs as follows. (For a more complete description of these functions see Ref. 30.) When packets arrive from the network, interrupts from the network interface card are serviced by a device-specific interrupt handler. The device driver copies data from buffers on the adapter card into a chain of fixed-size kernel memory buffers sufficient to hold the entire packet. This chain of buffers is passed on a procedure call to a general interface input routine for a class of input devices. This procedure determines which network protocol should receive the packet and enqueues the packet on that protocol’s input queue. It then posts a software interrupt that will cause the protocol layer to be executed when no higher-priority hardware or software activities are pending.

Processing by the protocol layer occurs asynchronously with respect to the device driver processing. When the software interrupt posted by the device driver is serviced, a processing loop commences wherein on each iteration the buffer chain at the head of the input queue is removed and processed by the appropriate routines for the transport protocol. This results in the buffer chain enqueued on the receive queue for the destination socket. If any process is blocked in a kernel system call awaiting input on the socket, it is unblocked and rescheduled. Normally, software interrupt processing returns when no more buffers remain on the protocol input queue.

The kernel socket layer code executes when an application task invokes some form of receive system call on a socket descriptor. When data exist on the appropriate socket queue, the data are copied into the

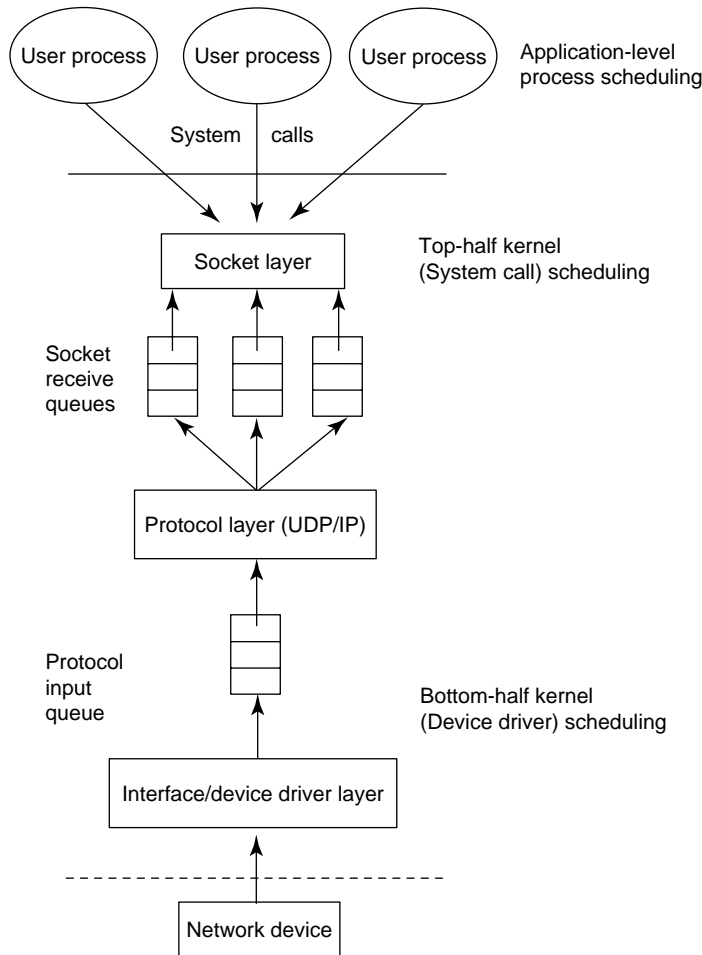


FIGURE 4.1 Architectural diagram of UDP/IP protocol processing in FreeBSD.

receiving task's local buffers from the buffer chain(s) at the head of that socket's receive queue. When there is sufficient data on the socket receive queue to satisfy the current request, the kernel completes the system call and returns to the application task.

The problem of processing inbound network packets was chosen for study because it involves a range of resource allocation problems at different layers in the operating system. Specifically, there are three distinct scheduling problems: scheduling of device drivers and network protocol processing within the operating system kernel, scheduling system calls made by applications to read and write data to and from the network, and finally the scheduling of user applications. These are distinct problems because the schedulable work is invoked in different ways in different layers. Asynchronous events cause device drivers and user applications to be scheduled, but synchronous events cause system calls to be scheduled. Systems calls are, in essence, extensions of the application's thread of control into the operating system. Moreover, these problems are of interest because of the varying amount of information that is available to make real-time scheduling decisions at each level of the operating system. At the application and system call-level it is known exactly which real-time entity should be "charged" for use of system resources, while at the device driver-level one cannot know which entity to charge. For example, in the case of inbound packet processing, it cannot be determined which application to charge for the processing of a packet until the packet is actually processed and the destination application is discovered. In contrast, the cost of device

processing can be known exactly as device drivers typically perform simple, bounded-time functions (such as placing a string of buffers representing a packet on a queue). This is in contrast to the application level where often one can only estimate the time required for an application to complete.

The challenge is to allocate resources throughout the operating system so that end-to-end system performance measures (i.e., network interface to application performance) can be ensured.

4.4.3 Workload Performance under Proportional Share Allocation

A version of FreeBSD was constructed that used EEVDF PS scheduling (with a 1 ms quantum) at the device, protocol processing, and the application layers. In this system, each real-time task is assigned a share of the CPU equal to its expected utilization of the processor (e.g., the Internet phone application requires 5% of the CPU and hence is assigned a weight of 0.05). Non-real-time tasks are assigned a weight equal to the unreserved capacity of the CPU. Network protocol processing is treated as a real-time (kernel-level) task that processes a fixed number of packets when it is scheduled for execution.

In the well-behaved sender's case, all packets are moved from the network interface to the socket layer to the application and processed in real time. When the file-transfer sender misbehaves and sends more packets than the ftp receiver can process given its CPU reservation, EEVDF does a good job of isolating the other well-behaved processes from the ill effects of ftp. Specifically, the excess ftp packets are dropped at the lowest level of the kernel (at the IP layer) before any significant processing is performed on these packets. These packets are dropped because the kernel task associated with their processing is simply not scheduled often enough (by design) to move the packets up to the socket layer. In a static priority system (such as the unmodified FreeBSD system), network interface processing is the second highest-priority task. In this system, valuable time would be "wasted" processing packets up through the kernel only to have them discarded later (because of buffer overflow) because the application was not scheduled often enough to consume them.

The cost of this isolation comes in the form of increased packet processing overhead as now the device layer must spend time demultiplexing packets (typically a higher-layer function) to determine if they should be dropped. Performance is also poorer when data arrive for all applications in a bursty manner. This is an artifact of the quantum-based allocation nature of EEVDF. Over short intervals, data arrive faster than it can be serviced at the IP layer and the IP interface queue overflows. With a 1 ms quantum, it is possible that IP processing can be delayed for upwards of 8–10 ms and this time is sufficient for the queue to overflow in a bursty environment. This problem could be ameliorated to some extent by increasing the length of the IP queue, however, this would also have the effect of increasing the response time for packet processing.

4.4.4 Workload Performance under Generalized Liu and Layland Allocation

When RBE scheduling was used for processing at the device, protocol, and application layers, each task had a simple rate specification of $(1, p, p)$ (i.e., one event will be processed every p time units with a deadline of p), where p is the period of the corresponding application or kernel function.

Perfect real-time performance is realized in the well-behaved and bursty arrivals cases but performance is significantly poorer than EEVDF in the case of the misbehaved file-transfer application. On the one hand, RBE provides good isolation between the file-transfer and the other real-time applications, however, this isolation comes at the expense of the performance of non-real-time/background applications. On the other, unlike EEVDF, as an algorithm, RBE has no mechanism for directly ensuring isolation between tasks as there is no mechanism for limiting the amount of CPU time an RBE task consumes. All events in an RBE system are assigned deadlines for processing. When the work arrives at a faster rate than is expected, the deadlines for the work are simply pushed further and further out in time. Assuming sufficient buffering, all will eventually be processed.

In the FreeBSD system, packets are enqueued at the socket layer but with deadlines that are so large that processing is delayed such that the socket queue quickly fills and overflows. Because time is spent processing packets up to the socket layer that are never consumed by the application, the performance of non-real-time applications suffers. Because of this, had the real-time workload consumed a larger cumulative fraction of the CPU, RBE would not have isolated the well-behaved and misbehaved real-time applications.

(That is, the fact that isolation was observed in these experiments is an artifact of the specific real-time workload.)

Nonetheless, because the RBE scheduler assigns deadline to all packets, and because the system under study was not overloaded, RBE scheduling results in smaller response times for real-time events than seen under EEVDF (at the cost non-real-time task performance).

4.4.5 Workload Performance under Server-Based Allocation

When CBS server tasks were used for processing throughout the operating system and at the application layers, each server task was assigned a capacity equal to its application's/function's CPU utilization. The server's period was made equal to the expected interarrival time of data (packets). Non-real-time tasks were again assigned to a server with capacity equal to the unreserved capacity of the CPU.

As expected, performance is good when work arrives in a well-behaved manner. In the case of the misbehaved file transfer, CBS also does a good job of isolating the other well-behaved tasks. The excess ftp packets are dropped at the IP layer and thus little overhead is incurred. In the case of bursty senders, CBS scheduling outperforms EEVDF scheduling. This is because like RBE, scheduling of CBS tasks is largely event driven and hence CBS tasks respond quicker to the arrival of packets. Under EEVDF the rate at which the IP queue can be serviced is a function of the quantum size and the number of processes currently active (which determines the length of a scheduling "round" [4]). In general, these parameters are not directly related to the real-time requirements of applications. Under CBS, the service rate is a function of the server's period, which is a function of the expected arrival rate and thus is a parameter that is directly related to application requirements. For the choices of quantum size for EEVDF, and server period for CBS, good performance under CBS and poor performance under EEVDF results. However, we conjecture that is likely the case that these parameters could be tuned to reverse this result.

Although CBS outperforms EEVDF in terms of throughput, the results are mixed for response times for real-time tasks. Even when senders are well behaved some deadlines are missed under CBS. This results in a significant number of packets being processed later than with EEVDF or RBE scheduling. This is problematic since in these experiments the theory predicts that no deadlines should be missed. The cause of the problem here relates to the problem of accounting for the CPU time consumed when a CBS task executes. In the implementation of CBS, the capacity of a CBS task is updated only when the task sleeps or is awoken by the kernel. Because of this, many other kernel-related functions that interrupt servers (e.g., Ethernet driver execution) are inappropriately charged to CBS tasks and hence bias scheduling decisions. This accounting problem is fundamental to the server-based approach and cannot be completely solved without significant additional mechanism (and overhead).

4.5 Hybrid Rate-Based Scheduling

The results of applying a single rate-based resource allocation policy to the problems of device, protocol, and application processing were mixed. When processing occurs at rates that match the underlying scheduling model (e.g., when work arrives at a constant rate), all the policies considered achieved real-time performance. When work arrives for an application that exceeds the application's rate specification or resource reservation, then only the CBS server-based scheme and the EEVDF PS scheme provide true isolation between well-behaved and misbehaved applications. When work arrives in a bursty manner, the quantum-based nature of EEVDF leads to less-responsive protocol processing and more (unnecessary) packet loss. CBS performs better but suffers from the complexity of the CPU-time accounting problem that must be solved. RBE provides the best response times but only at the expense of decreased throughput for the non-real-time activities.

The solution to these problems is to apply different rate-based resource allocation schemes in different layers of the kernel to better match the solution to a particular resource allocation problem to the characteristics of the problem. Consider two hybrid rate-based FreeBSD systems [9]. In one system, EEVDF

scheduling was used for application and system call-level processing. This choice was made because the quantum nature of EEVDF, while bad for intrakernel resource allocation, is a good fit given the existing round-robin scheduling architecture in FreeBSD (and many other operating systems such as Linux). It is easy to implement and to precisely control and gives good real-time response when schedulable entities execute for long periods relative to the size of a quantum. In the second system, both CBS and RBE scheduling were used for device and protocol processing inside the kernel. Since the lower kernel layers operate more as an event-driven system, a scheduling method which takes into account the notion of event arrivals is appropriate. Both of these policies are also well-suited for resource allocation within the kernel because, in the case of CBS, it is easier to control the levels and degrees of preemption within the kernel and hence it is easier to account for CPU usage within the kernel (and hence easier to realize the results predicted by the CBS theory). In the case of RBE, processing within the kernel is more deterministic and hence RBE's inherent inability to provide isolation between tasks that require more computation than they reserved is less of a factor.

The results of experiments with these systems show that when work arrives at well-behaved rates both CBS + EEVDF and RBE + EEVDF systems perform flawlessly [7]. (Thus hybrid allocation performs no worse than the universal application of a single rate-based scheme throughout the system.) In the misbehaved ftp application case, both hybrid implementations provide good isolation, comparable to the best single-policy systems. However, in both hybrid approaches, response times and deadline miss ratios are now much improved. In the case of bursty arrivals, all packets are eventually processed and although many deadlines are missed, both hybrid schemes miss fewer deadlines than did the single-policy systems. Overall, the RBE + EEVDF system produces the lowest overall deadline miss ratios. While we do not necessarily believe this is a fundamental result (i.e., there are numerous implementation details to consider), it is the case that the polling nature of the CBS server tasks increases response times over the direct event scheduling method of RBE.

4.6 Summary and Conclusions

Static priority scheduling is a popular and useful method for implementing many types of real-time and embedded systems. However, there are many aspects of practical real-time systems for which static priority scheduling methods are a poor fit. In many cases, rate-based methods can provide an alternate means of resource allocation. For example, rate-based resource allocation schemes are a good fit for providing real-time services in distributed real-time and embedded systems. Allocation schemes exist that are a good fit for the scheduling architectures used in the various layers of a traditional monolithic UNIX kernel such as FreeBSD. Three such rate-based schemes were considered: the EEVDF fluid-flow paradigm, the CBS polling server paradigm, and the generalization of Liu and Layland scheduling known as RBE. These algorithms have been shown to result in effective real-time control for three scheduling problems found in layer-structured operating systems: application-level scheduling of user programs, scheduling the execution of system calls made by applications in the "top-half" of the operating system, and scheduling asynchronous events generated by devices in the "bottom-half" of the operating system. For each scheduling problem, we considered the problem of network packet and protocol processing for a suite of canonical multimedia applications. We tested each implementation under three workloads: a uniform rate packet-arrival process, a bursty arrival process, and a misbehaved arrival process that generates work faster than the corresponding application process can consume it.

When work arrives at rates that match the underlying scheduling model (the constant rate sender's case), all the policies considered achieve real-time performance. When work arrives that exceeds the application's rate specification, only the CBS server-based scheme and the EEVDF PS scheme provide isolation between well-behaved and misbehaved applications. When work arrives in a bursty manner, the quantum-based nature of EEVDF gives less-responsive protocol processing and more packet loss. CBS performs better but suffers from CPU-time accounting problems that result in numerous missed deadlines. RBE provides the best response times but only at the expense of decreased throughput for the non-real-time activities.

The solution here was the application of different rate-based resource allocation schemes in different layers of the kernel. Specifically, EEVDF PS scheduling of applications and system calls combined with either CBS servers or RBE tasks in the bottom-half of the kernel was shown to be effective. The quantum nature of EEVDF scheduling proves to be well suited to the FreeBSD application scheduling architecture and the coarser-grained nature of resource allocation in the higher layers of the kernel. The event-driven nature of RBE scheduling gives the best response times for packet and protocol processing. Moreover, the deterministic nature of lower-level kernel processing avoids the shortcomings observed when RBE scheduling is employed at the user level.

In summary, rate-based resource allocation can be effective in realizing the performance requirements and constraints of individual layers of an operating system kernel. All of the schemes tested worked well for application-level scheduling (the problem primarily considered by the developers of each algorithm). However, for intrakernel resource allocation, these schemes give significantly different results. By combining resource allocation schemes it is possible to alleviate specific shortcomings.

References

1. L. Abeni, G. Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 4–13.
2. D. Anderson, S. Tzou, R. Wahbe, R. Govindan, M. Andrews, Support for Live Digital Audio and Video, Proc. of the 10th International Conference on Distributed Computing Systems, Paris, France, May 1990, pp. 54–61.
3. A. Bavier, L.L. Peterson, BERT: A Scheduler for Best Effort and Real-Time Tasks, Technical Report, Department of Computer Science, Princeton University, 2001.
4. A. Demers, S. Keshav, S. Shenkar, Analysis and Simulation of a Fair Queueing Algorithm, Journal of Internetworking Research and Experience, October 1990, pp. 3–12.
5. P. Goyal, X. Guo, H. Vin, A Hierarchical CPU Scheduler for Multimedia Operating Systems, USENIX Symposium on Operating Systems Design and Implementation, Seattle, WA, October 1996, pp. 107–121.
6. M. Hamdaoui, P. Ramanathan, A Dynamic Priority Assignment Technique for Streams with (m,k) -Firm Deadlines, IEEE Transactions on Computers, April 1995.
7. K. Jeffay, D. Bennett, Rate-Based Execution Abstraction for Multimedia Computing, Proc. of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video, Durham, NH, April 1995, Lecture Notes in Computer Science, Vol. 1018, pp. 64–75, Springer, Heidelberg.
8. K. Jeffay, S. Goddard, A Theory of Rate-Based Execution, Proc. of the 20th IEEE Real-Time Systems Symposium, December 1999, pp. 304–314.
9. K. Jeffay, G. Lamastra, A Comparative Study of the Realization of Rate-Based Computing Services in General Purpose Operating Systems, Proc. of the 7th IEEE International Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, December 2000, pp. 81–90.
10. K. Jeffay, F.D. Smith, A. Moorthy, J. Anderson, Proportional Share Scheduling of Operating Systems Services for Real-Time Applications, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 480–491.
11. M.B. Jones, D. Rosu, M.-C. Rosu, CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, Proc. of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997, pp. 198–211.
12. J. Lehoczky, L. Sha, Y. Ding, The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, Proc. of the 10th IEEE Real-Time Systems Symposium, Santa Monica, CA, December 1989, pp. 166–171.
13. J. Leung, J. Whitehead, On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks, Performance Evaluation, Vol. 2, 1982, pp. 237–250.
14. J.W.S. Liu, Real-Time Systems, Prentice-Hall, 2000.

15. C.L. Liu, J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM*, Vol. 20, No. 1, January 1973, pp. 46–61.
16. LynxWorks, LynxOS and BlueCat real-time operating systems, <http://www.lynx.com/index.html>.
17. Mentor Graphics, VRTX Real-Time Operating System.
18. A.K.-L. Mok, Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment, Ph.D. Thesis, MIT, Dept. of EE and CS, MIT/LCS/TR-297, May 1983.
19. J. Nieh, M.S. Lam, Integrated Processor Scheduling for Multimedia, *Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, NH, April 1995, *Lecture Notes in Computer Science*, T.D.C. Little and R. Gusella, eds., Vol. 1018, Springer, Heidelberg.
20. A.K. Parekh and R.G. Gallager, A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks—The Single Node Case, *ACM/IEEE Transactions on Networking*, Vol. 1, No. 3, 1992, pp. 344–357.
21. pSOS+TM/68K Real-Time Executive, User's Manual, Motorola, Inc.
22. QNX Operating System, System Architecture and Neutrino System Architecture Guide, QNX Software Systems Ltd., 1999.
23. M. Spuri, G. Buttazzo, Efficient Aperiodic Service under the Earliest Deadline Scheduling, *Proc. of the 15th IEEE Real-Time Systems Symposium*, December 1994, pp. 2–11.
24. M. Spuri, G. Buttazzo, F. Sensini, Robust Aperiodic Scheduling Under Dynamic Priority Systems, *Proc. of the 16th IEEE Real-Time Systems Symposium*, December 1995, pp. 288–299.
25. I. Stoica, H. Abdel-Wahab, K. Jeffay, On the Duality between Resource Reservation and Proportional Share Resource Allocation, *Multimedia Computing and Networking '97*, SPIE Proceedings Series, Vol. 3020, February 1997, pp. 207–214.
26. I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, C. Plaxton, A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems, *Proc. of the 17th IEEE Real-Time Systems Symposium*, December 1996, pp. 288–299.
27. VxWorks Programmer's Guide, WindRiver System, Inc., 1997.
28. C.A. Waldspurger, W.E. Weihl, Lottery Scheduling: Flexible Proportional-Share Resource Management, *Proc. of the 1st Symposium on Operating System Design and Implementation*, November 1994, pp. 1–12.
29. R. West, K. Schwan, C. Poellabauer, Scalable Scheduling Support for Loss and Delay Constrained Media Streams, *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.
30. G.R. Wright, W.R. Stevens, *TCP/IP Illustrated*, Vol. 2, The Implementation, Addison-Wesley, Reading, MA, 1995.

5

Compositional Real-Time Schedulability Analysis

5.1	Introduction	5-1
5.2	Compositional Real-Time Scheduling Framework	5-3
	Scheduling Unit, Component, and Interface • System Models • Compositional Framework Overview • Problem Statement	
5.3	Workload Models	5-6
5.4	Resource Models	5-7
	Bounded-Delay Resource Model • Periodic Resource Model	
5.5	Schedulability Analysis	5-9
	Bounded-Delay Resource Model • Periodic Resource Model	
5.6	Schedulable Workload Utilization Bounds	5-12
	Bounded-Delay Resource Model • Periodic Resource Model	
5.7	Extension	5-16
	Supporting Sporadic Tasks • Supporting Interacting Tasks within a Component	
5.8	Conclusions	5-18

Insik Shin
University of Pennsylvania
Insup Lee
University of Pennsylvania

5.1 Introduction

A hierarchical scheduling framework has been introduced to support hierarchical resource sharing among applications under different scheduling services. The hierarchical scheduling framework can be generally represented as a tree of nodes, where each node represents an application with its own scheduler for scheduling internal workloads (threads), and resources are allocated from a parent node to its children nodes, as illustrated in Figure 5.1. Goyal et al. [9] first proposed a hierarchical scheduling framework for supporting different scheduling algorithms for different application classes in a multimedia system.

The hierarchical scheduling framework can be used to support multiple applications while guaranteeing independent execution of those applications. This can be correctly achieved when the system provides *partitioning*, where the applications may be separated functionally for fault containment and for compositional verification, validation, and certification. Such a partitioning prevents one partitioned function from causing a failure of another partitioned function.

The hierarchical scheduling framework is particularly useful in the domain of open systems [7], where applications may be developed and validated independently in different environments. For example, the hierarchical scheduling framework allows an application to be developed with its own scheduling algorithm

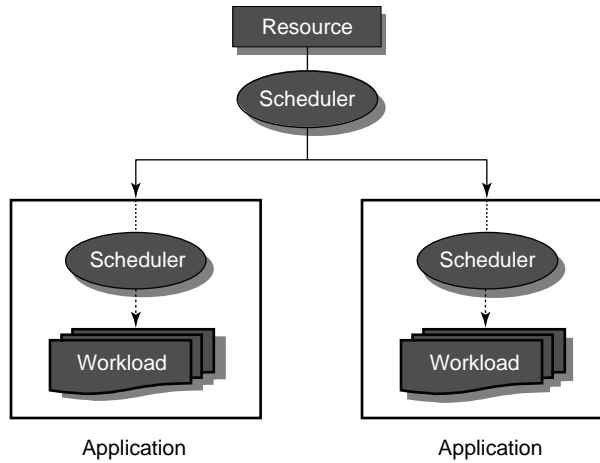


FIGURE 5.1 Hierarchical scheduling framework: A resource is scheduled by a scheduler and each share of the resource is subsequently scheduled by a different scheduler.

internal to the application and then later included in a system that has a different metalevel scheduler for scheduling applications.

In the real-time systems research community, there has been a growing attention to hierarchical scheduling frameworks [1,7,8,13,15,16,20,23–26]. Many studies have been proposed for the schedulability analysis of hierarchical scheduling frameworks. These studies commonly consider the Liu and Layland periodic task model [18], and earliest deadline first (EDF) and rate-monotonic (RM) scheduling.

Deng and Liu [7] proposed a two-level real-time scheduling framework for open systems. Kuo and Li [13] presented an exact schedulability condition for such a two-level framework with the RM system scheduler, under the assumption that all periodic tasks across components are harmonic. Lipari and Baruah [15,17] presented exact schedulability conditions for the two-level framework with the EDF system scheduler, while placing a server between the system scheduler and each component scheduler. The main role of the server is to compute the resource requirements of components at runtime so that the EDF system scheduler can use them in scheduling components. Each server is assumed to have knowledge of the task-level deadlines of its corresponding component. The common assumption shared by these previous approaches is that the system scheduler has a (schedulable) utilization bound of 100%. In open systems, however, it is desirable to be more general since there could be more than two levels and different schedulers may be used at different levels.

Mok et al. [21] introduced the notion of real-time resource model to specify the real-time resource allocation guarantees that a parent component provides to its child component in a hierarchical scheduling framework. This notion of resource model allows the parent and child components to be separated such that they interact with each other only through their resource model. This approach enables the parent and child components to have any different schedulers and allows the schedulability of the child component to be analyzed independent of its context. As a real-time resource model, they proposed the bounded-delay resource model and presented sufficient schedulability conditions for this proposed resource model [8,21,22]. Shin and Lee [26] presented the utilization bounds of the bounded-delay resource model under RM and EDF scheduling.

Another real-time resource model, the periodic resource model [25], has been proposed to specify the periodic behavior of resource allocations provided to a component from its parent component in a hierarchical scheduling framework. There has been work [1,4,16,24,25] on the schedulability analysis of a component involving the periodic resource model. Saewong et al. [24] introduced a worst-case response time analysis technique under RM scheduling. Almeida and Pedreiras [1] and Davis and Burns [4] have extended this initial technique with the notion of worst-case response time of periodic resource model;

they refined the technique for a case where the worst-case response time of periodic resource model is known. Lipari and Bini [16] and Shin and Lee [25] presented other RM schedulability conditions based on time demand calculation. Shin and Lee [25] presented an EDF schedulability condition. The utilization bounds of periodic resource models have been introduced under RM scheduling [24,25] and under EDF scheduling [25], respectively.

Shin and Lee [25,26] proposed the compositional real-time scheduling framework, where global (system-level) timing properties can be established by composing together independently (specified and) analyzed local (component-level) timing properties. They introduced two problems, *component abstraction* and *composition*, for developing such a framework. Component abstraction is to abstract the collective real-time requirements of a component as a single real-time requirement, called *real-time interface*. Component composition is to compose independently analyzed local timing properties (represented as real-time interfaces) into a global timing property.

The common assumption shared by the aforementioned studies is that all periodic tasks are independent. Matic and Henzinger [20] considered the issue of addressing the component abstraction problem in the presence of interacting tasks within a component. They considered two approaches, the real-time workshop (RTW) [19] and the logical execution time (LET) [11] semantics, for supporting tasks with intra- and intercomponent data dependencies. They showed that previous studies [25] on hierarchical scheduling frameworks can be extended for supporting tasks with both intra- and intercomponent dependencies when one of the two approaches is employed. They also showed that these two approaches produce a trade-off between the end-to-end latency and the component abstraction overhead.

There have been studies on the development of interface theory for supporting incremental design of component-based real-time systems, applying the interface theories [5,6] into real-time context. Wandeler and Thiele [29,30] proposed an assume-guarantee interface theory for real-time components with a generic real-time interface model. Henzinger and Matic [12] proposed another assume-guarantee interface theory using the bounded-delay resource partition model [8] as a real-time interface model. Both interface theories support associative composition of components, i.e., the incremental addition of new components, and the independent stepwise refinement of existing components.

Regehr and Stankovic [23] introduced another hierarchical scheduling framework that considers various kinds of real-time guarantees. Their work focused on converting one kind of guarantee to another kind of guarantee such that whenever the former is satisfied, the latter is also satisfied. With their conversion rules, the schedulability of a child component is sufficiently analyzed such that it is schedulable if its parent component provides real-time guarantees that can be converted to the real-time guarantee that the child component demands. However, they did not consider the component abstraction and composition problems.

5.2 Compositional Real-Time Scheduling Framework

This section presents an overview of the compositional real-time scheduling framework and identifies the problems that need to be addressed for the proposed framework. Section 5.2.1 defines terms and notions used in the proposed framework, Section 5.2.2 provides the system models and assumptions of the framework, Section 5.2.3 describes the framework, and Section 5.2.4 presents the problem statement.

5.2.1 Scheduling Unit, Component, and Interface

Scheduling assigns resources according to scheduling algorithm to service workloads. We use the term *scheduling unit* to mean the basic unit of scheduling and define scheduling unit S as a triple (W, R, A) , where W describes the workloads supported in the scheduling unit, R is a resource model that describes the resource allocations available to the scheduling unit, and A a scheduling algorithm that describes how the workloads share the resources at all times. For scheduling unit $S(W, R, A)$, we assume that the workload

W and the resource model R may not be synchronized. That is, suppose workloads in W start at time t_w and R begins providing resource allocations at time t_r . We assume t_w is not necessarily equal to t_r .

We consider that *component* C consists of a workload set W and a scheduling algorithm A for W , denoted as $C(W, A)$. The *resource demand* of component $C(W, A)$ represents the collective resource requirements that its workload set W requests under its scheduling algorithm A . The *demand bound function* $\text{dbf}_A(W, t)$ calculates the maximum possible resource demand that W could request to satisfy the timing requirements of task i under A within a time interval of length t .

Resource model R is said to be *dedicated* if it is exclusively available to a single scheduling unit, or *shared* otherwise. The *resource supply* of resource model R represents the amount of resource allocations that R provides. The *supply bound function* $\text{sbf}_R(t)$ calculates the minimum possible resource supplies that R provides during a time interval of length t . Resource model R is said to *satisfy* the resource demand of component $C(W, A)$ if $\text{dbf}_A(W, t, i) \leq \text{sbf}_R(t)$ for all task $i \in W$ and for all interval length $t > 0$.

We now define the schedulability of scheduling unit as follows: scheduling unit $S(W, R, A)$ is said to be *schedulable* if and only if the minimum possible resource supply of R can satisfy the maximum resource demand of W under A , i.e.,

$$\forall i \in W, \quad \forall t, \quad \text{dbf}_A(W, t, i) \leq \text{sbf}_R(t) \quad (5.1)$$

It should be noted that we define the schedulability of scheduling unit with the notion of worst-case resource supply of resource model R . Under the assumption that the workload W and the resource model R may not be synchronized, it is possible that W and R are aligned at the worst-case scenario, where W receives the minimum possible resource supply from R . Considering this, we define the schedulability such that all the timing requirements of W should be satisfiable under A even with the (worst-case) minimum possible resource supply of R .

The *real-time interface* I of a component $C(W, A)$ specifies the collective real-time requirements of the component C , which W demands under A , without revealing the internal information of the component such as the number of its workloads and its scheduling algorithm. A real-time interface I of component $C(W, A)$ is said to be *schedulable* if scheduling unit $S(W, R, A)$ is schedulable with $R = I$.

5.2.2 System Models

As a workload model, we consider the Liu and Layland periodic task model $T(p, e)$ [18], where p is a period and e an execution time requirement ($e \leq p$). The task utilization U_T of task T is defined as e/p . For a workload set $W = \{T_i\}$, a workload utilization U_W is defined as $\sum_{T_i \in W} U_{T_i}$. In this chapter, let P_{\min} denote the smallest task period in the workload set W . We consider that all tasks in a scheduling unit are synchronous, i.e., they release their initial jobs at the same time. We assume that each task is independent and preemptable with no preemption cost.

As a scheduling algorithm, we consider the EDF algorithm, which is an optimal dynamic uniprocessor scheduling algorithm [18], and the RM algorithm, which is an optimal fixed-priority uniprocessor scheduling algorithm [18].

As a resource model, we consider the bounded-delay resource model $\Phi(\alpha, \Delta)$ [21] and the periodic resource model $\Gamma(\Pi, \Theta)$ [25]. The bounded-delay resource model characterizes resource allocations with capacity α and bounded-delay Δ . The resource capacity U_Φ of $\Phi(\alpha, \Delta)$ is defined as α . The periodic resource model can characterize the periodic behavior of resource allocations, where Π is a resource period and Θ a periodic resource allocation time. The resource capacity U_Γ of Γ is defined as Θ/Π . Section 5.4 explains these resource models in detail.

As a real-time interface model, we consider the bounded-delay interface model $\mathcal{B}(\alpha, \Delta)$, where α is a resource utilization and Δ a bounded-delay, and the periodic interface model $\mathcal{P}(P, E)$, where P is a period and E an execution time requirement. The interface utilization $U_{\mathcal{B}}$ of \mathcal{B} is α , and the interface utilization $U_{\mathcal{P}}$ of \mathcal{P} is E/P .

5.2.3 Compositional Framework Overview

In a hierarchical scheduling framework, a parent component provides resource allocations to its child components. Once a child component C_1 finds a schedulable real-time interface \mathcal{I}_1 , it exports the real-time interface to its parent component. The parent component treats the real-time interface \mathcal{I}_1 as a single periodic task T_1 . As long as the parent component satisfies the resource requirements imposed by the single periodic task T_1 , the parent component is able to satisfy the resource demand of the child component C_1 . This scheme makes it possible for a parent component to supply resources to its child components without controlling (or even understanding) how the child components schedule resources for their own tasks.

We define the *component abstraction* problem as deriving a real-time interface of a component. That is, the problem is to abstract the collective real-time requirements of the component as a single real-time requirement, called the real-time interface, without revealing the internal structure of the component, e.g., the number of tasks and its scheduling algorithm. We formulate this problem as follows: given a component $C(W, A)$, the problem is to find an “optimal” real-time resource model R such that a scheduling unit $S(W, R, A)$ is schedulable. Then, we consider the optimal real-time resource model R as an optimal real-time interface \mathcal{I} of component $C(W, A)$. We define the optimality with respect to minimizing the resource capacity U_R of resource model R .

Example 5.1

As an example, let us consider component C_1 in Figure 5.2. Component C_1 has two periodic tasks under EDF scheduling, i.e., $C_1 = C(W_1, A_1)$, where $W_1 = \{T(40, 5), T(25, 4)\}$ and $A_1 = \text{EDF}$. Now, we consider the problem of finding an optimal schedulable periodic interface \mathcal{P}_1 of C_1 . This problem is equivalent to the problem of finding a periodic resource model $\Gamma^* = \Gamma(\Pi^*, \Theta^*)$ that makes scheduling unit $S_1 = S(W_1, \Gamma^*, A_1)$ schedulable with the minimum resource capacity U_{Γ^*} . When the period Π^* of Γ^* is given as 10, the optimal periodic resource model is $\Gamma^* = \Gamma(10, 3.1)$. Here, $\mathcal{P}_1 = \mathcal{P}(10, 3.1)$ is a solution to the component abstraction problem.

We define the *component composition* problem as combining multiple components into a single component through real-time interfaces, preserving the principle of compositionality, i.e., the properties of components hold in a larger component. We formulate the component composition problem as follows: given component $C_0 = C(W_0, A_0)$ that consists of two subcomponents C_1 and C_2 under A_0 , the problem is to find an “optimal” periodic interface \mathcal{P}_0 of C_0 . Our approach is to develop the optimal schedulable periodic interfaces \mathcal{P}_1 and \mathcal{P}_2 of C_1 and C_2 , respectively, and to consider $C_0 = C(W_0, A_0)$ as consisting

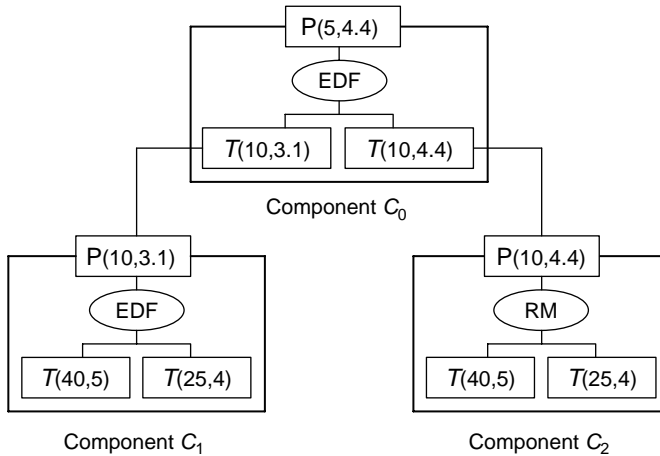


FIGURE 5.2 Compositional real-time scheduling framework.

of two periodic tasks, i.e., $W_0 = \{T_1, T_2\}$, where $T_i = \mathcal{P}_i$, $i = 1, 2$. The component composition problem then becomes equivalent to the component abstraction problem. So, we can address the component composition problem.

Example 5.2

As an example, let us consider component $C_0 = C(W_0, A_0)$ that consists of two subcomponents, C_1 and C_2 , in Figure 5.2. Let $A_0 = \text{EDF}$. We first determine the optimal schedulable periodic interfaces of the two subcomponents, C_1 and C_2 . An optimal schedulable periodic interface \mathcal{P}_1 of C_1 is $\mathcal{P}_1 = \mathcal{P}(10, 3.1)$, and that of C_2 is $\mathcal{P}_2 = \mathcal{P}(10, 4.4)$. By treating \mathcal{P}_i as a single periodic task T_i , for $i = 1, 2$, we can consider the workload set W_0 having two periodic tasks, i.e., $W = \{T_1, T_2\}$, where $T_1 = T(10, 3.1)$ and $T_2 = T(10, 4.4)$. Then, the problem of finding an optimal schedulable periodic interface \mathcal{P}_0 of C_0 is equivalent to the component abstraction problem for C_0 . Here, assuming the period Π^* of Γ^* is given as 5, $\Gamma^* = \Gamma(5, 4.4)$ is the optimal periodic resource model that makes scheduling unit $S(W_0, \Gamma^*, A_0)$ schedulable. Thus, $\mathcal{P}_0 = \mathcal{P}(5, 4.4)$ is a solution to the component composition problem for C_0 .

We define the *compositional real-time scheduling framework* as a hierarchical scheduling framework that supports the abstraction and composition of components. That is, it supports abstracting the collective real-time requirements of a component as a real-time interface and composing independently analyzed local timing properties into a global timing property.

5.2.4 Problem Statement

We consider the following problems: for a scheduling unit $S_0 = S(W_0, R_0, A_0)$, where $W_0 = \{T(p_i, e_i)\}$, $R_0 = \Phi(\alpha, \Delta)/\Gamma(\Pi_0, \Theta_0)$, and $A_0 = \text{EDF}/\text{RM}$, the problems are to (1) analyze its schedulability and (2) derive its schedulable workload utilization.

5.3 Workload Models

As a workload model of the proposed framework, we consider the standard real-time workload model, which is the Liu and Layland periodic model [18]. This model is defined as $T(p, e)$, where p is a period and e an execution time requirement ($e \leq p$). A task utilization U_T is defined as e/p . For a workload set $W = \{T_i\}$, a workload utilization U_W is defined as $\sum_{T_i \in W} U_{T_i}$. In this chapter, let P_{\min} denote the smallest task period in the workload set W .

For the periodic workload model $T_i(p_i, e_i)$, its demand bound function $\text{dbf}(T_i, t)$ can be defined as follows:

$$\text{dbf}(T_i, t) = \left\lfloor \frac{t}{p_i} \right\rfloor \cdot e_i$$

EDF scheduling. For a periodic task set W under EDF scheduling, Baruah et al. [2] proposed a *demand bound function* that computes the total resource demand $\text{dbf}_{\text{EDF}}(W, t)$ of W for every interval length t :

$$\text{dbf}_{\text{EDF}}(W, t) = \sum_{T_i \in W} \text{dbf}(T_i, t) \quad (5.2)$$

For an easier mathematical analysis, we can have a linear upper-bound function $\text{l dbf}_{\text{EDF}}(W, t)$ of the demand bound function $\text{dbf}_{\text{EDF}}(W, t)$ as follows:

$$\text{l dbf}_{\text{EDF}}(W, t) = U_W \cdot t \geq \text{dbf}_{\text{EDF}}(W, t)$$

where U_W is the utilization of the workload set W .

RM scheduling. For a periodic task set W under RM scheduling, Lehoczky et al. [14] proposed a demand bound function $\text{dbf}_{\text{RM}}(W, t, i)$ that computes the worst-case cumulative resource demand of a task T_i for an interval of length t .

$$\text{dbf}_{\text{RM}}(W, t, i) = \sum_{T_k \in \text{HP}_W(i)} \text{dbf}(T_k, t_k^*)$$

where

$$t_k^* = \left\lceil \frac{t}{p_k} \right\rceil \cdot p_k$$

where $\text{HP}_W(i)$ is a set of higher-priority tasks than T_i in W , including T_i . For a task T_i over a resource R , the worst-case response time $r_i(R)$ of T_i can be computed as follows:

$$r_i(R) = \min\{t\} \quad \text{such that } \text{dbf}_{\text{RM}}(W, t, i) \leq \text{sbf}_R(t)$$

5.4 Resource Models

In hierarchical scheduling frameworks, resources are allocated from a parent node to its child nodes. Resource models have been proposed to specify the resource allocations that a child node receives from its parent node. This section explains two real-time resource models: the bounded-delay resource model [21] and the periodic resource model [25].

5.4.1 Bounded-Delay Resource Model

Mok et al. [21] introduced the bounded-delay resource model $\Phi(\alpha, \Delta)$, where α is an available factor (resource capacity) ($0 < \alpha \leq 1$) and Δ a partition delay bound ($0 \leq \Delta$). The resource capacity U_Φ of bounded-delay resource model $\Phi(\alpha, \Delta)$ is defined as α . Intuitively, this bounded-delay resource model $\Phi(\alpha, \Delta)$ can be used to characterize resource allocations provided to a node, if the node receives the x -amount of resource supply during every time interval of length t such that $(t - \Delta)\alpha \leq x \leq (t + \Delta)\alpha$. That is, the bounded-delay resource model $\Phi(\alpha, \Delta)$ is defined to specify the following property:

$$\forall t_1, \quad \forall t_2 \geq t_1, \quad (t_2 - t_1 - \Delta)\alpha \leq \text{supply}_\Phi(t_1, t_2) \leq (t_2 - t_1 + \Delta)\alpha$$

where the supply function $\text{supply}_\Phi(t_1, t_2)$ computes the amount of resource allocations that the resource model R provides during the interval $[t_1, t_2]$. Figure 5.3 shows a bounded-delay resource example.

For a bounded-delay model Φ , its supply bound function $\text{sbf}_\Phi(t)$ is defined to compute the minimum resource supply for every interval length t as follows:

$$\text{sbf}_\Phi(t) = \begin{cases} \alpha(t - \Delta) & \text{if } (t \geq \Delta) \\ 0 & \text{otherwise} \end{cases} \quad (5.3)$$

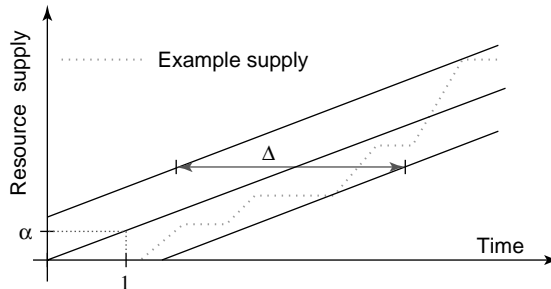


FIGURE 5.3 Bounded-delay model: example.

5.4.2 Periodic Resource Model

Shin and Lee [25] introduced the periodic resource model $\Gamma(\Pi, \Theta)^*$, where Π is a period ($\Pi > 0$) and Θ a periodic allocation time ($0 < \Theta \leq \Pi$). The resource capacity U_Γ of a periodic resource model $\Gamma(\Pi, \Theta)$ is defined as Θ/Π . The periodic resource model $\Gamma(\Pi, \Theta)$ is defined to characterize the following property:

$$\text{supply}_\Gamma(k\Pi, (k+1)\Pi) = \Theta, \quad \text{where } k = 0, 1, 2, \dots \quad (5.4)$$

For the periodic model $\Gamma(\Pi, \Theta)$, its supply bound function $\text{sbf}_\Gamma(t)$ is defined to compute the minimum possible resource supply for every interval length t as follows:

$$\text{sbf}_\Gamma(t) = \begin{cases} t - (k+1)(\Pi - \Theta) & \text{if } t \in [(k+1)\Pi - 2\Theta, (k+1)\Pi - \Theta] \\ (k-1)\Theta & \text{otherwise} \end{cases} \quad (5.5)$$

where $k = \max(\lceil (t - (\Pi - \Theta))/\Pi \rceil, 1)$. Figure 5.4 illustrates how the supply bound function $\text{sbf}_\Gamma(t)$ is defined for $k=3$.

The supply bound function $\text{sbf}_\Gamma(t)$ is a step function and, for easier mathematical analysis, we can define its linear lower bound function $\text{lsbf}_\Gamma(t)$ as follows:

$$\text{lsbf}_\Gamma(t) = \begin{cases} \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta)) & \text{if } (t \geq 2(\Pi - \Theta)) \\ 0 & \text{otherwise} \end{cases} \quad (5.6)$$

We define the *service time* of a resource model as a time interval that it takes for the resource to provide a resource supply. For the periodic resource model $\Gamma(\Pi, \Theta)$, we define its service time bound function $\text{tsbf}_\Gamma(t)$ that calculates the maximum service time of Γ required for a t -time-unit resource supply as follows:

$$\text{tsbf}_\Gamma(t) = (\Pi - \Theta) + \Pi \cdot \left\lfloor \frac{t}{\Theta} \right\rfloor + \epsilon_t \quad (5.7)$$

where

$$\epsilon_t = \begin{cases} \Pi - \Theta + t - \Theta \left\lfloor \frac{t}{\Theta} \right\rfloor & \text{if } (t - \Theta \left\lfloor \frac{t}{\Theta} \right\rfloor) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (5.8)$$

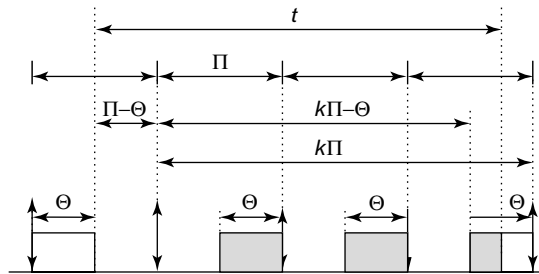


FIGURE 5.4 The supply bound function of a periodic resource model $\Gamma(\Pi, \Theta)$ for $k=3$.

*We note that the notion of periodic resource model has also been considered in other works [16,24].

5.5 Schedulability Analysis

For scheduling unit $S(W, R, A)$, schedulability analysis determines whether a set of timing requirements imposed by the workload set W can be satisfied with the resource supply of R under the scheduling algorithm A . The schedulability analysis is essential to solve the component abstraction and composition problems. In this section, we present conditions under which the schedulability of a scheduling unit can be satisfied when the scheduling unit consists of a periodic workload set scheduled by EDF or RM and bounded-delay or periodic resource models. We then show how to address the component abstraction and composition problems with bounded-delay or periodic interfaces using the schedulability conditions.

5.5.1 Bounded-Delay Resource Model

Mok et al. [21] suggested *virtual time scheduling* for scheduling tasks with the bounded-delay resource model. Virtual time scheduling is a scheduling scheme whereby schedules are performed according to virtual time instead of actual physical time. For any actual physical time t , its virtual time $V(t)$ over a bounded-delay resource model $\Phi(\alpha, \Delta)$ is defined as $\text{supply}_\Phi(t)/\alpha$. For any virtual time t , its actual time $P(t)$ is defined as the smallest time value such that $V(P(t)) = t$.

This virtual scheduling scheme can be applied to any scheduling algorithms such as EDF and RM. That is, under the virtual time EDF scheduling, the job with the earliest virtual time deadline has the highest priority. Then, we can have the virtual time EDF (VT-EDF) and virtual time RM (VT-RM).

Mok et al. [21] introduced the following theorem for schedulability analysis of a scheduling unit consisting of a periodic task set scheduled by virtual scheduling schemes with the bounded-delay resource model.

Theorem 5.1

Let W denote a periodic task set, $W = \{T_i(p_i, e_i)\}$, and $R_D(\alpha)$ denote a dedicated resource with capacity α . If scheduling unit $S(W, R_D(\alpha), A)$ is schedulable such that every job of W finishes its execution at least Δ earlier than its deadline, then scheduling unit $S(W, \Phi(\alpha, \Delta), A')$ is schedulable, where A' is the virtual scheduling scheme of A .

Sketch of proof

Consider a scheduling algorithm based on actual time being used over $R_D(\alpha)$ and its virtual time scheduling scheme being used over $\Phi(\alpha, \Delta)$. Then, if there is an event taking place at actual time t over $R_D(\alpha)$, this event takes place at virtual time t over $\Phi(\alpha, \Delta)$ as well. That is, if a job J_i finishes at time t over $R_D(\alpha)$, then the same job J_i finishes at virtual time t over $\Phi(\alpha, \Delta)$. For a virtual time t over $\Phi(\alpha, \Delta)$, its physical time $P(t)$ is given as $P(t) \in [t - \Delta, t + \Delta]$. That is, J_i will finish at actual time $t + \Delta$ at the latest over $\Phi(\alpha, \Delta)$. This means that if every job finishes Δ earlier than its deadline over $R_D(\alpha)$, then it is guaranteed to finish prior to its deadline over $\Phi(\alpha, \Delta)$.

Virtual time scheduling increases scheduling complexity by introducing the conversion between virtual time and physical time. Shin and Lee [26] introduced the following theorem for the schedulability analysis of scheduling units consisting of a periodic task set scheduled by EDF, rather than VT-EDF, over bounded-delay resource model.

Theorem 5.2

Scheduling unit $S(W, R, A)$, where $W = \{T_i(p_i, e_i)\}$, $R = \Phi(\alpha, \Delta)$, and $A = \text{EDF}$, is schedulable even with the worst-case resource supply of R if and only if

$$\forall t > 0, \quad \text{dbf}_{\text{EDF}}(W, t) \leq \text{sbf}_\Phi(t) \quad (5.9)$$

Proof

To show the necessity, we prove the contrapositive, i.e., if Equation 5.9 is false, there are some workload members of W that are not schedulable by EDF. If the total resource demand of W under EDF scheduling during t exceeds the total resource supply provided by Φ during t , then there is clearly no feasible schedule.

To show the sufficiency, we prove the contrapositive, i.e., if all workload members of W are not schedulable by EDF, then Equation 5.9 is false. Let t_2 be the first instant at which a job of some workload member T_i of W that misses its deadline. Let t_1 be the latest instant at which the resource supplied to W was idle or was executing a job whose deadline is after t_2 . By the definition of t_1 , there is a job whose deadline is before t_2 and which was released at t_1 . Without loss of generality, we can assume that $t = t_2 - t_1$. Since T_i misses its deadline at t_2 , the total demand placed on W in the time interval $[t_1, t_2)$ is greater than the total supply provided by Φ in the same time interval length t .

Example 5.3

Let us consider a workload set $W = \{T_1(100, 11), T_2(150, 22)\}$ and a scheduling algorithm $A = \text{EDF}$. The workload utilization U_W is 0.26. We now consider the problem of finding a schedulable bounded-delay interface $\mathcal{B}(\alpha, \Delta)$ of component $C(W, A)$. This problem is equivalent to finding a bounded-delay resource model $\Phi(\alpha, \Delta)$ that makes scheduling unit $S(W, \Phi(\alpha, \Delta), A)$ schedulable. We can obtain a solution space of $\Phi(\alpha, \Delta)$ by simulating Equation 5.9. For any given bounded-delay Δ , we can find the smallest α such that $S(W, \Phi(\alpha, \Delta), A)$ is schedulable according to Theorem 5.2. Figure 5.5a shows such a solution space as the gray area for $\Delta = 1, 10, 20, \dots, 100$. For instance, when $\Delta = 70$, the minimum resource capacity α that guarantees the schedulability of $S(W, \Phi(\alpha, \Delta), A)$ is 0.4. That is, the bounded-delay interface $\mathcal{B}(0.4, 70)$ is an optimal schedulable bounded-delay interface of $C(W, A)$, when $\Delta = 70$.

Shin and Lee [26] introduced the following theorem for the schedulability analysis of scheduling units consisting of a periodic task set scheduled by RM, rather than VT-RM, over bounded-delay resource model.

Theorem 5.3

Scheduling unit $S(W, R, A)$, where $W = \{T_i(p_i, e_i)\}$, $R = \Phi(\alpha, \Delta)$, and $A = \text{RM}$, is schedulable even with the worst-case resource supply of R if and only if

$$\forall T_i \in W \quad \exists t_i \in [0, p_i]: \quad \text{dbf}_{\text{RM}}(W, t_i, i) \leq \text{sbf}_{\Phi}(t_i) \quad (5.10)$$

Proof

Task T_i completes its execution requirement at time $t \in [0, p_i]$ if and only if all the execution requirements from all the jobs of higher-priority tasks than T_i and e_i , the execution requirement of T_i , are completed at time t_i .

The total of such requirements is given by $\text{dbf}_{\text{RM}}(W, t_i, i)$, and they are completed at t_i if and only if $\text{dbf}_{\text{RM}}(W, t_i, i) = \text{sbf}_{\Phi}(t_i)$ and $\text{dbf}_{\text{RM}}(W, t'_i, i) > \text{sbf}_{\Phi}(t'_i)$ for $0 \leq t'_i < t_i$. It follows that a

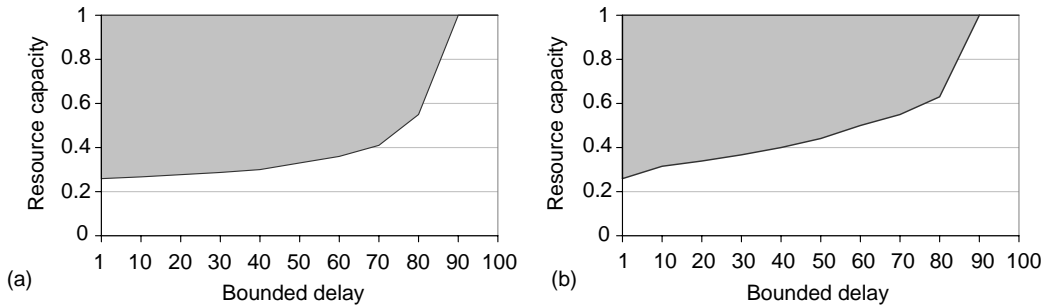


FIGURE 5.5 Example of solution space of a bounded-delay scheduling interface model $\Phi(\alpha, \Delta)$ for a workload set $W = \{T_1(100, 11), T_2(150, 22)\}$ under (a) EDF and (b) RM scheduling.

necessary and sufficient condition for T_i to meet its deadline is the existence of a $t_i \in [0, p_i]$ such that $\text{dbf}_{\text{RM}}(W, t_i, i) = \text{sbfc}_{\Phi}(t_i)$.

The entire task set is schedulable if and only if each of the tasks is schedulable. This means that there exists a $t_i \in [0, p_i]$ such that $\text{dbf}_{\text{RM}}(W, t_i, i) = \text{sbfc}_{\Phi}(t_i)$ for each task $T_i \in W$.

Example 5.4

Let us consider a workload set $W = \{T_1(100, 11), T_2(150, 22)\}$ and a scheduling algorithm $A = \text{RM}$. The workload utilization U_W is 0.26. We now consider the problem of finding a schedulable bounded-delay interface $\mathcal{B}(\alpha, \Delta)$ of component $C(W, A)$. This problem is equivalent to finding a bounded-delay resource model $\Phi(\alpha, \Delta)$ that makes scheduling unit $S(W, \Phi(\alpha, \Delta), A)$ schedulable. We can obtain a solution space of $\Phi(\alpha, \Delta)$ by simulating Equation 5.3. For any given bounded-delay Δ , we can find the smallest α such that $S(W, \Phi(\alpha, \Delta), A)$ is schedulable according to Theorem 5.3. Figure 5.5b shows such a solution space as the gray area for $\Delta = 1, 10, 20, \dots, 100$. For instance, when $\Delta = 40$, the minimum resource capacity α that guarantees the schedulability of $S(W, \Phi(\alpha, \Delta), A)$ is 0.4. That is, the bounded-delay interface $\mathcal{B}(0.4, 40)$ is an optimal schedulable bounded-delay interface of $C(W, A)$, when $\Delta = 40$.

5.5.2 Periodic Resource Model

Shin and Lee [25] present the following theorem to provide an exact condition under which the schedulability of scheduling unit $S(W, R, \text{EDF})$ can be satisfied for the periodic resource model R .

Theorem 5.4

Scheduling unit $S(W, R, A)$, where $W = \{T_i(p_i, e_i)\}$, $R = \Gamma(\Pi, \Theta)$, and $A = \text{EDF}$, is schedulable even with the worst-case resource supply of R if and only if

$$\forall 0 < t \leq \text{LCM}_W, \quad \text{dbf}_{\text{EDF}}(W, t) \leq \text{sbfc}_{\Phi}(t) \quad (5.11)$$

where LCM_W is the least common multiple of p_i for all $T_i \in W$.

Proof

The proof of this theorem can be obtained if Φ is replaced with Γ in the proof of Theorem 5.2.

The schedulability condition in Theorem 5.4 is necessary in addressing the component abstraction problem for a component with the EDF scheduling algorithm. We illustrate this with the following example.

Example 5.5

Let us consider a workload set $W_0 = \{T(50, 7), T(75, 9)\}$ and a scheduling algorithm $A_0 = \text{EDF}$. The workload utilization U_{W_0} is 0.26. We now consider the problem of finding a schedulable periodic interface $\mathcal{P}_0 = \mathcal{P}(P_0, E_0)$ of component $C_0 = C(W_0, A_0)$. This problem is equivalent to finding a periodic resource model $\Gamma_0 = \Gamma(\Pi_0, \Theta_0)$ that makes scheduling unit $S_0 = S(W_0, \Gamma_0, A_0)$ schedulable. We can obtain a solution space of Γ_0 to this problem by simulating Equation 5.11. For any given resource period Π_0 , we can find the smallest Θ_0 such that the scheduling unit S_0 is schedulable according to Theorem 5.4. Figure 5.6a shows such a solution space as the gray area for each integer resource period $\Pi_0 = 1, 2, \dots, 75$. For instance, when $\Pi_0 = 10$, the minimum resource capacity U_{Γ_0} that guarantees the schedulability of S_0 is 0.28. So, $\Theta_0 = 2.8$. That is, the periodic interface $\mathcal{P}_0 = \mathcal{P}(10, 2.8)$ is an optimal schedulable interface of C_0 , when P is given as 10.

Shin and Lee [25] present the following theorem to provide an exact condition under which the schedulability of scheduling unit $S(W, R, \text{RM})$ can be satisfied for the periodic resource model R .

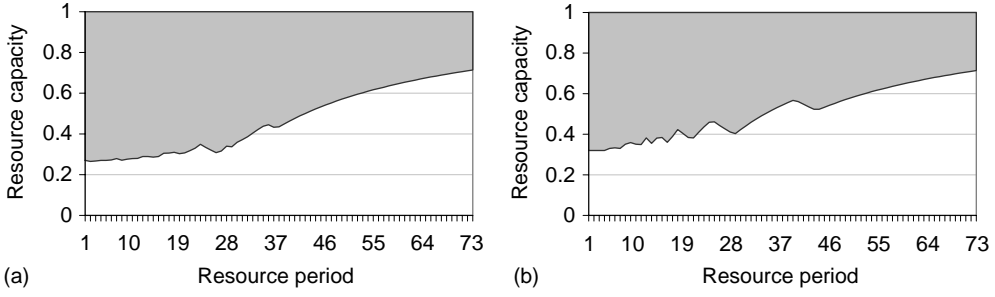


FIGURE 5.6 Schedulable region of periodic resource $\Gamma(\Pi, \Theta)$: (a) under EDF scheduling in Example 5.5 and (b) under RM scheduling in Example 5.6.

Theorem 5.5

Scheduling unit $S(W, R, A)$, where $W = \{T_i(p_i, e_i)\}$, $R = \Gamma(\Pi, \Theta)$, and $A = \text{RM}$, is schedulable even with the worst-case resource supply of R if and only if

$$\forall T_i \in W \quad \exists t_i \in [0, p_i] : \text{dbf}_{\text{RM}}(W, t_i, i) \leq \text{sbfr}_R(t_i) \quad (5.12)$$

Proof

The proof of this theorem can be obtained if Φ is replaced with Γ in the proof of Theorem 5.3.

We present the following example to show that using Theorem 5.5, we can address the component abstraction problem for a component with the RM scheduling algorithm.

Example 5.6

Let us consider a workload set $W_0 = \{T(50, 7), T(75, 9)\}$ and a scheduling algorithm $A_0 = \text{RM}$. The workload utilization U_{W_0} is 0.26. We now consider the problem of finding a schedulable periodic interface $\mathcal{P}_0 = \mathcal{P}(P_0, E_0)$ of component $C_0 = C(W_0, A_0)$. This problem is equivalent to finding a periodic resource model $\Gamma_0 = \Gamma(\Pi_0, \Theta_0)$ that makes scheduling unit $S_0 = S(W_0, \Gamma_0, A_0)$ schedulable. We can obtain a solution space of Γ_0 to this problem by simulating Equation 5.12. For any given resource period Π_0 , we can find the smallest Θ_0 such that the scheduling unit S_0 is schedulable according to Theorem 5.5. Figure 5.6b shows such a solution space as the gray area for each integer resource period $\Pi_0 = 1, 2, \dots, 75$. For instance, when $\Pi_0 = 10$, the minimum resource capacity U_{Γ_0} that guarantees the schedulability of S_0 is 0.35. So, $\Theta_0 = 3.5$. That is, the periodic interface $\mathcal{P}_0 = \mathcal{P}(10, 3.5)$ is an optimal schedulable interface of C_0 , when P is given as 10.

5.6 Schedulable Workload Utilization Bounds

This section starts by presenting the definition of (schedulable workload) utilization bound and provides the utilization bounds on the periodic resource model under EDF and RM scheduling.

Definition 5.1

The (schedulable workload) utilization bound, denoted as UB , of a periodic task set W on resource model R under scheduling algorithm A is defined such that the scheduling unit $S(W, R, A)$ is schedulable if the workload utilization (U_W) is no greater than the utilization bound.

The utilization bound is particularly suited for online acceptance tests. When checking whether or not a new periodic task can be scheduled with the existing tasks, computing the utilization bound takes a constant amount of time, which is much faster than doing an exact schedulability analysis based on a demand bound function.

Liu and Layland [18] presented the utilization bounds of a periodic task set W on a dedicated resource R_D under EDF and RM scheduling; the utilization bounds are 1 under EDF and $n(2^{1/n} - 1)$ under RM, respectively, where n is the number of tasks in W . This section extends these seminal results with the periodic resource model.

5.6.1 Bounded-Delay Resource Model

In this section, we present the utilization bounds of a bounded-delay resource model $\Phi(\alpha, \Delta)$ under EDF and RM scheduling. Shin and Lee [26] present the following theorem to introduce a utilization bound of a set of periodic tasks under EDF scheduling over $\Phi(\alpha, \Delta)$.

Theorem 5.6

Scheduling unit $S(W, R, A)$, where $W = \{T_i(p_i, e_i)\}$, $R = \Phi(\alpha, \Delta)$, and $A = \text{EDF}$, is schedulable if

$$U_W \leq \alpha \left(1 - \frac{\Delta}{P_{\min}} \right)$$

where $P_{\min} = \min_{T_i \in W} \{p_i\}$.

Proof

The possible values of a time interval length t fall into two ranges: (1) $0 < t < P_{\min}$ and (2) $P_{\min} \leq t$.

For the first case where $0 < t < P_{\min}$, from the definition of $\text{dbf}_{\text{EDF}}(W, t)$, we can see that

$$\forall 0 < t < P_{\min}, \quad \text{dbf}_{\text{EDF}}(W, t) = 0$$

Then, it is obvious that

$$\forall 0 < t < P_{\min}, \quad \text{dbf}_{\text{EDF}}(W, t) \leq \text{sbf}_{\Phi}(t) \quad (5.13)$$

For the second case where $P_{\min} \leq t$, we can see that since $\alpha \geq U_W$,

$$\forall t \geq t^*, \quad (\text{lbf}_W(t^*) \leq \text{sbf}_{\Phi}(t^*)) \rightarrow (\text{lbf}_W(t) \leq \text{sbf}_{\Phi}(t))$$

We have

$$\begin{aligned} \text{sbf}_{\Phi}(P_{\min}) &= \alpha(P_{\min} - \Delta) \\ &= P_{\min} \cdot \alpha \left(1 - \frac{\Delta}{P_{\min}} \right) \\ &\geq U_W \cdot P_{\min} \\ &= \text{lbf}_{\text{EDF}}(W, P_{\min}) \end{aligned}$$

Then, we can see that

$$\forall t \geq P_{\min}, \quad \text{dbf}_{\text{EDF}}(W, t) \leq \text{lbf}_{\text{EDF}}(W, t) \leq \text{sbf}_{\Phi}(t) \quad (5.14)$$

Now, we can see that a component $C(W, \Phi, \text{EDF})$ is schedulable according to Theorem 5.4.

Shin and Lee [26] present another theorem to introduce a utilization bound of a bounded-delay resource model $\Phi(\alpha, \Delta)$ for a set of periodic tasks under RM scheduling.

Theorem 5.7

Scheduling unit $S(W, R, A)$, where $W = \{T_1(p_1, e_1), \dots, T_n(p_n, e_n)\}$, $R = \Phi(\alpha, \Delta)$, and $A = \text{RM}$, is schedulable if

$$U_W \leq \alpha \left(n(\sqrt[n]{2} - 1) - \frac{\Delta}{2^{(n-1)/n} \cdot P_{\min}} \right)$$

where $P_{\min} = \min_{T_i \in W} \{p_i\}$.

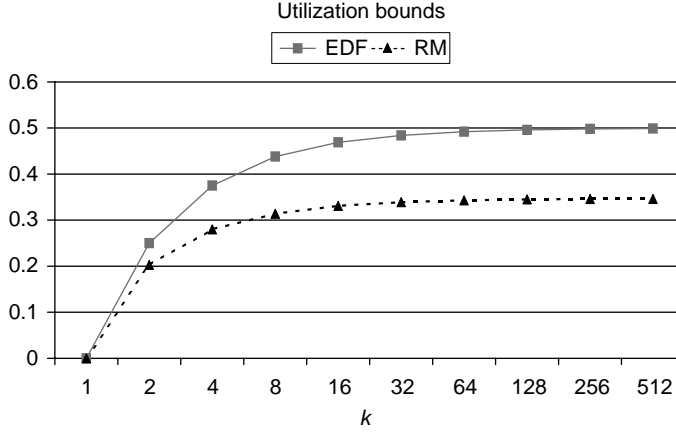


FIGURE 5.7 Utilization bounds of a bounded-delay resource model $\Phi(\alpha, \Delta)$, where $\alpha = 0.5$, as a function of k , where $k = P_{\min}/\Delta$, under EDF and RM scheduling.

Proof

Owing to the space limit, we refer to Ref. 27 for a full proof.

Figure 5.7 shows how the utilization bounds of the bounded-delay resource model grow with respect to k under EDF and RM scheduling, where k represents the relationship between the delay bound Δ and the smallest period in the task set P_{\min} , $k = P_{\min}/\Delta$. It is shown in the figure that as k increases, the utilization bounds converge to their limits that are α under EDF scheduling in Theorem 5.6 and $\log 2 \cdot \alpha$ under RM scheduling in Theorem 5.7.

5.6.2 Periodic Resource Model

We observe that a key relationship between a workload set W and a periodic resource model $\Gamma(\Pi, \Theta)$ is the *period multiple relationship*, which basically indicates how many times that $\Gamma(\Pi, \Theta)$ can provide W with the whole periodic allocation time of Θ during a time interval of length P_{\min} at the worst-case resource supply scenario, where P_{\min} is the smallest task period of W . Here, we define the functions, $K_A(P_{\min}, \Gamma(\Pi, \Theta))$, that computes this period multiple relationship under RM and EDF scheduling as follows:

- under RM scheduling, $K_{RM}(P_{\min}, \Gamma(\Pi, \Theta))$ is defined as follows:

$$K_{RM}(P_{\min}, \Gamma(\Pi, \Theta)) = \max\{k \text{ is Integer} \mid (k+1)\Pi - \Theta < P_{\min}\} \quad (5.15)$$

- under EDF scheduling, $K_{EDF}(P_{\min}, \Gamma(\Pi, \Theta))$ is defined as follows:

$$K_{EDF}(P_{\min}, \Gamma(\Pi, \Theta)) = \max \left\{ k \text{ is Integer} \mid (k+1)\Pi - \Theta - \frac{k\Theta}{k+2} < P_{\min} \right\} \quad (5.16)$$

Based on the period multiple relationship between $\Gamma(\Pi, \Theta)$ and W with P_{\min} , this section presents the utilization bounds of W on $\Gamma(\Pi, \Theta)$ as a function of P_{\min} under EDF scheduling and as a function of n and P_{\min} under RM scheduling.

For scheduling unit $S(W, R, A)$, where $W = \{T(p_i, e_i)\}$, $R = \Gamma(\Pi, \Theta)$, and $A = \text{EDF}$, we represent its utilization bound as a function of P_{\min} , denoted as $UB_{\Gamma, \text{EDF}}(P_{\min})$, such that $S(W, R, A)$ is schedulable if

$U_W \leq \text{UB}_{\Gamma, \text{EDF}}(P_{\min})$. Shin and Lee [28] present the following theorem to introduce the utilization bound $\text{UB}_{\Gamma, \text{EDF}}(P_{\min})$.

Theorem 5.8

For scheduling unit $S(W, R, A)$, where $W = \{T(p_i, e_i)\}$, $R = \Gamma(\Pi, \Theta)$, and $A = \text{EDF}$, its utilization bound $\text{UB}_{\Gamma, \text{EDF}}(P_{\min})$, as a function of P_{\min} , is

$$\text{UB}_{\Gamma, \text{EDF}}(P_{\min}) = \frac{k \cdot U_{\Gamma}}{k + 2(1 - U_{\Gamma})} \quad (5.17)$$

where $k = K_{\text{EDF}}(P_{\min}, R)$.

Proof

Owing to the space limit, we refer to Ref. 28 for a full proof.

It should be noted that the utilization bound $\text{UB}_{\Gamma, \text{EDF}}(P_{\min})$ becomes 1 without regard to P_{\min} if the resource capacity U_{Γ} of periodic resource model Γ is 1, i.e., Γ represents a dedicated resource. Thus, $\text{UB}_{\Gamma, \text{EDF}}(P_{\min})$ is a generalization of Liu and Layland's [18] result.

Example 5.7

As an example, we consider a scheduling unit $S_0 = S(W_0, \Gamma_0, A_0)$, where $\Gamma_0 = \Gamma(10, 4)$ and $A_0 = \text{EDF}$. Then, the resource capacity U_{Γ_0} of Γ_0 is 0.4. We assume that the smallest task period P_{\min} of the workload set W_0 is greater than 50, i.e., $P_{\min} \geq 50$. Here, k is 4 by the definition of $K_{\text{EDF}}(P_{\min}, R)$. According to Theorem 5.8, the EDF utilization bound $\text{UB}_{\Gamma_0, \text{EDF}}(50)$ is 0.32. That is, if $U_{W_0} \leq 0.32$, then the scheduling unit S_0 is schedulable.

Figure 5.8a shows the effect of resource period, in terms of k , on the utilization bound as a function of resource capacity under EDF scheduling, where $k = K_{\text{EDF}}(P_{\min}, \Gamma(\Pi, \Theta))$. The solid line, labeled “limit,” shows the limit of the utilization bound of a periodic resource, which is obtained when $k = \infty$. The other curves show the utilization bounds of a periodic resource when k is given as shown in the corresponding labels. It is shown that as k increases, the utilization bound of a periodic resource converges to its limit.

For scheduling unit $S(W, R, A)$, where $W = \{T(p_1, e_1), \dots, T(p_n, e_n)\}$, $R = \Gamma(\Pi, \Theta)$, and $A = \text{RM}$, Saewong et al. [24] represented its utilization bound as a function of n , denoted as $\text{UB}_{\Gamma, \text{RM}}(n)$. They presented the following result to introduce the utilization bound $\text{UB}_{\Gamma, \text{RM}}(n)$, derived from Liu and Layland's utilization bound [18].

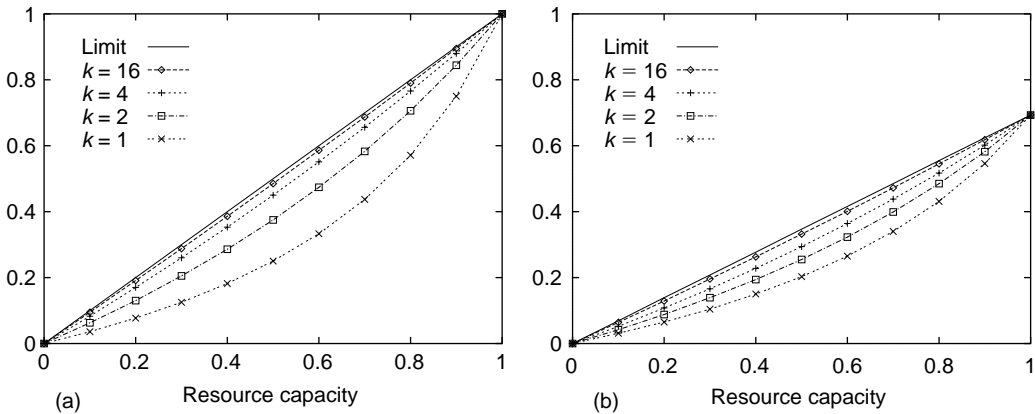


FIGURE 5.8 Utilization bound as a function of its resource capacity: (a) under EDF scheduling and (b) under RM scheduling.

Theorem 5.9

For scheduling unit $S(W, R, A)$, where $W = \{T(p_1, e_1), \dots, T(p_n, e_n)\}$, $R = \Gamma(\Pi, \Theta)$, $A = \text{RM}$, and $p_i \geq 2\Pi - \Theta$, $1 \leq i \leq n$, its utilization bound $\text{UB}_{\Gamma, \text{RM}}(n)$ is

$$\text{UB}_{\Gamma, \text{RM}}(n) = n \left[\left(\frac{3 - U_{\Gamma}}{3 - 2 \cdot U_{\Gamma}} \right)^{1/n} - 1 \right]$$

Proof

Owing to the space limit, we refer to Ref. 24 for a full proof.

Shin and Lee [28] present a different utilization bound as a function of n and P_{\min} , denoted as $\text{UB}_{\Gamma, \text{RM}}(n, P_{\min})$. They provide the following theorem to introduce $\text{UB}_{\Gamma, \text{RM}}(n, P_{\min})$.

Theorem 5.10

For scheduling unit $S(W, R, A)$, where $W = \{T(p_1, e_1), \dots, T(p_n, e_n)\}$, $R = \Gamma(\Pi, \Theta)$, $A = \text{RM}$, and $p_i \geq 2\Pi - \Theta$, $1 \leq i \leq n$, its utilization bound $\text{UB}_{\Gamma, \text{RM}}(n, P_{\min})$ is

$$\text{UB}_{\Gamma, \text{RM}}(n, P_{\min}) = U_{\Gamma} \cdot n \left[\left(\frac{2 \cdot k + 2(1 - U_{\Gamma})}{k + 2(1 - U_{\Gamma})} \right)^{1/n} - 1 \right] \quad (5.18)$$

where $k = K_{\text{RM}}(P_{\min}, R)$.

Proof

Owing to the space limit, we refer to Ref. 28 for a full proof.

It should be noted that the utilization bound $\text{UB}_{\Gamma, \text{RM}}(n, P_{\min})$ becomes Liu and Layland's RM utilization bound [18], which is $n(2^{1/n} - 1)$, without regard to P_{\min} if the capacity of periodic resource U_{Γ} is 1, i.e., the periodic resource is essentially a dedicated resource. Thus, $\text{UB}_{\Gamma, \text{RM}}(n, P_{\min})$ is a generalization of Liu and Layland's result [18].

Example 5.8

As an example, we consider a scheduling unit $S_0 = S(W_0, \Gamma_0, A_0)$, where $\Gamma_0 = \Gamma(10, 4)$ and $A_0 = \text{RM}$. Let $\Pi_0 = 10$ and $\Theta_0 = 4$. Then, the resource capacity U_{Γ_0} of Γ_0 is 0.4. We assume that the smallest task period P_{\min} of the workload set W_0 is greater than 50, i.e., $P_{\min} \geq 50$. Here, k is 4 according to Equation 5.18. According to Theorem 5.10, the RM utilization bound $\text{UB}_{\Gamma_0, \text{EDF}}(50)$ is 0.27. That is, if $U_{W_0} \leq 0.27$, then the scheduling unit S_0 is schedulable.

Figure 5.8b shows the effect of resource period, in terms of k , on the utilization bound as a function of resource capacity under RM scheduling, where $k = K_{\text{RM}}(P_{\min}, \Gamma(\Pi, \Theta))$. The solid line, labeled "limit," shows the limit of the utilization bound, which is obtained when $k = \infty$. The other curves show the utilization bound when k is given as shown in their labels. It is shown in the graph that as k increases, the utilization bound of a periodic resource converges to its limit.

5.7 Extension

So far, we have addressed the component abstraction problem under the assumptions that (1) the task model is the pure periodic task model and (2) tasks are independent. This section discusses the issues of extending our framework in relaxing these assumptions.

5.7.1 Supporting Sporadic Tasks

In addition to the periodic tasks that execute repeatedly at regular time intervals, real-time systems may consist of tasks whose release times are not known *a priori*, since they are to respond to asynchronous external events. The sporadic task model characterizes the workload generated in response to these events. Sporadic tasks are released with hard deadlines at random time instants with the minimum interarrival separation time between consecutive jobs. Here, we discuss how to extend our framework with the sporadic task model.

A sporadic task τ_i can be defined by a triple (e_i, d_i, s_i) , where e_i is a worst-case execution time requirement, d_i a relative deadline, and s_i a minimum interarrival separation between any two jobs of τ_i . Baruah et al. [2] have shown that the cumulative resource demands of jobs of τ_i over an interval $[t_0, t_0 + t)$ is maximized if it first arrives at the start of the interval (i.e., at time instant t_0) and subsequent jobs arrive as rapidly as permitted (i.e., at instants $t_0 + k \cdot s_i$, $k = 1, 2, 3, \dots$). They presented the demand bound function that computes the total resource demand of a sporadic task set W under EDF scheduling. This demand bound function $\text{dbf}_{\text{EDF}}^*(W, t)$ calculates the maximum possible resource demands of W for every interval length t as follows:

$$\text{dbf}_{\text{EDF}}^*(W, t) = \sum_{\tau_i \in W} \max \left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1 \right) \times e_i \right) \quad (5.19)$$

In Section 5.5.2, Theorem 5.4 shows a schedulability condition for a periodic task set under EDF scheduling over a periodic resource model. Since the demand bound function $\text{dbf}_{\text{EDF}}(W, t)$ of a periodic task set W computes the resource demand of W under EDF scheduling, we can simply extend this schedulability condition for a sporadic task set by substituting $\text{dbf}_{\text{EDF}}(W, t)$ with $\text{dbf}_{\text{EDF}}^*(W, t)$.

Given the demand bound function of a task model, which computes the maximum possible resource demands of the task model under the EDF or RM scheduling algorithm, our framework can be extended with the task model by plugging the demand bound function into Theorem 5.4 or 5.5.

5.7.2 Supporting Interacting Tasks within a Component

Until now, we have addressed the component abstraction problem under the assumption that the tasks of a component are independent, i.e., they can execute in any order. In many real-time systems, data and control dependencies among tasks may constrain the order in which they can execute. For example, in a radar surveillance system, the signal-processing task is the producer of track records, while the tracker task is the consumer. There are two approaches for supporting data dependencies between the producer and consumer tasks.

One approach is to place the precedence constraint between the producer and consumer tasks so that a consumer job synchronize with the corresponding producer job(s) and wait until the latter completes to execute.

The other approach is not to synchronize producer and consumer tasks. Rather, each producer places the data generated by it in a shared address space to be used by the consumer at any time. In this case, the producer and consumer are independent because they are not explicitly constrained to execute in turn. In this approach, a problem of data integrity can happen between data producer and consumer tasks running at different rates. The data integrity problem exists when the input to a data consumer task changes during the execution of that task. For example, a faster producer task supplies the input to a slower consumer task. The slower consumer reads an input value v_1 from the faster producer and begins computations using that value. The computation are preempted by another execution of the faster producer, which computes a new output value v_2 . A data integrity problem now arises: when the slower consumer resumes execution, it continues its computations, now using the “new” input value v_2 .

Here, we briefly examine an approach, namely, the RTW approach, that can resolve the data integrity problem in real-time data transfer between the producer and consumer tasks with different periods.

RTW is a tool for automatic code generation in the MATLAB/Simulink environment [19]. The tool addresses the data integrity problem by placing *race transition* blocks, *hold* or *delay* blocks, between the

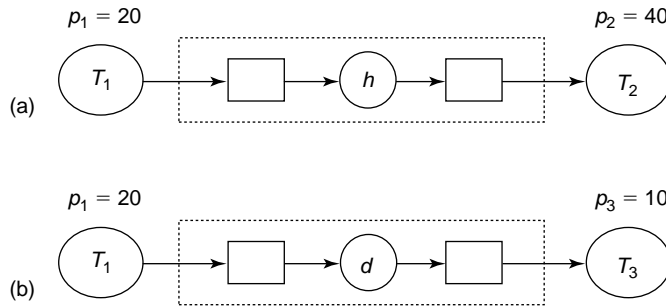


FIGURE 5.9 Data producer and consumer tasks in the RTW model: (a) with a hold block and (b) with a delay block.

data producer and consumer tasks of different periods, under the restriction that the periods of producer and consumer tasks should be harmonic. The tool assumes that a priority-based preemption mechanism will be used for task execution. Figure 5.9 shows an example of adding hold and delay blocks.

A hold block is inserted between a faster producer task T_1 and a slower consumer task T_2 , for guaranteeing that the input value of T_2 from T_1 does not change during the execution of T_2 . The hold block has the same period of T_2 but is a higher-priority task than T_2 , so that it reads the latest output value of T_1 before T_2 executes and holds it until T_2 finishes executing.

A delay block is used for the inverse case, being inserted between a slower producer task T_1 and a faster consumer task T_3 . The delay block has the same period of T_1 but is a higher-priority task than T_1 , for ensuring that no matter when T_1 finishes, the delay block does not overwrite the input value of T_3 during the execution of T_3 .

Resolving the data integrity problem between the producer and consumer tasks using the hold and delay blocks, the RTW approach inherently allows these producer and consumer tasks to treat as independent tasks. That is, even though a component C consists of interacting tasks with data dependency, the component C can be treated as consisting of only independent tasks. Assuming the negligible execution times of the hold and delay blocks, Matic and Henzinger [20] have showed that our proposed framework can be used for abstracting the component C . In addition to the RTW approach, they have also showed that another approach, namely, the LET approach [11], can be used for supporting interacting tasks with data dependencies and that our proposed framework can be used with LET for abstracting components with the interacting tasks.

5.8 Conclusions

In this chapter, we have considered the problem of analyzing the schedulability of hierarchical scheduling systems through component abstraction and composition. We gave schedulability conditions and utilization bounds for a periodic tasks under EDF and RM scheduling over the bounded-delay resource model and the periodic resource model.

In this chapter, we have considered only hard real-time resource models. For future research, it will be interesting to investigate soft real-time resource models for hierarchical scheduling systems. Soft real-time task models such as the (m, k) -firm deadline model [10] and the weakly hard task model [3] can be a good basis for the development of soft real-time resource models.

References

1. L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proc. of the Fourth ACM International Conference on Embedded Software*, September 2004.

2. S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 182–190, December 1990.
3. G. Bernat, A. Burns, and A. Llamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4): 308–321, 2001.
4. R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proc. of IEEE Real-Time Systems Symposium*, December 2005.
5. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, New York, NY, USA, 2001.
6. L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proc. of the First International Workshop on Embedded Software*, pp. 148–165. Lecture Notes in Computer Science 2211, Springer, Berlin, 2001.
7. Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 308–319, December 1997.
8. X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 26–35, December 2002.
9. P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 107–121, 1996.
10. M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k) -firm deadlines. *IEEE Transactions on Computers*, 44(12): 1443–1451, 1995.
11. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of IEEE*, 91: 84–99, 2003.
12. T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pp. 253–263, April 2006.
13. T.-W. Kuo and C. H. Li. A fixed-priority-driven open environment for real-time applications. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 256–267, December 1999.
14. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.
15. G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pp. 166–175, May 2000.
16. G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proc. of Euromicro Conference on Real-Time Systems*, July 2003.
17. G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multi-programmed hard-real-time environments. In *Proc. of IEEE Real-Time Systems Symposium*, December 2000.
18. C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of the ACM*, 20(1): 46–61, 1973.
19. Mathworks. Models with multiple sample rates. In *Real-Time Workshop User Guide*, pp. 1–34. The MathWorks Inc., Natick, MA, USA, 2005.
20. S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 99–110, December 2005.
21. A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pp. 75–84, May 2001.
22. A. K. Mok and X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 129–138, December 2001.
23. J. Regehr and J. Stankovic. HLS: a framework for composing soft real-time schedulers. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 3–14, December 2001.
24. S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proc. of Euromicro Conference on Real-Time Systems*, June 2002.

25. I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of IEEE Real-Time Systems Symposium*, pp. 2–13, December 2003.
26. I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proc. of IEEE Real-Time Systems Symposium*, December 2004.
27. I. Shin and I. Lee. Compositional real-time scheduling framework. Technical report, MS-CIS-04-20, University of Pennsylvania, 2004.
28. I. Shin. Compositional framework for real-time embedded systems. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, USA, 2006.
29. E. Wandeler and L. Thiele. Real-time interface for interface-based design of real-time systems with fixed priority scheduling. In *Proc. of the Fifth ACM International Conference on Embedded Software (EMSOFT '05)*, pp. 80–89, October 2005.
30. E. Wandeler and L. Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pp. 243–252, April 2006.

6

Power-Aware Resource Management Techniques for Low-Power Embedded Systems

Jihong Kim

Seoul National University

Tajana Simunic Rosing

University of California

6.1	Introduction	6-1
6.2	Dynamic Voltage Scaling	6-2
	Intrataask Voltage Scaling • Intertask Voltage Scaling	
6.3	Dynamic Power Management	6-8
	Heuristic Policies • Stochastic Policies • OS and Cross-Layer DPM	
6.4	Conclusions	6-12

6.1 Introduction

Energy consumption has become one of the most important design constraints for modern embedded systems, especially for mobile embedded systems that operate with a limited energy source such as batteries. For these systems, the design process is characterized by a trade-off between a need for high performance and low power consumption emphasizing high performance to meeting the performance constraints while minimizing the power consumption. Decreasing the power consumption not only helps with extending the battery lifetime in portable devices but is also a critical factor in lowering the packaging and the cooling costs of embedded systems. While better low-power circuit design techniques have helped to lower the power consumption [9,34,41], managing power dissipation at higher abstraction levels can considerably decrease energy requirements [3,14].

Since we will be focusing on dynamic power P_{dynamic} in this chapter, the total power dissipation P_{CMOS} can be approximated as $P_{\text{CMOS}} \approx P_{\text{dynamic}}$. The dynamic power of CMOS circuits is dissipated when the output capacitance is charged or discharged, and is given by $P_{\text{dynamic}} = \alpha \cdot C_L \cdot V_{\text{dd}}^2 \cdot f_{\text{clk}}$, where α is the switching activity (the average number of high-to-low transitions per cycle), C_L the load capacitance, V_{dd} the supply voltage, and f_{clk} the clock frequency. The energy consumption during the time interval $[0, T]$ is given by $E = \int_0^T P(t) dt \propto V_{\text{dd}}^2 \cdot f_{\text{clk}} \cdot T = V_{\text{dd}}^2 \cdot N_{\text{cycle}}$, where $P(t)$ is the power dissipation at t and N_{cycle} is the number of clock cycles during the interval $[0, T]$. These equations indicate that a significant energy saving can be achieved by reducing the supply voltage V_{dd} ; a decrease in the supply voltage by a factor of two yields a decrease in the energy consumption by a factor of four.

In this chapter, we focus on the *system-level* power-aware resource management techniques. System-level power management techniques can be roughly classified into two categories: dynamic voltage scaling (DVS) and dynamic power management (DPM). DVS [7], which can be applied in both hardware and software design abstractions, is one of most effective design techniques in minimizing the energy consumption of VLSI systems. Since the energy consumption E of CMOS circuits has a quadratic dependency on the supply voltage, lowering the supply voltage reduces the energy consumption significantly. When a given application does not require the peak performance of a VLSI system, the clock speed (and its corresponding supply voltage) can be dynamically adjusted to the lowest level that still satisfies the performance requirement, saving the energy consumed without perceivable performance degradations.

For example, consider a task with a deadline of 25 ms, running on a processor with the 50 MHz clock speed and 5.0 V supply voltage. If the task requires 5×10^5 cycles for its execution, the processor executes the task in 10 ms and becomes idle for the remaining 15 ms. (We call this type of an idle interval the *slack* time.) However, if the clock speed and the supply voltage are lowered to 20 MHz and 2.0 V, it finishes the task at its deadline ($= 25$ ms), resulting in 84% energy reduction.

Since lowering the supply voltage also decreases the maximum achievable clock speed [43], various DVS algorithms for real-time systems have the goal of reducing supply voltage dynamically to the lowest possible level while satisfying the tasks' timing constraints. For real-time systems where timing constraints must be strictly satisfied, a fundamental energy-delay trade-off makes it more challenging to dynamically adjust the supply voltage so that the energy consumption is minimized while not violating the timing requirements. In this chapter, we focus on DVS algorithms for hard real-time systems.

In contrast, DPM decreases the energy consumption by selectively placing idle components into lower-power states. At the minimum, the device needs to stay in the low-power state for long enough (defined as the break even time) to recuperate the cost of transitioning in and out of the state. The break even time T_{BE} , as defined in Equation 6.1, is a function of the power consumption in the active state, P_{on} ; the amount of power consumed in the low-power state, P_{sleep} ; and the cost of transition in terms of both time, T_{tr} and power, P_{pr} . If it was possible to predict ahead of time the exact length of each idle period, then the ideal power management policy would place a device in the sleep state only when idle period will be longer than the break even time. Unfortunately, in most real systems such perfect prediction of idle period is not possible. As a result, one of the primary tasks DPM algorithms have is to predict when the idle period will be long enough to amortize the cost of transition to a low-power state, and to select the state to transition. Three classes of policies can be defined—timeout-based, predictive, and stochastic. The policies in each class differ in the way prediction of the length of the idle period is made, and the timing of the actual transition into the low-power state (e.g., transitioning immediately at the start of an idle period versus after some amount of idle time has passed).

$$T_{BE} = T_{tr} + T_{tr} \frac{P_{pr} - P_{on}}{P_{on} - P_{sleep}} \quad (6.1)$$

This chapter provides an overview of state-of-the-art DPM and DVS algorithms. The remainder of the chapter is organized as follows. An overview of DVS techniques is presented in Section 6.2. Section 6.3 provides an overview of DPM algorithms and the models used in deriving the policies. We conclude with a summary in Section 6.4.

6.2 Dynamic Voltage Scaling

For hard real-time systems, there are two kinds of voltage scheduling approaches depending on the voltage scaling granularity: intratask DVS (IntraDVS) and intertask DVS (InterDVS). The intratask DVS algorithms adjust the voltage within an individual task boundary, while the intertask DVS algorithms determine the voltage on a task-by-task basis at each scheduling point. The main difference between them is whether the slack times are used for the current task or for the tasks that follow. InterDVS algorithms

distribute the slack times from the current task for the following tasks, while IntraDVS algorithms use the slack times from the current task for the current task itself.

6.2.1 Intratask Voltage Scaling

The main feature of IntraDVS algorithms is how to select the program locations where the voltage and clock will be scaled. Depending on the selection mechanism, we can classify IntraDVS algorithms into five categories: segment-based IntraDVS, path-based IntraDVS, memory-aware IntraDVS, stochastic IntraDVS, and hybrid IntraDVS.

6.2.1.1 Segment-Based IntraDVS

Segment-based IntraDVS techniques partition a task into several segments [29,33]. After executing a segment, they adjust the clock speed and supply voltage exploiting the slack times from the executed segments of a program. In determining the target clock frequency using the slack times available, different slack distribution policies are often used. For example, all the identified slack may be given to the immediately following segment. Or, it is possible to distribute the slack time evenly to all remaining segments. For a better result, the identified slack may be combined with the estimated future slacks (which may come from the following segments' early completions) for a slower execution.

A key problem of the segment-based IntraDVS is how to divide an application into segments. Automatically partitioning an application code is not trivial. For example, consider the problem of determining the granularity of speed changes. Ideally, the more frequently the voltage is changed, the more efficiently the application can exploit dynamic slacks, saving more energy. However, there is energy and time overhead associated with each speed adjustment. Therefore, we should determine how far apart any two voltage scaling points should be. Since the distance between two consecutive voltage scaling points varies depending on the execution path taken at runtime, it is difficult to determine the length of voltage scaling intervals statically.

One solution is to use both the compiler and the operating system to adapt performance and reduce energy consumption of the processor. The collaborative IntraDVS [1] uses such an approach and provides a systematic methodology to partition a program into segments considering branch, loop, and procedure call. The compiler does not insert voltage scaling codes between segments but annotates the application program with the so-called power management hints (PMH) based on program structure and estimated worst-case performance. A PMH conveys path-specific runtime information about a program's progress to the operating system. It is a very low-cost instrumentation that collects path-specific information for the operating system about how the program is behaving relative to the worst-case performance. The operating system periodically invokes a power management point (PMP) to change the processor's performance based on the timing information from the PMH. This collaborative approach has the advantage that the lightweight hints can collect accurate timing information for the operating system without actually changing performance. Further, the periodicity of performance/energy adaptation can be controlled independent of PMH to better balance the high overhead of adaptation.

We can also partition a program based on the workload type. For example, the required decoding time for each frame in an MPEG decoder can be separated into two parts [10]: a frame-dependent (FD) part and a frame-independent (FI) part. The FD part varies greatly according to the type of the incoming frame, whereas the FI part remains constant regardless of the frame type. The computational workload for an incoming frame's FD part (W_{FD}^P) can be predicted by using a frame-based history, that is, maintaining a moving average of the FD time for each frame type. The FI time is not predicted since it is constant for a given video sequence (W_{FI}). Because the total predicted workload is ($W_{FD}^P + W_{FI}$), given a deadline D , the program starts with the clock speed f_{FD} as follows:

$$f_{FD} = \frac{W_{FD}^P + W_{FI}}{D} \quad (6.2)$$

For the MPEG decoder, since the FD part (such as the IDCT and motion compensation steps) is executed before the FI part (such as the frame dithering and frame display steps), the FD time prediction error is

recovered inside that frame itself, that is, during the FI part, so that the decoding time of each frame can be maintained satisfying the given frame rate. This is possible because the workload of the FI part is constant for a given video stream and easily obtained after decoding the first frame. When a misprediction occurs (which can be detected by comparing the predicted FD time (W_{FD}^P) with the actual FD time (W_{FD}^A)), an appropriate action must be taken during the FI part to compensate for the misprediction.

If the actual FD time was smaller than the predicted value, there will be an idle interval before the deadline. Hence, we can scale down the voltage level during the FI part's processing. In contrast, if the actual FD time was larger than the predicted value, a corrective action must be taken to meet the deadline. This is accomplished by scaling up the voltage and frequency during the FI part so as to make up for the lost time. Since the elapsed time consumed during the FD part is W_{FD}^A/f_{FD} , the FI part should start with the following clock speed f_{FI} :

$$f_{FI} = \frac{W_{FI}}{D - (W_{FD}^A/f_{FD})} \quad (6.3)$$

6.2.1.2 Path-Based IntraDVS

At a specific program point, two kinds of slack times can be identified: backward slack and forward slack. While the backward slack is generated from the early completion of the executed program segments, the forward slack is generated when the change of remaining workload is estimated. Though the segment-based IntraDVS utilizes the backward slack times, the path-based IntraDVS exploits the forward slack times based on the program's control flow.

Consider a hard real-time program P with the deadline of 2 s shown in Figure 6.1a. The control flow graph (CFG) G_P of the program P is shown in Figure 6.1b. In G_P , each node represents a basic block of P and each edge indicates the control dependency between basic blocks. The number within each node indicates the number of execution cycles of the corresponding basic block. The back edge from b_5 to b_{wh} models the **while** loop of the program P . The worst-case execution cycles of this program are 2×10^8 cycles.

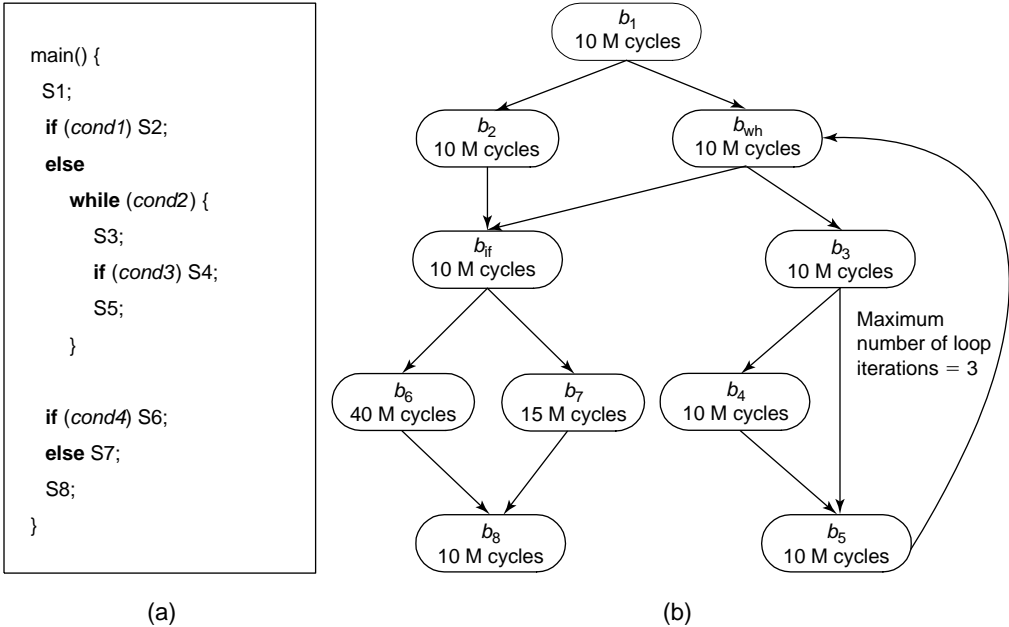


FIGURE 6.1 An example program P : (a) an example real-time program with the 2 s deadline and (b) its CFG representation G_P .

Path-based IntraDVS [44] adjusts the clock speed within the task depending on the control flow. For example, when the program control flow follows the execution path $\pi_1 = (b_1, b_2, b_{if}, b_6, b_8)$ of Figure 6.1b, the clock speed is initially determined to complete the worst-case execution path (WCEP) before the deadline, that is, 100 MHz. However, we can reduce the clock speed at the edge (b_1, b_2) because we know this control flow does not follow the WCEP. In the path-based IntraDVS algorithm, we identify appropriate program locations where the clock speed should be adjusted, and inserts clock and voltage scaling codes to the selected program locations at compile time. The branching edges of the CFG, that is, branch or loop statements, are the candidate locations for inserting voltage scaling codes because the remaining execution cycles are changed at those locations.

The path-based IntraDVS consists of two key steps: (1) to predict the execution path of application program at compile time and (2) to adjust the clock speed depending on the real execution path taken at runtime. In the first step, using the predicted execution path, we calculate the remaining predicted execution cycles (RPEC) $\delta(b_i)$ at a basic block b_i , which is a branching node in G_P as follows:

$$\delta(b_i) = c(b_i) + \mathcal{P}(\delta(b_j), \delta(b_k)) \quad (6.4)$$

where $c(b_i)$ is the execution cycles for the basic block b_i and \mathcal{P} is the prediction function. The basic blocks b_j and b_k are the immediate successor nodes of b_i in the CFG. Depending on the prediction method for execution path, the function \mathcal{P} is determined. For example, if we take the WCEP as an execution path to be taken at runtime, $\mathcal{P}(\alpha, \beta)$ will be equal to $\max(\alpha, \beta)$. With the predicted value of $\delta(b_i)$, we set the initial clock frequency and its corresponding voltage assuming that the task execution will follow the predicted execution path. We call the predicted execution path as the *reference path* because the clock speed is determined based on the execution path.

For a loop, we use the following equation to predict the remaining execution cycles for the loop L :

$$\delta(L) = c(H_L) + (c(H_L) + c(B_L)) \cdot N_{\text{pred}}(L) + \delta(\text{post}_L) \quad (6.5)$$

where $c(H_L)$ and $c(B_L)$ mean the execution cycles of the header and the body of the loop L , respectively. $N_{\text{pred}}(L)$ is the predicted number of loop iterations and post_L denotes the successor node of the loop, which is executed just after the loop termination.

At runtime, if the actual execution deviates from the (predicted) reference path (say, by a branch instruction), the clock speed can be adjusted depending on the difference between the remaining execution cycles of the reference path and that of the newly deviated execution path. If the new execution path takes significantly longer to complete its execution than the reference execution path, the clock speed should be *raised* to meet the deadline constraint. In contrast, if the new execution path can finish its execution earlier than the reference execution path, the clock speed can be *lowered* to save energy.

For runtime clock speed adjustment, voltage scaling codes are inserted into the selected program locations at compile time. The branching edges of the CFG, that is, branch or loop statements, are the candidate locations for inserting voltage scaling codes. They are called *voltage scaling points* (VSPs) because the clock speed and voltage are adjusted at these points. There are two types of VSPs, the B-type VSP and L-type VSP. The B-type VSP corresponds to a branch statement while the L-type VSP maps into a loop statement. VSPs can also be categorized into up-VSPs or down-VSPs, where the clock speed is raised or lowered, respectively. At each VSP (b_i, b_j) , the clock speed is determined using $\delta(b_i)$ and $\delta(b_j)$ as follows:

$$f(b_j) = \frac{\delta(b_j)}{T_j} = f(b_i) \cdot \frac{\delta(b_j)}{\delta(b_i) - c(b_i)} = f(b_i) \cdot r(b_i, b_j) \quad (6.6)$$

where $f(b_i)$ and $f(b_j)$ are the clock speeds at the basic blocks b_i and b_j , respectively. T_j is the remaining time until the deadline from the basic block b_j and $r(b_i, b_j)$ the *speed update ratio* of the edge (b_i, b_j) .

For a loop, if the actual number of loop iterations is N_{actual} , the clock speed is changed after the loop as follows:

$$f(\text{post}_L) = f(\text{pre}_L) \cdot \frac{\max((c(H_L) + c(B_L)) \cdot (N_{\text{actual}}(L) - N_{\text{pred}}(L)), 0) + \delta(\text{post}_L)}{\max((c(H_L) + c(B_L)) \cdot (N_{\text{pred}}(L) - N_{\text{actual}}(L)), 0) + \delta(\text{post}_L)} \quad (6.7)$$

where $f(\text{pre}_L)$ is the clock speed before executing the loop L . If N_{actual} is larger (smaller) than N_{pred} , the clock speed is increased (decreased).

Although the WCEP-based IntraDVS reduces the energy consumption significantly while guaranteeing the deadline, this is a pessimistic approach because it always predicts that the longest path will be executed. If we use the average-case execution path (ACEP) as a reference path, a more efficient voltage schedule can be generated. (The ACEP is an execution path that will be most likely to be executed.) To find the ACEP, we should utilize the profile information on the program execution.

6.2.1.3 Other IntraDVS Techniques

Memory-aware IntraDVS utilizes the CPU idle times owing to external memory stalls. While the *compiler-driven* IntraDVS [19] identifies the program regions where the CPU is mostly idle due to memory stalls at compile time, the *event-driven* IntraDVS [11,51] uses several performance monitoring events to capture the CPU idle time at runtime. The memory-aware IntraDVS differs from the path-based IntraDVS in the type of CPU slacks being exploited. While the path-based IntraDVS takes advantage of the difference between the predicted execution path and the real execution path of applications, the memory-aware IntraDVS exploits slacks from the memory stalls. The idea is to identify the program regions in which the CPU is mostly idle due to memory stalls and slow them down for energy reduction. If the system architecture supports the overlapped execution of the CPU and memory operations, such a CPU slow-down will not result in a serious system performance degradation, hiding the slow CPU speed behind the memory hierarchy accesses, which are on the critical path. There are two kinds of approaches to identify the memory-bound regions: analyzing a program at compile time and monitoring runtime hardware events.

The compiler-directed IntraDVS [19] partitions a program into multiple program regions. It assigns different slow-down factors to different selected regions so as to maximize the overall energy savings without violating the global performance penalty constraint. The application program is partitioned not to introduce too much overhead due to switches between different voltages/frequencies. That is, the granularity of the region needs to be large enough to compensate for the overhead of voltage and frequency adjustments.

Event-driven IntraDVS [11,51] makes use of runtime information about the external memory access statistics to perform CPU voltage and frequency scaling with the goal of minimizing the energy consumption while controlling the performance penalty. The technique relies on dynamically constructed regression models that allow the CPU to calculate the expected workload and slack time for the next time slot. This is achieved by estimating and exploiting the ratio of the total off-chip access time to the total on-chip computation time. To capture the CPU idle time, several performance monitoring events (such as ones collected by the performance monitoring unit (PMU) of the XScale processor) can be used. Using the performance monitoring events, we can count the number of instructions being executed and the number of external memory accesses at runtime.

Stochastic IntraDVS uses the stochastic information on the program's execution time [17,31]. This technique is motivated by the idea that, from the energy consumption perspective, it is usually better to "start at low speed and accelerate execution later when needed" than to "start at high speed and reduce the speed later when the slack time is found" in the program execution. It finds a speed schedule that minimizes the expected energy consumption while still meeting the deadline. A task starts executing at a low speed and then gradually accelerates its speed to meet the deadline. Since an execution of a task might not follow the WCEP, it can happen that high-speed regions are avoided.

Hybrid IntraDVS overcomes the main limitation of IntraDVS techniques described before, which is that they have no global view of the task set in multitask environments. Based on an observation that a

cooperation between IntraDVS and InterDVS could result in more energy-efficient systems, the *hybrid IntraDVS* technique selects either the *intramode* or the *intermode* when slack times are available during the execution of a task [45]. At the intermode, the slack time identified during the execution of a task is transferred to the following other tasks. Therefore, the speed of the current task is not changed by the slack time produced by itself. At the intramode, the slack time is used for the current task, reducing its own execution speed.

6.2.2 Intertask Voltage Scaling

InterDVS algorithms exploit the “run-calculate-assign-run” strategy to determine the supply voltage, which can be summarized as follows: (1) run a current task; (2) when the task is completed, calculate the maximum allowable execution time for the next task; (3) assign the supply voltage for the next task; and (4) run the next task. Most InterDVS algorithms differ during step (2) in computing the maximum allowed time for the next task τ , which is the sum of the worst-case execution time (WCET) of τ and the slack time available for τ .

A generic InterDVS algorithm consists of two parts: slack estimation and slack distribution. The goal of the slack estimation part is to identify as much slack times as possible, while the goal of the slack distribution part is to distribute the resulting slack times so that the resulting speed schedule is as uniform as possible. Slack times generally come from two sources; static slack times are the extra times available for the next task that can be identified statically, while dynamic slack times are caused from runtime variations of the task executions.

6.2.2.1 Slack Estimation and Distribution Methods

One of the most commonly used *static* slack estimation methods is to compute the maximum constant speed, which is defined as the lowest possible clock speed that guarantees the feasible schedule of a task set [46]. For example, in EDF scheduling, if the worst-case processor utilization (WCPU) U of a given task set is lower than 1.0 under the maximum speed f_{\max} , the task set can be scheduled with a new maximum speed $f'_{\max} = U \cdot f_{\max}$. Although more complicated, the maximum constant speed can be statically calculated as well for rate-monotonic (RM) scheduling [17,46].

Three widely used techniques of estimating *dynamic* slack times are briefly described below. *Stretching-to-NTA* is based on a slack between the deadline of the current task and the arrival time of the next task. Even though a given task set is scheduled with the maximum constant speed, since the actual execution times of tasks are usually much less than their WCETs, the tasks usually have dynamic slack times. One simple method to estimate the dynamic slack time is to use the arrival time of the next task [46]. (The arrival time of the next task is denoted by NTA.) Assume that the current task τ is scheduled at time t . If NTA of τ is later than $(t + \text{WCET}(\tau))$, task τ can be executed at a lower speed so that its execution completes exactly at the NTA. When a single task τ is activated, the execution of τ can be stretched to NTA. When multiple tasks are activated, there can be several alternatives in stretching options. For example, the dynamic slack time may be given to a single task or distributed equally to all activated tasks.

Priority-based slack stealing exploits the basic properties of priority-driven scheduling such as RM and earliest deadline first (EDF). The basic idea is that when a higher-priority task completes its execution earlier than its WCET, the following lower-priority tasks can use the slack time from the completed higher-priority task. It is also possible for a higher-priority task to utilize the slack times from completed lower-priority tasks. However, the latter type of slack stealing is computationally expensive to implement precisely. Therefore, the existing algorithms are based on heuristics [2,27].

Utilization updating is based on an observation that the actual processor utilization during runtime is usually lower than the WCPU. The utilization updating technique estimates the required processor performance at the current scheduling point by recalculating the expected WCPU using the actual execution times of completed task instances [38]. When the processor utilization is updated, the clock speed can be adjusted accordingly. The main merit of this method is its simple implementation, since only the processor utilization of completed task instances have to be updated at each scheduling point.

In distributing slack times, most InterDVS algorithms have adopted a greedy approach where all the slack times are given to the next activated task. This approach is not an optimal solution, but the greedy approach is widely used because of its simplicity.

6.2.2.2 Example InterDVS Algorithms

In this section, we briefly summarize some of the representative DVS algorithms proposed for hard real-time systems. Here, eight InterDVS algorithms are chosen, two [38,46] of which are based on the RM scheduling policy, while the other six algorithms [2,27,38,46] are based on the EDF scheduling policy.

In these selected DVS algorithms, one or sometimes more than one slack estimation methods explained in the previous section were used. In lppsEDF and lppsRM, which were proposed by Shin et al. [46], slack time of a task is estimated using the maximum constant speed and stretching-to-NTA methods.

The ccRM algorithm proposed by Pillai and Shin [38] is similar to lppsRM in the sense that it uses both the maximum constant speed and the stretching-to-NTA methods. However, while lppsRM can adjust the voltage and clock speed only when a single task is active, ccRM extends the stretching-to-NTA method to the case where multiple tasks are active.

Pillai and Shin [38] also proposed two other DVS algorithms, ccEDF and laEDF, for EDF scheduling policy. These algorithms estimate slack time of a task using the utilization updating method. While ccEDF adjusts the voltage and clock speed based on runtime variation in processor utilization alone, laEDF takes a more aggressive approach by estimating the amount of work required to be completed before NTA.

DRA and AGR, which were proposed by Aydin et al. [2], are two representative DVS algorithms that are based on the priority-based slack stealing method. The DRA algorithm estimates the slack time of a task using the priority-based slack stealing method along with the maximum constant speed and the stretching-to-NTA methods. Aydin et al. also extended the DRA algorithm and proposed another DVS algorithm called AGR for more aggressive slack estimation and voltage/clock scaling. In AGR, in addition to the priority-based slack stealing, more slack times are identified by computing the amount of work required to be completed before NTA.

lpSHE is another DVS algorithm, which is based on the priority-based slack stealing method [27]. Unlike DRA and AGR, lpSHE extends the priority-based slack stealing method by adding a procedure that estimates the slack time from lower-priority tasks that were completed earlier than expected. DRA, AGR, and lpSHE algorithms are somewhat similar to one another in the sense that all of them use the maximum constant speed in the offline phase and the stretching-to-NTA method in the online phase in addition to the priority-based slack stealing method.

6.3 Dynamic Power Management

DPM techniques selectively place system components into low-power states when they are idle. A power managed system can be modeled as a *power state machine*, where each state is characterized by the power consumption and the performance. In addition, state transitions have power and delay cost. Usually, lower power consumption also implies lower performance and longer transition delay. When a component is placed into a low-power state, such as a sleep state, it is unavailable for the time period spent there, in addition to the transition time between the states. The transitions between states are controlled by commands issued by a *power manager* (PM) that observes the workload of the system and decides when and how to force power state transitions. The PM makes state transition decisions according to the *power management policy*. The choice of the policy that minimizes power under performance constraints (or maximizes performance under power constraint) is a constrained optimization problem.

The system model for power management therefore consists of three components: the user, the device, and the queue as shown in Figure 6.2. The user, or the application that accesses each device by sending requests, can be modeled with a request interarrival time distribution. When one or more requests arrive, the user is said to be in the active state, otherwise it is in the idle state. Figure 6.2 shows three different power states for the device: active, idle, and sleep. Often the device will have multiple active states, which

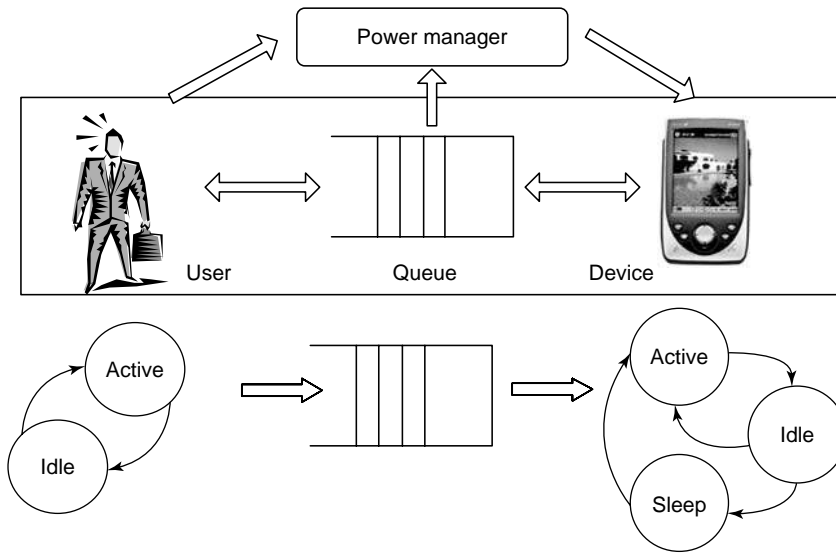


FIGURE 6.2 System model.

can be differentiated by the frequency and voltage of operation. Similarly, there can be multiple inactive states—both idle, each potentially corresponding to a specific frequency of operation in the active state (e.g., XScale processor [23]) and sleep states. Service time distribution describes the behavior of the device in the active state. When the device is in either the idle or the sleep state, it does not service any requests. Typically, the transition to the active state is shorter from the idle state, but the sleep state has lower power consumption. The transition distribution models the time taken by the device to transition between its power states. The queue models a buffer associated with each device. The combination of interarrival time distribution (incoming requests to the buffer) and service time distribution (requests leaving the buffer) can be used to fully characterize the behavior of the queue.

PM's job can consist of a number of tasks: (1) tracking and modeling of incoming service requests with a goal of predicting when longer idle periods occur; (2) traffic shaping—buffering the traffic so larger idle periods are created that enable the device to stay asleep for a longer time period; (3) deciding when and to what sleep state a device should transition to; and (4) making a decision as to when a device should wake up to process requests. In this section, we provide an overview of DPM policies that address various combinations of these tasks.

6.3.1 Heuristic Policies

Most commercial power management implementations focus only on deciding when a device should go to sleep. The cost of transition between the active and the sleep state typically determines how aggressive a policy can be. When the cost is reasonably low, or is at least not perceivable by the users, then policies transition device to sleep as soon as it becomes idle. For example, μ Sleep has been introduced in Ref. 6 as a way of reducing the idle power consumption of HP's IPAQ platform. IBM's wristwatch enters the standby state as soon as it is idle [25]. Intel's QuickStart technology [22] puts a processor to sleep between the keystrokes. In all the three cases the cost of waking up is not perceivable by the user. When the cost of transition to a low-power state is significant, then commercial implementations typically implement policies based on timeout. For example, HP's IPAQ uses a timeout value to decide when display should be dimmed. The timeout value is usually either fixed at design time or can be set by a user.

Timeout policies assume that if the incoming workload has an idle period that is longer than some timeout value T_{to} , then there is a very high likelihood that the idle period will be long enough to justify going to a sleep state. Therefore, the total length of the idle time needs to be longer than $T_{to} + T_{be}$.

The timeout policies waste energy during the timeout period, as they keep a device in the active state until the timeout expires. Therefore, shorter timeout values are better for saving energy while waiting to transition to sleep. In contrast, if the timeout value is too long, then there is a good chance that the rest of the idle period is not long enough to amortize the cost of transition to sleep, so the overall cost is then actually higher than it would have been if the device had not gone to sleep at all. A good example of this situation is setting too short of a timeout on a hard drive. Typical hard drive can take up to a few seconds to spin up from sleep. The spinning up process can cost more than twice the power consumption of the active state. Thus, if the timeout value is only a few hundred milliseconds, chances are that a number of times just as the hard drive spins down, it will have to immediately spin back up, thus causing large performance and energy overhead. Therefore, the selection of the timeout value has to be done with a good understanding of both device characteristics and the typical workloads. A study of hard drive traces is presented in Ref. 30. One of the major conclusions is that timeout values on the order of multiple seconds are appropriate for many hard drives. This timeout value for hard disks can be shown to be within a factor of two of the optimal policy using competitive ratio analysis [26].

Although most commercial implementations have a single fixed timeout value, a number of studies have designed methods for adapting the timeout value to changes in the workloads (e.g., Refs. 13, 18, 24, 28). Machine learning techniques [18], along with models based on economic analysis [24,28] have been employed as ways to adapt the timeout value. In Ref. 42, adaptive policy learns the distribution of idle periods and based on that it selects which sleep state is most appropriate. Competitive analysis is then used to show how close the adaptive policy is to optimum. Adaptive timeout policies also suffer from wasting energy while waiting for the timeout to expire, but hopefully the amount of energy wasted is lower as the timeout value is more closely fine tuned to the changes in the workload.

Predictive policies attempt to not only predict the length of the next idle period by studying the distribution of request arrivals, but also try to do so with enough accuracy to be able to transition a device into a sleep state with no idleness. In addition, some predictive policies also wake up a device from the sleep state in anticipation of service request arrival. When prediction of timing and the length of the idle period is correct, then predictive policies provide a solution with no overhead. In contrast, if the prediction is wrong, then potential cost can be quite large. If a PM transitions a device to sleep expecting a long idle period, but the idle period is actually short, the overall cost in terms of both energy and performance can be quite large. However, if the manager predicts an idle period that is short, then it will wake up a device before it is needed and thus waste energy the same way standard timeout policies do. The quality of idle period prediction is the keystone of these policies. A study on prediction of idleness in hard drives is presented in Ref. 16. Policies based on study of idle period prediction are presented in both Refs. 16 and 50. Two policies are introduced in Ref. 50: the first one is based on a regression model of idle periods, while the second one is based on the analysis of a length of a previously busy period. Exponential averaging is used to predict the idle period length in Ref. 20. Analysis of user interface interactions is used to guide a decision on when to wake up a given component [54]. In multicore designs, a signal has been proposed that can be used to notify the system components of impending request arrival and of instances when no more requests are expected [49]. These signals enable both predictive wake up and predictive sleep. Both timeout and predictive policies have one thing in common—they are heuristic. None of these policies can guarantee optimality. In contrast, policies that use stochastic models to formulate and solve the power management problem can provide optimal solutions within restrictive assumption made when developing the system model.

6.3.2 Stochastic Policies

Stochastic policies can loosely be categorized into time and event driven, and based on assumption that all distributions modeling the system are memoryless (e.g., geometric and exponential distribution) or that some distributions are history dependent. Power management policies can be classified into two categories by the manner in which decisions are made: *discrete time* (or clock based) and *event driven*. In addition, policies can be *stationary* (the same policy applies at any point in time) or *nonstationary* (the policy changes

over time). In both discrete and event-driven approaches, optimality of the algorithm can be guaranteed since the underlying theoretical model is based on Markov chains. An overview of stochastic DPM policies follows.

Benini et al. [4] formulated a probabilistic system model using stationary discrete-time Markov decision processes (DTMDP). They rigorously formulate the policy optimization problem and showed that it can be solved exactly and in polynomial time in the size of the system model. The DTMDP approach requires that all state transitions follow stationary geometric distributions, which is not true in many practical cases. Nonstationary user request rates can be treated using an adaptive policy interpolation procedure presented in Ref. 12. A limitation of both stationary and adaptive DTMDP policies is that decision evaluation is repeated periodically, even when the system is idle, thus wasting power. For example, for a 10 W processor, the DTMDP policy with evaluation period of 1 s would waste as much as 1800 J of energy from the battery during a 30 min break. The advantage of the discrete time approach is that decisions are reevaluated periodically so the decision can be reconsidered thus adapting better to arrivals that are not truly geometrically distributed.

An alternative to the DTMDP model is a continuous-time Markov decision process (CTMDP) model [40]. In a CTMDP, the PM issues commands upon event occurrences instead of at discrete time settings. As a result, more energy can be saved since there is no need to continually reevaluate the policy in the low-power state. Results are guaranteed optimal assuming that the exponential distribution describes well the system behavior. Unfortunately, in many practical cases, the transition times may be distributed according to a more general distribution. Figure 6.3 shows a tail distribution of wireless LAN (WLAN) service request interarrival times. The plot highlights that for longer idle times of interest to power management, the exponential distribution shows a very poor fit, while a heavy tailed distribution, such as Pareto, is a much better fit to the data. As a result, in real implementation the results can be far from optimal [48]. Work presented in Ref. 40 uses series and parallel combinations of exponential distributions to approximate general distribution of transition times. Unfortunately, this approach is very complex and does not give a good approximation for the bursty behavior observed in real systems [37,48].

Time-indexed semi-Markov decision process (TISMDP) model has been introduced to solve the DPM optimization problem without the restrictive assumption of memoryless distribution for system transitions [48]. The power management policy optimization is solved *exactly* and in polynomial time with guaranteed optimal results. Large savings are measured with TISMDP model as it handles general user request interarrival distributions and makes decisions in the event-driven manner. The resulting DPM policy is event driven and can be implemented as a randomized timeout. The value of randomization depends on the parameters of the policy derived from TISMDP optimization. The policy itself is in the form of a distribution that provides the probability of transitioning into a sleep state for every timeout value. Renewal theory has been used in Ref. 49 for joint optimization of both DPM and DVS on multicore systems-on-a-chip. As in TISMDP, the renewal model allows for modeling a general transition distribution.

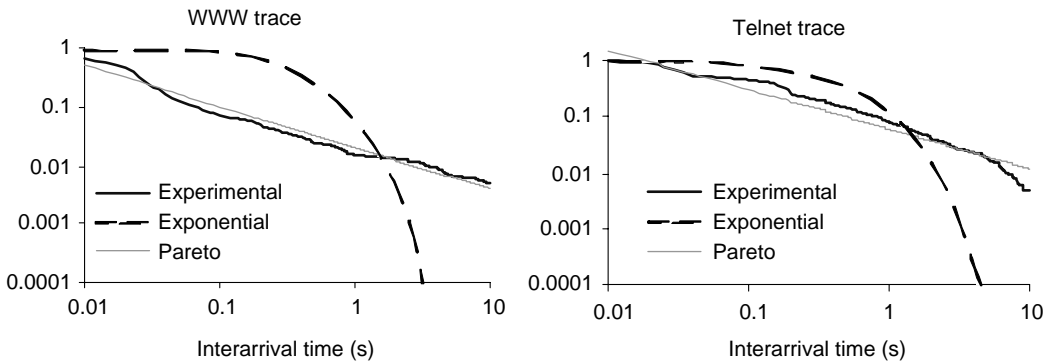


FIGURE 6.3 WLAN idle state arrival tail distribution.

Probabilistic model checking has been employed to verify the correctness of stochastic policies [47]. Although both TISMDP and renewal models limit the restriction of only memoryless distributions, they do require that all transitions be statistically independent and stationary. An approach presented in Ref. 24 removes the stationary assumption by learning the interarrival distribution online. Although this approach does not guarantee optimality, the quality of estimation has been evaluated using competitive analysis.

6.3.3 OS and Cross-Layer DPM

Most power management policies are implemented as a part of an operating system. Intel, Microsoft, and Toshiba created an Advanced Configuration and Power Interface (ACPI) specification that can be used to implement power management policies in Windows OS [21]. ACPI gives only a structure for implementing DPM but does not actually provide any policies beyond simple timeouts for specific devices. In contrast, Nemesis OS [35] and ECOSystem [53] both use pricing mechanisms to efficiently allocate a system's hardware resources based on energy consumption. Both of these approaches use high-level approximations for energy cost of access to any given hardware resource, thus incurring significant inaccuracies. Their implementation for a different system requires that both kernel and the hardware-specific functions be changed. Another OS-level approach is to compute an equivalent utilization for each hardware component and then to transition a device to sleep when its utilization falls below a predefined threshold [32]. A few energy-aware software implementations integrate information available at multiple system layers to be able to manage energy more efficiently [15,52].

All DPM policies discussed thus far do not perform any traffic reshaping of the incoming workload. Thus, their primary goal is to evaluate for a given idle period if a device should transition to sleep. Various buffering mechanisms have been suggested by a number of researchers as a way to enhance the availability of longer idle periods suitable for power management. Buffering for streaming multimedia applications has been proposed as a methodology to lower the energy consumption in wireless network interface cards [5]. Similar techniques have been used to help increase the idle times available for spinning down the hard disk power [36]. A general buffer management methodology based on the idea of inventory control has been proposed in Ref. 8. An advantage of this approach is that it can be applied to multiple devices in a single system. When combined with OS-level scheduling, the adaptive workload buffering can be used for both power management and voltage scaling. Shorter idle times created by adaptive buffering can be used to slow down a device, while the longer ones are more appropriate for transitions to sleep.

6.4 Conclusions

We have described two representative system-level power-aware resource management approaches for low-power embedded systems. DVS techniques take advantage of workload variations within a single-task execution as well as workload fluctuations from running multiple tasks, and adjust the supply voltage, reducing the energy consumption of embedded systems. DPM techniques identify idle system components using various heuristics and place the identified components into low-power states. We reviewed the key steps of these techniques.

In this chapter, we reviewed DVS and DPM as an independent approach. Since both approaches are based on the system idleness, DVS and DPM can be combined into a single-power management framework. For example, shorter idle periods are more amiable to DVS, while longer ones are more appropriate for DPM. Thus, a combination of the two approaches is necessary for more power-efficient embedded systems.

References

1. N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven, Energy management for real-time embedded applications with compiler support, in *Proc. of Conference on Language, Compiler, and Tool Support for Embedded Systems*, pp. 238–246, 2002.

2. H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez, Dynamic and aggressive scheduling techniques for power-aware real-time systems, in *Proc. of IEEE Real-Time Systems Symposium*, December 2001.
3. L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer, Norwell, MA, USA, 1997.
4. L. Benini, G. Paleologo, A. Bogliolo, and G. De Micheli, Policy optimization for dynamic power management, *IEEE Transactions on Computer-Aided Design*, 18(6), pp. 813–833, June 1999.
5. D. Bertozzi, L. Benini, and B. Ricco, Power aware network interface management for streaming multimedia, in *Proc. of IEEE Wireless Communication Network Conference*, pp. 926–930, March 2002.
6. L. S. Brakmo, D. A. Wallach, and M. A. Viredaz, Sleep: a technique for reducing energy consumption in handheld devices, in *Proc. of USENIX/ACM International Conference on Mobile Systems, Applications, and Services*, pp. 12–22, June 2004.
7. T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, A dynamic voltage scaled micro-processor system, *IEEE Journal of Solid State Circuits*, 35(11), pp. 1571–1580, 2000.
8. L. Cai and Y.-H. Lu, Energy management using buffer memory for streaming data, *IEEE Transactions on Computer-Aided Design of IC and Systems*, 24(2), pp. 141–152, February 2005.
9. A. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design*, Kluwer, Norwell, MA, USA, 1995.
10. K. Choi, W.-C. Cheng, and M. Pedram, Frame-based dynamic voltage and frequency scaling for an MPEG player, *Journal of Low Power Electronics*, 1(1), pp. 27–43, 2005.
11. K. Choi, R. Soma, and M. Pedram, Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times, *IEEE Transactions on Computer Aided Design*, 24(1), pp. 18–28, 2005.
12. E. Chung, L. Benini, and G. De Micheli, Dynamic power management for non-stationary service requests, in *Proc. of Design, Automation and Test in Europe*, pp. 77–81, 1999.
13. F. Douglass, P. Krishnan, and B. N. Bershad, Adaptive disk spin-down policies for mobile computers, in *Proc. of 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pp. 121–137, April 1995.
14. C. Ellis, The case for higher-level power management, in *Proc. of 7th IEEE Workshop on Hot Topics in Operating Systems*, pp. 162–167, 1999.
15. J. Flinn and M. Satyanarayanan, Managing battery lifetime with energy-aware adaptation, *ACM Transactions on Computer Systems*, 22(2), pp. 137–179, May 2004.
16. R. Golding, P. Bosch, and J. Wilkes, Idleness is not sloth, in *Proc. of USENIX Winter Technical Conference*, pp. 201–212, January 1995.
17. F. Gruian, Hard real-time scheduling using stochastic data and DVS processors, in *Proc. of International Symposium on Low Power Electronics and Design*, pp. 46–51, 2001.
18. D. P. Helmbold, D. D. E. Long, and B. Sherrod, A dynamic disk spin-down technique for mobile computing, in *Proc. of ACM Annual International Conference on Mobile Computing and Networking*, pp. 130–142, November 1996.
19. C.-H. Hsu and U. Kremer, The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction, in *Proc. of ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, pp. 38–48, 2003.
20. C.-H. Hwang and A. Wu, A predictive system shutdown method for energy saving of event-driven computation, in *Proc. of International Conference on Computer-Aided Design*, pp. 28–32, 1997.
21. Intel, Microsoft and Toshiba, *Advanced Configuration and Power Interface Specification*, Intel, Microsoft, Toshiba, 1996.
22. Intel QuickStart Technology, www.intel.com.
23. Intel XScale Technology, <http://developer.intel.com/design/intelxscale>, 2001.
24. S. Irani, S. Shukla, and R. Gupta, Online strategies for dynamic power management in systems with multiple power-saving states, *ACM Transactions on Embedded Computing Systems*, 2(3), pp. 325–346, July 2003.

25. N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami, Energy tradeoffs in the IBM wristwatch computer, in *Proc. of IEEE International Symposium on Wearable Computers*, pp. 133–140, 2001.
26. A. Karlin, M. Manesse, L. McGeoch, and S. Owicki, Competitive randomized algorithms for nonuniform problems, *Algorithmica*, 11(6), pp. 542–571, 1994.
27. W. Kim, J. Kim, and S. L. Min, A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis, in *Proc. of Design, Automation and Test in Europe (DATE'02)*, pp. 788–794, March 2002.
28. P. Krishnan, P. Long, and J. Vitter, Adaptive disk spindown via optimal rent-to-buy in probabilistic environments, *Algorithmica*, 23(1), pp. 31–56, January 1999.
29. S. Lee and T. Sakurai, Run-time voltage hopping for low-power real-time systems, in *Proc. of Design Automation Conference*, pp. 806–809, 2000.
30. K. Li, R. Kumpf, P. Horton, and T. E. Anderson, A quantitative analysis of disk drive power management in portable computers, in *Proc. of USENIX Winter Technical Conference*, pp. 279–291, January 1994.
31. J. R. Lorch and A. J. Smith, Improving dynamic voltage scaling algorithms with PACE, in *Proc. of ACM SIGMETRICS Conference*, pp. 50–61, 2001.
32. Y.-H. Lu, L. Benini, and G. De Micheli, Power-aware operating systems for interactive systems, *IEEE Transactions on VLSI Systems*, 10(2), pp. 119–134, April 2002.
33. D. Mosse, H. Aydin, B. Childers, and R. Melhem, Compiler-assisted dynamic power-aware scheduling for real-time applications, in *Proc. of Workshop on Compiler and OS for Low Power*, 2000.
34. W. Nabel and J. Mermet (Editors), *Low Power Design in Deep Submicron Electronics*, Kluwer, Norwell, MA, USA, 1997.
35. R. Neugebauer and D. McAuley, Energy is just another resource: energy accounting and energy pricing in the Nemesis OS, in *Proc. of Workshop on Hot Topics in Operating Systems*, May 2001.
36. A. E. Papathanasiou and M. L. Scott, Energy efficient prefetching and caching, in *Proc. of USENIX Annual Technical Conference*, pp. 255–268, 2004.
37. V. Paxson and S. Floyd, Wide area traffic: the failure of poisson modeling, *IEEE Transactions on Networking*, 3(3), pp. 226–244, June 1995.
38. P. Pillai and K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pp. 89–102, October 2001.
39. Q. Qiu and M. Pedram, Dynamic power management based on continuous-time Markov decision processes, in *Proc. of Design Automation Conference*, pp. 555–561, 1999.
40. Q. Qiu and M. Pedram, Dynamic power management of complex systems using generalized stochastic Petri nets, in *Proc. of Design Automation Conference*, pp. 352–356, 2000.
41. J. Rabaey and M. Pedram (Editors), *Low Power Design Methodologies*, Kluwer, Norwell, MA, USA, 1996.
42. D. Ramanathan, S. Irani, and R. Gupta, Latency effects of system level power management algorithms, in *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pp. 350–356, November 2000.
43. T. Sakurai and A. Newton, Alpha-power law MOSFET model and its application to CMOS inverter delay and other formulas, *IEEE Journal of Solid State Circuits*, 25(2), pp. 584–594, 1990.
44. D. Shin, J. Kim, and S. Lee, Intra-task voltage scheduling for low-energy hard real-time applications, *IEEE Design and Test of Computers*, 18(2), pp. 20–30, 2001.
45. D. Shin, W. Kim, J. Jeon, and J. Kim, SimDVS: an integrated simulation environment for performance evaluation of dynamic voltage scaling algorithms, *Lecture Notes in Computer Science*, 2325, pp. 141–156, 2003.
46. Y. Shin, K. Choi, and T. Sakurai, Power optimization of real-time embedded systems on variable speed processors, in *Proc. of the International Conference on Computer-Aided Design*, pp. 365–368, November 2000.

47. S. Shukla and R. Gupta, A model checking approach to evaluating system level power management for embedded systems, in *Proc. of HLDVT*, 2001.
48. T. Simunic, L. Benini, P. Glynn, and G. De Micheli, Event-driven power management, *IEEE Transactions on CAD*, pp. 840–857, July 2001.
49. T. Simunic, S. Boyd, and P. Glynn, Managing power consumption in networks on chips, *IEEE Transactions on VLSI*, pp. 96–107, January 2004.
50. M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, Predictive system shutdown and other architectural techniques for energy efficient programmable computation, *IEEE Transactions on VLSI Systems*, 4(1), pp. 42–55, January 1996.
51. A. Weissel and F. Bellosa, Process cruise control: event-driven clock scaling for dynamic power management, in *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 238–246, 2002.
52. W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets, GRACE: cross-layer adaptation for multimedia quality and battery energy, *IEEE Transactions on Mobile Computing*, 2005.
53. H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, ECOSystem: managing energy as a first class operating system resource, in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 123–132, October 2002.
54. L. Zhong and N. K. Jha, Dynamic power optimization for interactive systems, in *Proc. of International Conference on VLSI Design*, pp. 1041–1047, January 2004.

Imprecise Computation Model: Total Weighted Error and Maximum Weighted Error

Joseph Y-T. Leung

New Jersey Institute of Technology

7.1	Introduction	7-1
7.2	Total Weighted Error	7-3
	Single Processor • Parallel and Identical Processors • Uniform Processors	
7.3	Maximum Weighted Error	7-9
7.4	Concluding Remarks	7-13

7.1 Introduction

Meeting deadline constraints is of paramount importance in real-time systems. Sometimes, it is not possible to schedule all the tasks to meet their deadlines, a situation that occurs quite often when the system is overloaded. In situations like this, it is often more desirable to complete some portions of every task rather than giving up completely the processing of some tasks. The Imprecise Computation Model was introduced [1–3] to allow for the trade-off of the quality of computations in favor of meeting the deadline constraints. In this model, a task is logically decomposed into two subtasks, mandatory and optional. The mandatory subtask of each task is required to be completed by its deadline, while the optional subtask can be left unfinished. If a task has an unfinished optional subtask, it incurs an error equal to the execution time of its unfinished portion. The Imprecise Computation Model is designed to model an iterative algorithm, where the task initially spends some time for initialization (the mandatory subtask) and then iterates to improve the quality of the solution (the optional subtask). Since the optional subtask corresponds to iterations to improve the quality of the solution, it can be left unfinished and still obtain a solution with a somewhat inferior quality. In this way, we can trade off the quality of the computation in favor of meeting the deadline constraints.

In the Imprecise Computation Model, each task T_i is represented by the quadruple $T_i = (r_i, \bar{d}_i, m_i, o_i)$, where r_i , \bar{d}_i , m_i , and o_i denote its release time, deadline, mandatory subtask's execution time, and optional subtask's execution time, respectively. Let $e_i = m_i + o_i$ denote its total execution time. A schedule for a given task system is said to be *feasible* if each mandatory subtask is executed fully in the time interval between its release time and deadline; a task system is said to be *feasible* if there is a feasible schedule for it. Feasibility of a task system can be determined in at most $O(n^2 \log^2 n)$ time for parallel and identical

processors [4], and $O(n \log n)$ time for a single processor [5]. In this chapter, we assume that all task systems are feasible and all task parameters are rational numbers. Furthermore, we will be concerned with preemptive scheduling only.

Let S be a feasible schedule for a task system TS with n tasks. For each task T_i , let $\alpha(T_i, S)$ denote the amount of time T_i is executed in S . The *error* of T_i , denoted by $\epsilon(T_i, S)$, is defined to be $e_i - \alpha(T_i, S)$. The *total error* of S , denoted by $\epsilon(S)$, is defined to be $\sum_{i=1}^n \epsilon(T_i, S)$. The *maximum error* of S , denoted by $\Upsilon(S)$, is defined to be $\max_{i=1}^n \{\epsilon(T_i, S)\}$. The minimum of total error of TS , denoted by $\epsilon(TS)$, is defined to be $\min\{\epsilon(S) : S \text{ is a feasible schedule for } TS\}$. The minimum of maximum error of TS , denoted by $\Upsilon(TS)$, is defined to be $\min\{\Upsilon(S) : S \text{ is a feasible schedule for } TS\}$. If the importance of the tasks are not identical, we can assign two weights w_i and w'_i to each task T_i , and the resulting task system is called a *weighted* task system. For a weighted task system, the goal is to minimize the total w -weighted error or the maximum w' -weighted error. We use $\epsilon_w(TS)$ to denote the minimum of the total w -weighted error of TS , and $\Upsilon_{w'}(TS)$ to denote the minimum of the maximum w' -weighted error of TS . Note that the w -weight is applied only to the total weighted error, whereas the w' -weight is applied only to the maximum weighted error.

In this chapter, we will give algorithms for finding a schedule that minimizes the total w -weighted error as well as the maximum w' -weighted error. Our study encompasses a single processor, parallel and identical processors, as well as uniform processors. In the next chapter, we will give algorithms for bicriteria scheduling problems and other related problems. For bicriteria scheduling problems, the goal is to minimize the total w -weighted error, subject to the constraint that the maximum w' -weighted error is minimum, or to minimize the maximum w' -weighted error, subject to the constraint that the total w -weighted error is minimum.

Blazewicz [6] was the first to study the problem of minimizing the total w -weighted error for a special case of the weighted task system, where each task has its optional subtask only; i.e., $m_i = 0$ for each $1 \leq i \leq n$. He showed that both parallel and identical processor, and uniform processor systems can be reduced to minimum-cost-maximum-flow problems, which can be transformed into linear programming problems. Blazewicz and Finke [7] later gave faster algorithms for both problems; see also Leung [8]. Potts and Van Wassenhove [9] studied the same problem on a single processor, assuming that all tasks have identical release times and identical weights. They gave an $O(n \log n)$ -time algorithm for preemptive scheduling and showed that the problem becomes NP-hard for nonpreemptive scheduling. They [10] later gave a polynomial approximation scheme and two fully polynomial approximation schemes for the nonpreemptive case.

For imprecise computation tasks, Shih et al. [4] gave an $O(n^2 \log^2 n)$ -time algorithm to minimize the total error on a parallel and identical processor system. Shih et al. [11] and Leung et al. [12] later gave faster algorithms on a single processor that runs in $O(n \log n)$ time. For the weighted case, Shih et al. [4] again showed that the problem can be transformed into a minimum-cost-maximum-flow problem, thus giving an $O(n^2 \log^3 n)$ -time algorithm for parallel and identical processors. For a single processor, Shih et al. [11] gave a faster algorithm that runs in $O(n^2 \log n)$ time, which was subsequently improved by Leung et al. [12] to $O(n \log n + kn)$ time, where k is the number of distinct w -weights. The minimum-cost-maximum-flow approach can be extended to solve the uniform processors case by using the idea of Federgruen and Groenevelt [13] to handle the different speeds of the processors. Using Orlin's [14] minimum-cost-maximum-flow algorithm, the total weighted error problem on uniform processors can be solved in $O(m^2 n^4 \log mn)$ time.

Ho et al. [15] gave an algorithm to minimize the maximum w' -weighted error. Their algorithm runs in $O(n^3 \log^2 n)$ time for parallel and identical processors, and $O(n^2)$ time for a single processor. Recently, Wan et al. [16] extended the ideas of Ho et al. to solve the uniform processors case giving an algorithm that runs in $O(mn^4)$ time.

This chapter is organized as follows. In the next section, we will consider the total w -weighted error—Section 7.2.1 considers the single processor case, Section 7.2.2 considers the parallel and identical processor case, and Section 7.2.3 considers the uniform processor case. In Section 7.3, we will consider the maximum w' -weighted error for the cases of a single processor, parallel and identical processors, and uniform processors. Finally, we draw some concluding remarks in the last section.

7.2 Total Weighted Error

In this section, we will give algorithms to minimize the total weighted error. The case of a single processor, parallel and identical processors, and uniform processors will be given in the next three subsections.

7.2.1 Single Processor

In this subsection, we will give an $O(n \log n + kn)$ -time algorithm for the total weighted error problem (see Leung et al. [12]). Each imprecise computation task j is represented by two subtasks: mandatory (M_j) and optional (O_j). Both subtasks have ready time r_j and deadline \bar{d}_j . The optional subtask has execution time o_i and weight w_i , and the mandatory subtask has execution time m_i and weight $\max\{w_i\} + 1$. We assign the highest weight to each mandatory subtask to ensure that each of them will be executed to completion (i.e., no unfinished mandatory subtask). In this way, we can reduce the problem to that of minimizing the total weighted number of tardy units. Note that this technique is only applicable to a single processor but not applicable to parallel and identical processors or uniform processors. This is because for parallel and identical processors or uniform processors, the schedule might assign the mandatory and optional subtasks of an imprecise computation task to the same time interval on two different processors, which is not allowed in any feasible schedules.

We first describe an algorithm for minimizing the total (unweighted) number of tardy task units; the algorithm will later be extended to solve the weighted case. The algorithm only schedules the nontardy units of a task, assuming that the tardy units are either not scheduled or scheduled at the end. Let the tasks be ordered in descending order of release times. If several tasks have identical release times, they are ordered in descending order of their deadlines. Further ties can be broken arbitrarily. Let $0 = \min\{r_i\} = u_0 < u_1 < \dots < u_p = \max\{\bar{d}_i\}$ be the $p + 1$ distinct integers obtained from the multiset $\{r_1, \dots, r_n, \bar{d}_1, \dots, \bar{d}_n\}$. These $p + 1$ integers divide the time frame into p segments: $[u_0, u_1]$, $[u_1, u_2]$, \dots , $[u_{p-1}, u_p]$. The algorithm uses an $n \times p$ matrix S to represent a schedule, where $S(i, j)$ contains the number of time units task i is scheduled in segment j (i.e., $[u_{j-1}, u_j]$). Below is a formal description of the algorithm.

Algorithm NTU

- (1) **For** $i = 1, \dots, p$ **do:** $l_i \leftarrow u_i - u_{i-1}$.
- (2) **For** $i = 1, \dots, n$ **do:**
 - Find a satisfying $u_a = \bar{d}_i$ and b satisfying $u_b = r_i$.
 - For** $j = a, a - 1, \dots, b + 1$ **do:**
 - $\delta \leftarrow \min\{l_j, e_i\}$.
 - $S(i, j) \leftarrow \delta, l_j \leftarrow l_j - \delta, e_i \leftarrow e_i - \delta$.
 - repeat**
 - repeat**

The algorithm schedules tasks in descending order of their release times. When a task is scheduled, it is assigned from the latest segment $[u_{a-1}, u_a]$ in which it can be scheduled until the earliest segment $[u_b, u_{b+1}]$. Let us examine the time complexity of the algorithm. The time it takes to sort the tasks in descending order of their release times as well as obtaining the set $\{u_0, u_1, \dots, u_p\}$ is $O(n \log n)$. Step 1 of the algorithm takes linear time and a straightforward implementation of Step 2 takes $O(n^2)$ time. Thus, it appears that the running time of the algorithm is $O(n^2)$. However, observe that whenever a segment is scheduled, either all the units of a task are scheduled or the segment is saturated, or both. Hence, at most $O(n)$ segments have positive values. Thus, if we can avoid scanning those segments that have zero values, then Step 2 takes only linear time. As it turns out, this can be done by the special UNION-FIND algorithm owing to Gabow and Tarjan [17].

T_i	r_i	d_i	e_i	w_i
T_1	6	15	5	5
T_2	4	8	3	5
T_3	2	6	3	5
T_4	0	7	3	5
T_5	10	14	3	2
T_6	4	5	1	2
T_7	3	10	6	2
T_8	0	17	3	2
T_9	0	5	4	2

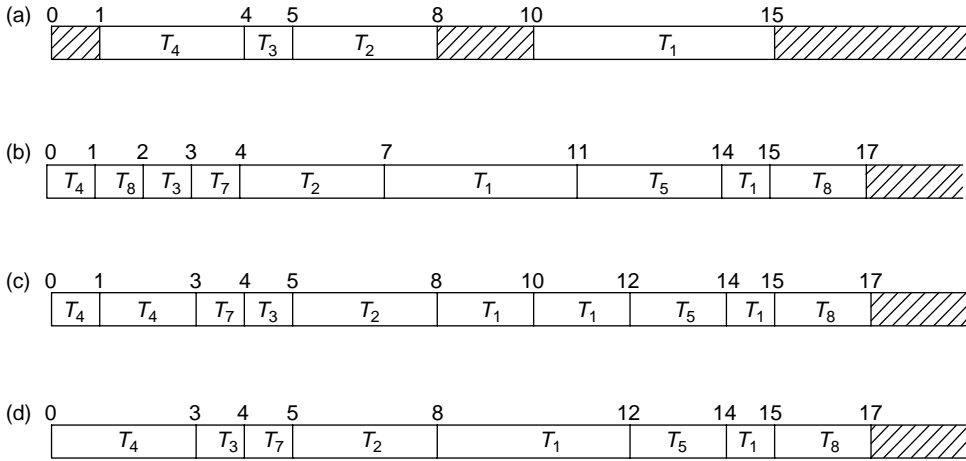


FIGURE 7.1 An example illustrating algorithm WNTU: (a) schedule S obtained after the first phase; (b) schedule S_2 obtained in the second phase; (c) schedule S'_2 obtained in the second phase; and (d) final schedule obtained by Algorithm WNTU.

A schedule produced by Algorithm NTU will be denoted as NTU-schedule. Define a *block* as a maximal time interval in which there is only one task assigned (task block) or the processor is idle (idle block). Without any increase in time complexity, the algorithm can be modified to produce a schedule represented by a doubly linked list of blocks. As shown in Leung et al. [12], the number of blocks in an NTU-schedule is no more than $2n + 1$.

We can use Algorithm NTU to solve the weighted case as follows. Sort the tasks in decreasing order of their weights. Let π_i , $1 \leq i \leq n$, denote the number of nontardy units of task i in an optimal schedule. Once the values of π_i are known, an optimal schedule can be obtained in $O(n \log n)$ time by the Earliest Due Date rule. We determine these values in phases as follows. After j phases, it would have already determined the values $\pi_1, \pi_2, \dots, \pi_j$. In the $(j + 1)$ th phase, it uses Algorithm NTU to solve the unweighted problem for tasks $1, 2, \dots, j + 1$, where the execution times of the first j tasks are π_1, \dots, π_j , and the execution time of the $(j + 1)$ th task is e_{j+1} . Let x be the number of tardy units obtained in the NTU-schedule. π_{j+1} is then set to be $e_{j+1} - x$, and the algorithm proceeds to the next phase. Note that we can assign x tardy units to task $j + 1$ since the first j tasks can be scheduled on time in previous iterations.

The above algorithm makes n calls to Algorithm NTU. Since Algorithm NTU takes linear time after the initial sorting, the running time of the algorithm becomes $O(n^2)$. The drawback of the above approach

is that tasks are scheduled one by one, which slows down the algorithm. To speed up the algorithm, we need to schedule several tasks simultaneously, say all tasks with identical weights. However, if we schedule several tasks together, then it is not clear how to assign tardy units to the tasks. The trick here is to find a way to allocate tardy units to the new tasks so that the previously scheduled tasks remain intact.

Let there be k different weights, $w_1 > w_2 > \dots > w_k$, and let T_j , $1 \leq j \leq k$, be the set of all tasks with weight w_j . We use T to store the tasks (and their execution times) whose nontardy units have already been determined; initially, T is an empty set. Let S be an empty schedule. The tasks are scheduled in phases. At the end of the j th phase, the algorithm would have already determined the nontardy units of the tasks in T_1, \dots, T_j . These tasks and their execution times are stored in T . In the $(j+1)$ th phase, the algorithm uses Algorithm NTU to construct a schedule S_{j+1} for $T \cup T_{j+1}$. It then goes through an adjustment step (described below) transforming S_{j+1} into S'_{j+1} with S as a template. The adjustment step is needed to allocate nontardy units to the new tasks (i.e., tasks in T_{j+1}) in such a way that the previously scheduled tasks (i.e., tasks in T) remain intact. We now set T to be $T \cup T_{j+1}$ and the execution times of the tasks in T are set to the nontardy units in S'_{j+1} . Finally, we apply Algorithm NTU to T to obtain the schedule S before we go to the next phase. We repeat this process until the k th phase is finished.

The adjustment step proceeds as follows. Let there be q blocks in S : $V_i = [v_{i-1}, v_i]$, $1 \leq i \leq q$. S_{j+1} is transformed block by block from V_1 to V_q . Transformation is applied only to the task blocks in S but not to the idle blocks. Let V_i be a task block in S and let task l be scheduled in the block. Let $N(l)$ (resp. $N_{j+1}(l)$) denote the number of time units task l has executed in S (resp. S_{j+1}) from the beginning until time v_i . If $N(l) > N_{j+1}(l)$, then assign $(N(l) - N_{j+1}(l))$ more time units to task l within V_i in S_{j+1} , by removing any task, except task l that was originally assigned in V_i . (Note that this reassignment can always be done.) Otherwise, no adjustment is needed.

Figure 7.1 gives a set of tasks with two distinct weights. The schedule S after the first phase is shown in Figure 7.1a. S_2 and S'_2 are shown in Figures 7.1b and 7.1c, respectively. Finally, the schedule S after the second phase is shown in Figure 7.1d, which is an optimal schedule for the set of tasks.

Algorithm WNTU

- (1) Let there be k distinct weights, $w_1 > \dots > w_k$, and let T_j be the set of tasks with weight w_j .
- (2) Let S be an empty schedule and T be an empty set.
- (3) **For** $j = 1, \dots, k$ **do**:
 - $S_j \leftarrow$ schedule obtained by Algorithm NTU for $T \cup T_j$.
 - Begin** (Adjustment Step)
 - Let there be q blocks in S : $V_i = [v_{i-1}, v_i]$, $1 \leq i \leq q$.
 - For** $i = 1, \dots, q$ **do**:
 - If** V_i is a task block in S , **then**
 - Let task l be executed within V_i in S . Let $N(l)$ (resp. $N_j(l)$) be the number of time units job l has executed in S (resp. S_j) from the beginning until time v_i .
 - If** $N(l) > N_j(l)$, **then**
 - assign $(N(l) - N_j(l))$ more time units to task l within V_i in S_j , by replacing any task, except task l , that was originally assigned within V_i .
 - endif**
 - endif**
 - repeat**
 - End**
 - $S'_j \leftarrow S_j$.
 - Set the execution time of each task in T_j to be the number of nontardy units in S'_j .
 - $T \leftarrow T \cup T_j$.
 - $S \leftarrow$ schedule obtained by Algorithm NTU for T .
 - repeat**

Let us examine the time complexity of Algorithm WNTU. Observe that Algorithm WNTU utilizes Algorithm NTU to construct schedules for various subsets of tasks. Algorithm NTU requires that the release times and deadlines of the tasks be ordered. With an initial sort of the release times and deadlines of the tasks, we can obtain in linear time an ordering of the release times and deadlines for various subsets of tasks. Once the tasks are ordered, Algorithm NTU requires only linear time to construct a schedule.

Steps 1 and 2 of Algorithm WNTU take $O(n \log n)$ time and Step 3 is iterated k times. If we can show that each iteration of Step 3 takes linear time (after an initial sort), then the overall running time of Algorithm WNTU becomes $O(n \log n + kn)$. It is clear that every substep in Step 3, with the possible exception of the adjustment step, takes linear time. We now show that the adjustment step can be implemented in linear time. As mentioned before, Algorithm NTU can be implemented, with no increase in time complexity, to produce a schedule represented by a doubly linked list of blocks. Thus, we may assume that S and S_j are in this representation. The adjustment process is performed by traversing the two linked lists, modifying S_j if necessary, as the lists are traversed. As mentioned before, the number of blocks is linear. The values $N(l)$ and $N_j(l)$ can be obtained with the help of two one-dimensional arrays L and L' : $L(l)$ (resp. $L'(l)$) contains the number of time units task l has executed in S (resp. S_j) since the beginning. L and L' initially have zero in each entry, and they are updated as the linked lists are traversed. Thus, the adjustment process takes linear time. From the above discussions, we have the following theorem.

Theorem 7.1

The total weighted error problem can be solved in $O(n \log n + kn)$ time on a single processor, where k is the number of distinct w -weights.

7.2.2 Parallel and Identical Processors

In this subsection, we consider the parallel and identical processor case. As we will see later, this problem can be solved by a network flow approach. However, it is not as straightforward as it appears since the mandatory subtasks must be executed in full and the optional subtasks cannot be executed in parallel with their corresponding mandatory subtask. Nonetheless, it can still be solved by a network flow approach. We first describe the unweighted case. Figure 7.2 shows such a network, which we will denote by $G(x)$; x is the flow capacity on the arc from vertex O to the sink S_2 . The source and sink are represented by S_1 and S_2 , respectively. Each task j has three vertexes— M_j , O_j , and T_j . There are arcs from the source to M_j and O_j , $1 \leq j \leq n$, with flow capacities set to m_j and o_j , respectively. Both vertexes M_j and O_j have an arc to T_j , with flow capacities set to m_j and o_j , respectively. In addition, each vertex O_j has an arc to a new vertex O and the flow capacity on the arc is o_j . Each time interval $[u_{i-1}, u_i]$, $1 \leq i \leq p$, is represented by an interval vertex I_i . There is an arc from each T_j to each interval vertex in which it can execute; the flow capacity on the arc is the length of the interval. Finally, each interval vertex has an arc to the sink and the flow capacity on the arc is m times the length of the interval. Vertex O has an arc to the sink with flow capacities x , which is a parameter that we can set.

Consider the network $G(0)$. Suppose we replace the flow capacity on each arc that has a flow capacity o_j , $1 \leq j \leq n$, by zero. Let this network be denoted by $G'(0)$. It is clear that $G'(0)$ is the network for mandatory subtasks only. First, we find the maximum flow in $G'(0)$. If the maximum flow is less than $\sum m_j$, then the set of tasks is infeasible. So we may assume that the maximum flow is exactly $\sum m_j$. Next, we find the maximum flow in $G(0)$, say u . If $u = \sum e_j$, then there is a schedule with no error; otherwise, the total error is $\epsilon = \sum e_j - u$. We now set x to be ϵ . The maximum flow in $G(\epsilon)$ is $\sum e_j$ and the flow pattern gives a schedule with total error ϵ . The network has $O(n)$ vertexes and $O(n^2)$ arcs; again, we can reduce the number of arcs to $O(n \log n)$ by using a balanced binary tree to represent the time intervals. Thus, the running time is again $O(n^2 \log^2 n)$.

Before we give the algorithm for the total weighted error, we first consider the problem of minimizing the total weighted number of tardy units. Consider a network constructed as follows. Let the source and sink be represented by S_1 and S_2 , respectively. Each task j will be represented by a task vertex T_j . There is an arc from S_1 to each T_j with the maximum flow on the arc equal to e_j and the cost of the flow equal to zero.

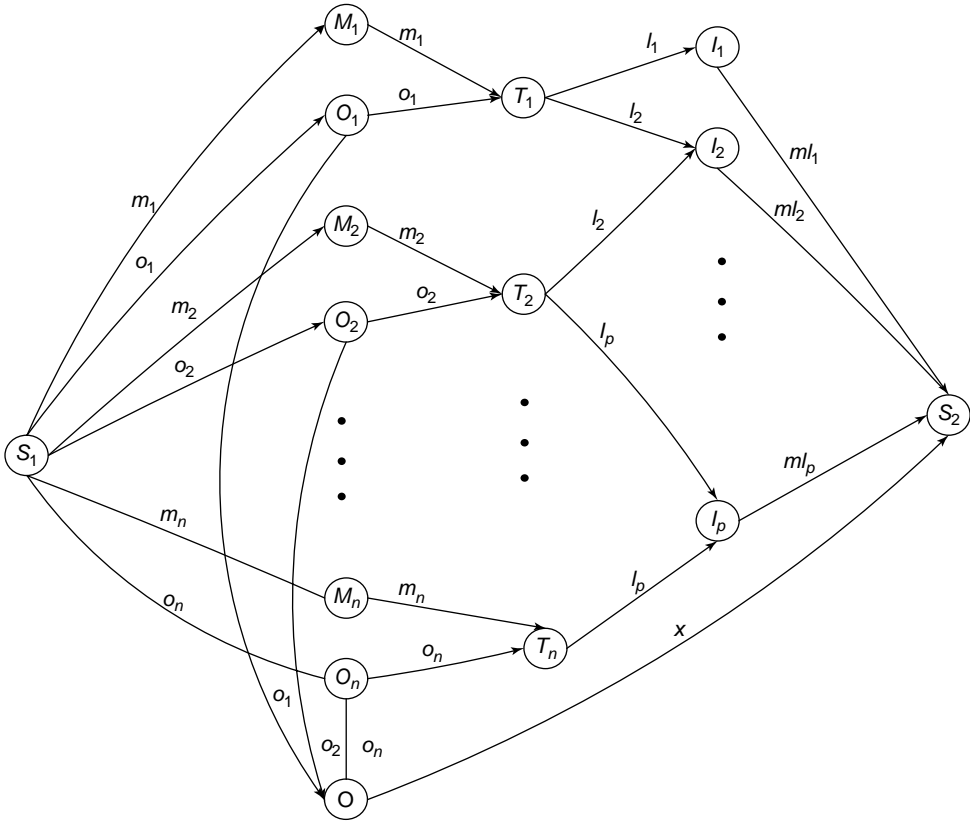


FIGURE 7.2 Network for minimizing total error for imprecise computation tasks.

For each segment $[u_{i-1}, u_i]$, $1 \leq i \leq p$, we create an interval vertex I_i that corresponds to the segment. Each task vertex T_j has an arc to each of the interval vertexes in which task j can execute. The maximum flow on each of these arcs is the length of the interval (i.e., $l_i = u_i - u_{i-1}$ for the interval vertex I_i) and the cost of the flow is zero. In addition, each task vertex T_j has an arc to S_2 with the maximum flow equal to e_j and the cost equal to w_j . Finally, each interval vertex I_i has an arc to S_2 with the maximum flow equal to ml_i and the cost equal to zero, where m is the number of processors. The maximum flow in this network is $\sum e_j$. The minimum-cost-maximum-flow yields a schedule with the minimum weighted number of tardy units.

Figure 7.3 shows the network described above. There are $O(n)$ vertexes and $O(n^2)$ arcs in the network; again, the number of arcs can be reduced to $O(n \log n)$. Using Orlin's minimum-cost-maximum-flow algorithm, the total weighted number of tardy units problem can be solved in $O(n^2 \log^3 n)$ time.

We can combine the ideas of the networks in Figures 7.2 and 7.3 to solve the total weighted error problem for imprecise computation tasks. From the above discussions, we have the following theorem.

Theorem 7.2

The total error problem can be solved in $O(n^2 \log^2 n)$ time on $m \geq 2$ parallel and identical processors. The total weighted error problem can be solved in $O(n^2 \log^3 n)$ time.

7.2.3 Uniform Processors

The minimum-cost-maximum-flow approach can be extended to solve the uniform processors case by using the idea in Ref. 13 to handle the different speeds of the processors. Again, we will first describe the solution for the total weighted number of tardy units problem. Assume that processing speeds are ordered

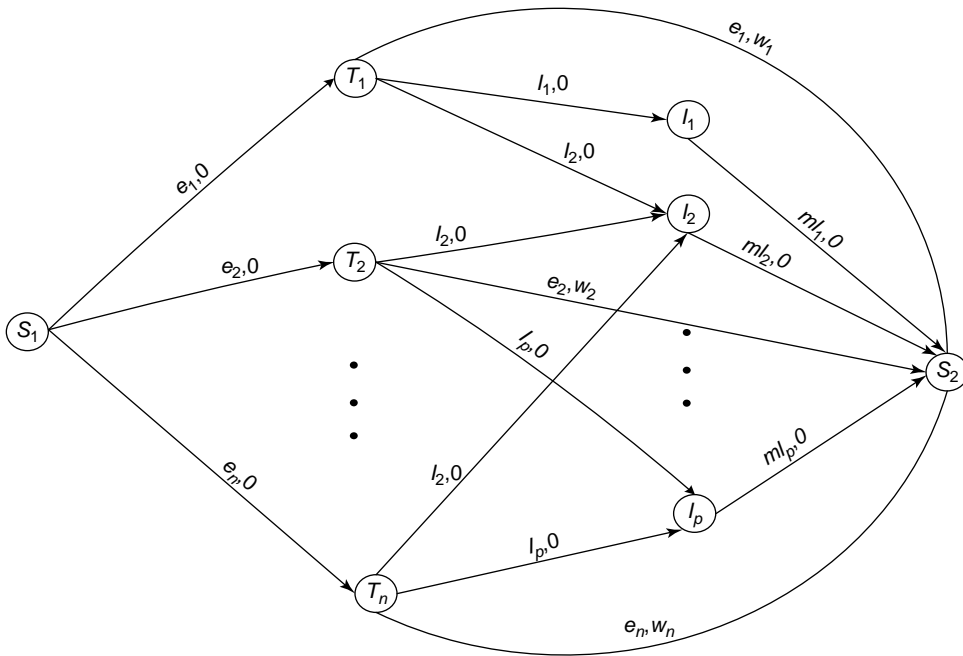


FIGURE 7.3 Network for minimizing the weighted number of tardy units on identical processors.

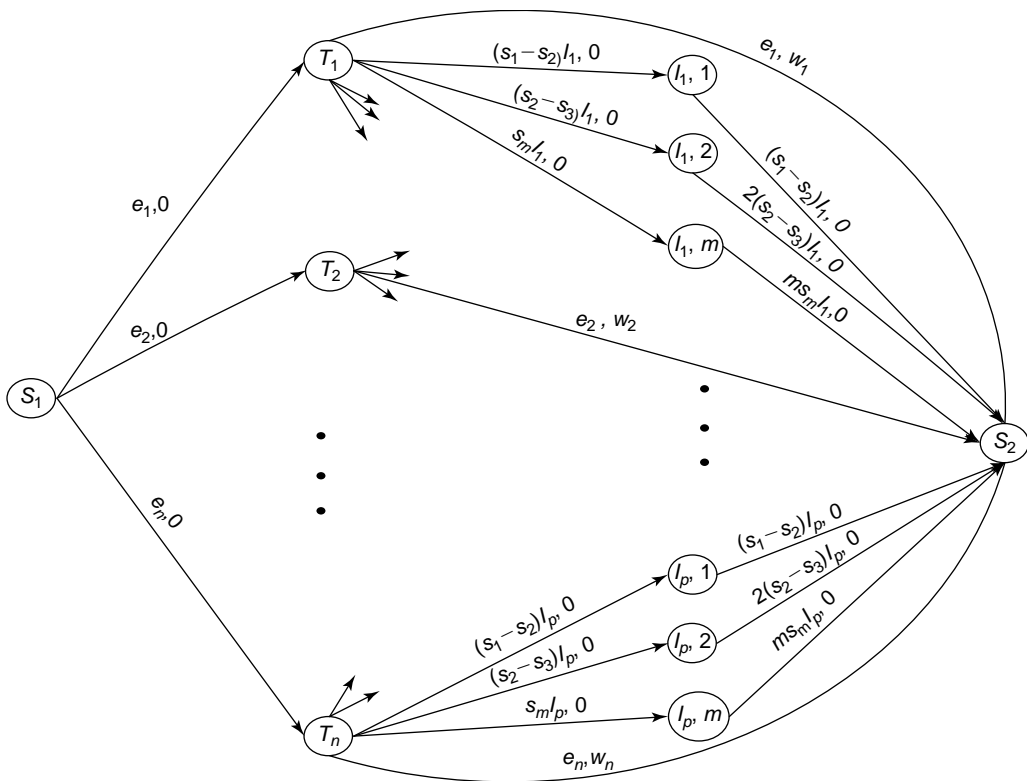


FIGURE 7.4 Network for minimizing the weighted number of tardy units on uniform processors.

in such a way that $s_1 > s_2 > \dots > s_m$ (the case of identical speeds for some processors can also be handled [13]). In each interval I_i , no task can obtain more than $s_1 l_i$ units of processing, two tasks not more than $(s_1 + s_2) l_i$ units, and so on. The network is the same as in Figure 7.3, except that each interval vertex I_i is expanded into m vertexes, $(I_i, 1), (I_i, 2), \dots, (I_i, m)$. If task j can execute in the segment $[u_{i-1}, u_i]$, then the task vertex T_j has an arc to each vertex interval (I_i, k) , $1 \leq k \leq m$, with maximum flow equal to $(s_k - s_{k+1}) l_i$ and cost equal to zero. We assume that $s_{m+1} = 0$. Finally, each interval vertex (I_i, k) has an arc to the sink with maximum flow equal to $k(s_k - s_{k+1}) l_i$ and cost equal to zero.

Figure 7.4 shows the network described above. There are $O(mn)$ vertexes and $O(mn^2)$ arcs; it is not clear if we can reduce the number of arcs in this case. Using Orlin's minimum-cost-maximum-flow algorithm, the total weighted number of tardy units problem can be solved in $O(m^2 n^4 \log mn)$ time.

We can also combine the ideas of the networks in Figures 7.2 and 7.4 to solve the total weighted error problem on uniform processors. The running time is again $O(m^2 n^4 \log mn)$. From the above discussions, we have the following theorem.

Theorem 7.3

The total weighted error problem can be solved in $O(m^2 n^4 \log mn)$ time on $m \geq 2$ uniform processors.

7.3 Maximum Weighted Error

In this section, we will give algorithms for the maximum weighted error problem; the algorithms are described in detail in Refs. 15 and 16. The algorithms for the single processor, identical and parallel processors, and uniform processors all operate with the same framework; the only difference is in the step of computing the total error, which is different for the three cases.

We first define some notations and terminologies. As defined in Section 7.2.1, let $\min\{r_i\} = t_0 < t_1 < \dots < t_p = \max\{\bar{d}_i\}$ be all the distinct values of the multiset $\{r_1, \dots, r_n, \bar{d}_1, \dots, \bar{d}_n\}$. These $p + 1$ values divide the time frame into p intervals: $[t_0, t_1], [t_1, t_2], \dots, [t_{p-1}, t_p]$, denoted by u_1, u_2, \dots, u_p . The length of the interval u_j , denoted by l_j , is defined to be $t_j - t_{j-1}$. A schedule S can be described by a $n \times p$ matrix SM_S , such that $SM_S(i, j)$ contains the amount of processor time assigned to T_i in u_j . The scheduling matrix SM_S satisfies the following constraints: (1) $SM_S(i, j) \leq l_j$ for $1 \leq i \leq n$ and $1 \leq j \leq p$; (2) $\sum_{i=1}^n SM_S(i, j) \leq m \times l_j$ for $1 \leq j \leq p$. Given a scheduling matrix $SM_S(i, j)$, a schedule can be constructed easily by McNaughton's rule [19]. In the following, we will describe a schedule by its scheduling matrix. An interval u_j is said to be *unsaturated* in the schedule S if $m \times l_j > \sum_{i=1}^n SM_S(i, j)$; otherwise, it is said to be *saturated*.

A task T_i is said to be *eligible* in the time interval $[t, t']$ if $r_i \leq t \leq t' \leq \bar{d}_i$. A task T_i eligible in the interval u_j is said to be *fully* scheduled in u_j in S if $SM_S(i, j) = l_j$; otherwise, it is said to be *partially* scheduled. T_i is said to be *precisely* scheduled in S if $\epsilon(T_i, S) = 0$; otherwise, it is said to be *imprecisely* scheduled. A task T_r is said to be *removable* if $\epsilon(TS) = \epsilon(TS - \{T_r\})$; otherwise, it is said to be *unremovable*. A task T_i is said to be *shrinkable* if $o_i > 0$; otherwise, it is said to be *unshrinkable*.

Our algorithm, to be called Algorithm MWE, proceeds in phases. Starting with the original task system TS , it constructs a sequence of task systems $TS(i)$, one in each phase. Suppose that we are in the i th phase. The task system $TS(i)$ is obtained by first initializing it to be $TS(i - 1)$. Then, the execution times of the optional subtasks of all the shrinkable tasks will be shrunk by a certain amount and some (not necessarily all) removable tasks may also be removed from $TS(i)$ before the algorithm proceeds to the next phase.

Initially, $TS(0)$ is initialized to TS and $\eta(0)$ (for storing $\Upsilon_{w'}(TS)$) is initialized to zero. Then, a schedule $S(0)$ with minimum total error is constructed for $TS(0)$. If $\epsilon(S(0)) = 0$, the algorithm terminates; otherwise, it proceeds to the next phase. (In the following, we will call the initialization phase the 0th phase.) Suppose that the algorithm is in the i th phase. It first initializes $TS(i)$ to $TS(i - 1)$. Then, it apportions $\epsilon(TS(i - 1))$ amount of error to all the shrinkable tasks in $TS(i)$ in such a way that the maximum w' -weighted error is minimized. It is easy to verify that the maximum w' -weighted error would be minimized if the execution

time of the optional subtask of each shrinkable task T_j in $TS(i)$ is shrunk by $\epsilon(TS(i-1))/(\Delta \times w'_j)$ amount (and hence incurs the same amount of w' -weighted error), where $\Delta = \sum_{T_k \text{ is shrinkable}, T_k \in TS(i)} 1/w'_k$. Thus, for each shrinkable task $T_j \in TS(i)$, $o_j(i)$ is shrunk by $\epsilon(TS(i-1))/(\Delta \times w'_j)$ amount (if it can), and T_j is marked as unshrinkable if $o_j(i)$ becomes zero. Now, $\eta(i)$ is set to $\eta(i-1) + \epsilon(TS(i-1))/\Delta$. A schedule $S(i)$ with minimum total error is then constructed for $TS(i)$. If $\epsilon(S(i)) = 0$, the algorithm terminates and $\eta(i)$ gives the $\Upsilon_{w'}(TS)$. Otherwise, it will remove some (not necessarily all) removable tasks in $TS(i)$ before proceeding to the next phase. This is done by finding an unsaturated interval u_j (if any) in $S(i)$ in which a nonempty set of tasks is partially scheduled; these (removable) tasks will then be removed from $TS(i)$. A formal description of the algorithm is given below.

Algorithm MWE

1. $\eta(0) \leftarrow 0$; $TS(0) \leftarrow TS$; $i \leftarrow 0$;
2. Construct a schedule $S(0)$ with minimum total error for $TS(0)$;
3. If $\epsilon(S(0)) = 0$, output $\eta(0)$ and stop;
4. $i \leftarrow i + 1$; $TS(i) \leftarrow TS(i-1)$; $\Delta \leftarrow \sum_{T_k \text{ is shrinkable}, T_k \in TS(i)} 1/w'_k$;
5. For each shrinkable task T_j in $TS(i)$ do:
 - $y_j \leftarrow \epsilon(S(i-1))/(\Delta \times w'_j)$;
 - $o_j(i) \leftarrow \max\{0, o_j(i-1) - y_j\}$;
 - If $o_j(i) = 0$, mark T_j as unshrinkable;
6. $\eta(i) \leftarrow \eta(i-1) + \epsilon(S(i-1))/\Delta$;
7. Construct a schedule $S(i)$ with minimum total error for $TS(i)$;
8. If $\epsilon(S(i)) = 0$, output $\eta(i)$ and stop;
9. Find an unsaturated interval u_j in $S(i)$ in which a nonempty set of tasks is partially scheduled, and delete these tasks from $TS(i)$;
10. Go to Step 4.

We use the task system $TS(0)$ shown in Figure 7.5a to illustrate how the algorithm works. Figure 7.5b shows the schedule $S(0)$ with minimum total error for $TS(0)$. Since $\epsilon(S(0)) > 0$, the algorithm proceeds to the first phase. Figure 7.6a shows the y_j value for each shrinkable task T_j in $TS(1)$. Figure 7.6b shows the task system $TS(1)$ after shrinking the execution times of the optional subtasks of the shrinkable tasks. The schedule $S(1)$ is shown in Figure 7.6c. From $S(1)$, we see that there is no removable task in $TS(1)$.

Since $\epsilon(S(1)) > 0$, the algorithm proceeds to the second phase. Figure 7.7a shows the y_j value for each shrinkable task T_j in $TS(2)$ and Figure 7.7b shows the task system $TS(2)$ after the shrinking operation. No new unshrinkable task is created in this phase. Figure 7.7c shows the schedule $S(2)$. From $S(2)$, we see

(a)

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
r_i	0	0	0	6	6	2	6	6
\bar{d}_i	6	6	5	9	9	10	9	10
m_i	1	2	0	1	1	2	0	1
o_i	7	6	4	5	7	5	5	3
w'_i	4	2	1	1	2	2	4	1

(b)

0	6	8	9	10
T_1	T_4	T_8		
T_2	T_5	T_6		

FIGURE 7.5 An example illustrating Algorithm MWE: (a) an example task system $TS(0)$; (b) the schedule $S(0)$ with $\epsilon(S(0)) = \epsilon(TS(0)) = 30$.

(a)

i	1	2	3	4	5	6	7	8
y_i	1.5	3	6	6	3	3	1.5	6

(b)

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
r_i	0	0	0	6	6	2	6	6
\bar{d}_i	6	6	5	9	9	10	9	10
m_i	1	2	0	1	1	2	0	1
o_i	5.5	3	0	0	4	2	3.5	0
w'_i	4	2	1	1	2	2	4	1

(c)

0					5	6	7		9	10
T_1						T_4	T_6			
T_2					T_6	T_5			T_8	

FIGURE 7.6 The first iteration: (a) $\epsilon(S(0))/(\Delta \times w'_i)$, $\epsilon(S(0)) = 30$ and $\Delta = 5$; (b) $TS(1)$ obtained from $TS(0)$; and (c) the schedule $S(1)$ with $\epsilon(S(1)) = \epsilon(TS(1)) = 6$.

(a)

i	1	2	3	4	5	6	7	8
y_i	0.75	1.5	**	**	1.5	1.5	0.75	**

(b)

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
r_i	0	0	0	6	6	2	6	6
\bar{d}_i	6	6	5	9	9	10	9	10
m_i	1	2	0	1	1	2	0	1
o_i	4.75	1.5	0	0	2.5	0.5	2.75	0
w'_i	4	2	1	1	2	2	4	1

(c)

0

3.5

5

5.75

6

6.75

7

8.75

9

10



T_1					T_4	T_5		T_6
T_2			T_6			T_7		T_4 T_8

FIGURE 7.7 The second iteration: (a) $\epsilon(S(1))/(\Delta \times w'_i)$, $\epsilon(S(1)) = 6$ and $\Delta = 2$; (b) $TS(2)$ obtained from $TS(1)$; and (c) the schedule $S(2)$ with $\epsilon(S(2)) = \epsilon(TS(2)) = 1.25$.

that T_1 , T_2 , and T_6 are removable tasks and hence they are removed from $TS(2)$ before proceeding to the next phase.

Since $\epsilon(S(2)) > 0$, the algorithm proceeds to the third phase. Figures 7.8a and 7.8b show the y_i values and the task system $TS(3)$ after the shrinking operation, respectively. Figure 7.8c shows the schedule $S(3)$. Since $\epsilon(S(3)) = 0$, the algorithm outputs $\Upsilon_{w'}(TS)$ (which happens to be $10\frac{2}{3}$) and terminates.

Owing to space constraints, we will omit the correctness proof of Algorithm MWE; it can be found in Ref. 15. We now analyze the time complexity of Algorithm MWE. Algorithm MWE iterates Steps 5–9 several times. Inside the loop (Step 5 to Step 9), the most expensive step is Step 7, which constructs a schedule with minimum total error for $TS(i)$. From the analysis in the previous section, we know the

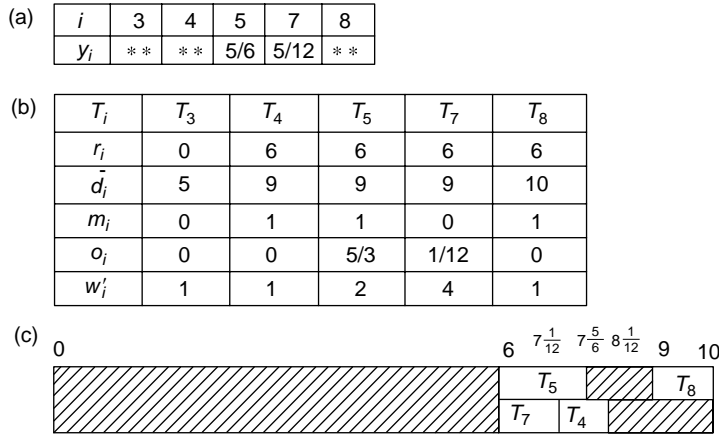


FIGURE 7.8 The third iteration: (a) $\epsilon(S(2))/(\Delta \times w'_i)$, $\epsilon(S(2)) = 1.25$ and $\Delta = 0.75$; (b) $TS(3)$ obtained from $TS(2)$; and (c) the schedule $S(3)$ with $\epsilon(S(3)) = \epsilon(TS(3)) = 0$.

time complexity for finding the minimum total error. Thus, if we can bound the number of iterations the algorithm needs to go through, then we can figure out the time complexity of the algorithm. The next lemma is instrumental in analyzing the number of iterations; the proof is omitted.

Lemma 7.1

If Algorithm MWE does not terminate in the i th phase, then either a task is marked as unshrinkable in Step 5 or a task is removed in Step 9 in the i th phase.

With Lemma 7.1, we can prove the following theorem.

Theorem 7.4

The time complexity of Algorithm MWE is $O(n^2)$ for a single processor, $O(n^3 \log^2 n)$ for identical and parallel processors, and $O(m^2 n^5 \log mn)$ for uniform processors.

Proof

By Lemma 7.1, a task is either marked unshrinkable or removed in each phase. Thus, in the worst case, a task is marked in one phase and then removed in a subsequent phase. Consequently, the remaining task will either be marked unshrinkable or removed in the $(2n - 1)$ th phase, and hence Algorithm MWE will terminate by the $(2n - 1)$ th phase.

For a single processor, the algorithm given in the previous section constructs a schedule with minimum total error in $O(n)$ time after an initial sort of the tasks in descending order of the release times. Thus, with a preprocessing step, Step 7 of Algorithm MWE can be done in $O(n)$ time. We will show that Step 9 of the algorithm can also be done in $O(n)$ time by using a preprocessing step. In $O(n^2)$ time, we can construct the sets of tasks eligible in all the intervals. The tasks will be doubly linked according to intervals and according to tasks. As tasks are removed in each phase, they will be deleted from the doubly linked lists. In Step 9, we can locate in $O(n)$ time the first unsaturated interval in which there is a nonempty set of eligible tasks. All these tasks must be partially scheduled and hence can be removed. Thus, Step 9 can be done in $O(n)$ time. Hence the time complexity of Algorithm MWE is $O(n^2)$ for a single processor.

For parallel and identical processors, Step 7 can be done in $O(n^2 \log^2 n)$ time by using the algorithm given in the previous section. Step 9 can be done in at most $O(n^2)$ time, since there are at most $O(n)$ intervals that need be checked and for each interval there are at most n tasks eligible. Thus, the time complexity of Algorithm MWE is $O(n^3 \log^2 n)$ for parallel and identical processors.

For uniform processors, Step 7 can be done in $O(m^2 n^4 \log mn)$ time by using an algorithm given in the previous section and Step 9 can be done in at most $O(n^2)$ time. Thus, the time complexity of Algorithm MWE is $O(m^2 n^5 \log mn)$ for uniform processors.

7.4 Concluding Remarks

In this chapter, we have given polynomial-time algorithms for the total weighted error problem and the maximum weighted error problem. Our algorithms are for a single processor, parallel and identical processors, and uniform processors.

Choi et al. [20] have given a faster algorithm to obtain an approximate value for the maximum weighted error problem on a single processor. Their algorithm runs in time $O(n \log n + cn)$, where

$$c = \min \left\{ q : \text{Error Bound} \leq \frac{\max\{w_i \times o_i\}}{2^q} \right\}$$

The *Error Bound* is the bound on the difference between the approximate value and the optimal value. Recently, Wan et al. [16] extended their algorithm to the case of parallel and identical processors, and the case of uniform processors. They obtained algorithms that run in time $O(cn^2 \log^2 n)$ for parallel and identical processors, and $O(cmn^3)$ for uniform processors.

References

1. K.-J. Lin, S. Natarajan, and J. W. S. Liu, Concord: a distributed system making use of imprecise results, in *Proc. of COMPSAC '87*, Tokyo, Japan, 1987.
2. K.-J. Lin, S. Natarajan, and J. W. S. Liu, Imprecise results: utilizing partial computations in real-time systems, in *Proc. of the 8th Real-Time Systems Symposium*, San Francisco, CA, 1987.
3. K.-J. Lin, S. Natarajan, and J. W. S. Liu, Scheduling real-time, periodic job using imprecise results, in *Proc. of the 8th Real-Time Systems Symposium*, San Francisco, CA, 1987.
4. W.-K. Shih, J. W. S. Liu, J.-Y. Chung, and D. W. Gillies, Scheduling tasks with ready times and deadlines to minimize average error, *ACM Operating Systems Review*, July 1989.
5. W. A. Horn, Some simple scheduling algorithms, *Naval Research Logistics Quarterly*, 21, 177–185, 1974.
6. J. Blazewicz, Scheduling preemptible tasks on parallel processors with information loss, *Technique et Science Informatiques*, 3, 415–420, 1984.
7. J. Blazewicz and G. Finke, Minimizing weighted execution time loss on identical and uniform processors, *Information Processing Letters*, 24, 259–263, 1987.
8. J. Y.-T. Leung, Minimizing total weighted error for imprecise computation tasks and related problems, in J. Y.-T. Leung (ed), *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, CRC Press, Boca Raton, FL, 2004, pp. 34-1–34-16.
9. C. N. Potts and L. N. Van Wassenhove, Single machine scheduling to minimize total late work, *Operations Research*, 40, 586–595, 1992.
10. C. N. Potts and L. N. Van Wassenhove, Approximation algorithms for scheduling a single machine to minimize total late work, *Operations Research Letters*, 11, 261–266, 1992.
11. W.-K. Shih, J. W. S. Liu, and J.-Y. Chung, Algorithms for scheduling imprecise computations with timing constraints, *SIAM Journal on Computing*, 20, 537–552, 1991.
12. J. Y.-T. Leung, V. K. M. Yu, and W.-D. Wei, Minimizing the weighted number of tardy task units, *Discrete Applied Mathematics*, 51, 307–316, 1994.
13. A. Federgruen and H. Groenevelt, Preemptive scheduling of uniform machines by ordinary network flow techniques, *Management Science*, 32, 341–349, 1986.

14. J. B. Orlin, A faster strongly polynomial minimum cost flow algorithm, in *Proc. of 20th ACM Symposium on Theory of Computing*, 377–387, 1988.
15. K. I-J. Ho, J. Y-T. Leung, and W.-D. Wei, Minimizing maximum weighted error for imprecise computation tasks, *Journal of Algorithms*, 16, 431–452, 1994.
16. G. Wan, J. Y-T. Leung, and M. L. Pinedo, Scheduling imprecise computation tasks on uniform processors, Working paper.
17. H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *Journal of Computer and System Science*, 30, 209–221, 1985.
18. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
19. R. McNaughton, Scheduling with deadlines and loss functions, *Management Science*, 6, 1–12, 1959.
20. K. Choi, G. Jung, T. Kim, and S. Jung, Real-time scheduling algorithm for minimizing maximum weighted error with $O(N \log N + cN)$ complexity, *Information Processing Letters*, 67, 311–315, 1998.

8

Imprecise Computation Model: Bicriteria and Other Related Problems

8.1	Introduction	8-1
8.2	Total w -Weighted Error with Constraints	8-2
8.3	Maximum w' -Weighted Error with Constraints	8-3
8.4	0/1-Constraints	8-6
	Feasibility Test • Total Error • Number of Imprecisely Scheduled Tasks	
8.5	Conclusions	8-10

Joseph Y-T. Leung

New Jersey Institute of Technology

8.1 Introduction

In the last chapter, we have given algorithms for the total w -weighted error and the maximum w' -weighted error problems. In this chapter, we will consider bicriteria scheduling problems. In the bicriteria scheduling problems, we are concerned with (1) minimizing the total w -weighted error, subject to the constraint that the maximum w' -weighted error is minimum and (2) minimizing the maximum w' -weighted error, subject to the constraint that the total w -weighted error is minimum. For a given task system TS , we use $\epsilon_w^{w'}(TS)$ to denote the minimum total w -weighted error, subject to the constraint that the maximum w' -weighted error is minimum. Similarly, we use $\Upsilon_{w'}^w(TS)$ to denote the maximum w' -weighted error, subject to the constraint that the total w -weighted error is minimum. In Sections 8.2 and 8.3, we shall give algorithms to compute $\epsilon_w^{w'}(TS)$ and $\Upsilon_{w'}^w(TS)$, respectively.

The Imprecise Computation Model has also been studied under the 0/1-constraint, where each optional subtask is either fully executed or totally discarded. With the 0/1-constraint, two problems have been studied: (1) minimize the total error and (2) minimize the number of *imprecisely scheduled* tasks (i.e., tasks whose optional subtasks have been discarded). The 0/1-constraint is motivated by some practical applications. In real life, many tasks can be implemented by either a fast or a slow algorithm, with the slow algorithm producing better-quality results than the fast one. Owing to deadline constraints, it may not be possible to meet the deadline of every task if each task were to execute the slow version. Thus, the problem of scheduling tasks with primary version (slow algorithm) and alternate version (fast algorithm) can be reduced to one of scheduling with 0/1-constraint. The execution time of the fast algorithm is the mandatory execution time, while the execution time of the slow algorithm is the total execution time (i.e., mandatory plus optional execution time).

In this chapter, we shall be concerned with scheduling imprecise computation tasks with 0/1-constraint on a single processor. In classical scheduling theory, Lawler [1] has given an algorithm to solve the problem

of preemptively scheduling a set of tasks with release dates on a single processor so as to minimize the weighted number of tardy tasks. In this problem, each task has a processing time, weight, release date, and due date. A task cannot start until its release date and it is expected to be complete by its due date. Preemption is allowed. A task is tardy if it is not completed by its due date. The problem is to find a schedule with the minimum weighted number of tardy tasks. Lawler's algorithm runs in $O(nk^2W^2)$ time, where n is the number of tasks, k is the number of release dates, and W is the total weights of the tasks. In the following we will show that Lawler's algorithm can be used to solve the above two 0/1-constraint scheduling problems on a single processor.

Consider the problem of minimizing the total error. Let TS be a set of n imprecise computation tasks and let $\sigma = \sum_{i=1}^n o_i$. For each task T_i , we create two tasks, an M -task with execution time m_i and weight $\sigma + 1$ and an O -task with execution time o_i and weight o_i . Both tasks have release date r_i and due date \bar{d}_i . It is clear that a schedule for the M - and O -tasks that minimizes the weighted number of tardy tasks is also a feasible schedule that minimizes the total error for TS . Using Lawler's algorithm, such a schedule can be found in $O(n^5\sigma^2)$ time. Hence it can be solved in pseudopolynomial time. We note that the problem of minimizing the total error is NP-hard in the ordinary sense [2].

Now consider the problem of minimizing the number of imprecisely scheduled tasks. Let TS be a set of n imprecise computation tasks. For each task T_i , we create two tasks—an M -task with execution time m_i and weight $n + 1$ and an O -task with execution time o_i and weight 1. Both tasks have release date r_i and due date \bar{d}_i . It is clear that a schedule for the M - and O -tasks that minimizes the weighted number of tardy tasks is also a feasible schedule that minimizes the number of imprecisely scheduled tasks for TS . Using Lawler's algorithm, such a schedule can be found in $O(n^7)$ time.

Since the running times of the above algorithms are unacceptably high, there is a need for fast approximation algorithms. Ho et al. [3] gave two approximation algorithms, both of which run in $O(n^2)$ time. The first one, the *largest-optional-execution-time-first* algorithm, is used to minimize the total error. It has been shown [3] that the total error produced by the algorithm is at most three times that of the optimal solution and the bound is tight. The second algorithm, the *smallest-optional-execution-time-first* algorithm, is used to minimize the number of imprecisely scheduled tasks. It was shown [3] that the number of imprecisely scheduled tasks produced by the algorithm is at most two times that of an optimal solution and the bound is tight.

In this chapter, we will present these two approximation algorithms in Section 8.4. In Section 8.4.1, we will give an algorithm to test if a set of tasks is feasible. In Section 8.4.2, we will present the largest-optional-execution-time-first algorithm and show that it has a worst-case bound of 3. In Section 8.4.3, we will present the smallest-optional-execution-time-first algorithm and show that it has a worst-case bound of 2. Finally, we will draw some concluding remarks in the last section.

8.2 Total w -Weighted Error with Constraints

In this section, we will give an algorithm for minimizing the total w -weighted error, subject to the constraint that the maximum w' -weighted error is minimum (see Ref. 4 for more details). In other words, from among all the schedules that minimize the maximum w' -weighted error, the algorithm will find the one with the minimum total w -weighted error.

The basic idea of the algorithm is as follows. Let TS be a task system and let S be a feasible schedule. If S were to have the property that $\Upsilon_{w'}(S) = \Upsilon_{w'}(TS)$, then each task T_i in TS must be assigned at least $\alpha_i = m_i + \max\{0, o_i - \Upsilon_{w'}(TS)/w'_i\}$ amount of processor times in S . To ensure that T_i is assigned at least α_i amount of processor times, we modify TS by resetting the execution times of the mandatory subtask and the optional subtask of T_i to be α_i and $e_i - \alpha_i$, respectively. The algorithm then invokes the algorithms described in the previous chapter to construct a schedule for the modified task system with the minimum total w -weighted error. A formal description of the algorithm, to be called Algorithm CTWE (Constrained Total Weighted Error), is given below.

Algorithm CTWE

1. Call Algorithm MWE in the previous chapter to compute $\Upsilon_{w'}(TS)$.
2. Construct a new task system \hat{TS} from TS as follows: For each task T_i in TS , create a task \hat{T}_i in \hat{TS} . \hat{T}_i has the same characteristics as T_i except when $o_i \times w'_i > \Upsilon_{w'}(TS)$ in which case \hat{m}_i and \hat{o}_i are set to $m_i + o_i - \Upsilon_{w'}(TS)/w'_i$ and $\Upsilon_{w'}(TS)/w'_i$, respectively.
3. Call the algorithm in the previous chapter to construct a schedule for \hat{TS} with the minimum total w -weighted error.

Theorem 8.1

The time complexity of Algorithm CTWE is $O(n^2)$ for a single processor, $O(n^3 \log^2 n)$ for parallel and identical processors, and $O(m^2 n^5 \log mn)$ for uniform processors.

Proof

The running time of Algorithm CTWE is dominated in Step 1 by the running time of Algorithm MWE. The theorem follows immediately from Theorem 7.4 in the previous chapter.

8.3 Maximum w' -Weighted Error with Constraints

In this section, we will give an algorithm for finding a schedule that minimizes the maximum w' -weighted error, subject to the constraint that the total w -weighted error is minimized (see Ref. 5 for more details).

Before we give the algorithm, we need to introduce some notations and terminologies. Let TS be a task system. Let $\min\{r_i\} = t_0 < t_1 < \dots < t_p = \max\{\bar{d}_i\}$ be all the distinct values of the multiset $\{r_1, \dots, r_n, \bar{d}_1, \dots, \bar{d}_n\}$. These $p+1$ values divide the time frame into p intervals: $[t_0, t_1], [t_1, t_2], \dots, [t_{p-1}, t_p]$, denoted by u_1, u_2, \dots, u_p . The length of the interval u_j , denoted by l_j , is defined to be $t_j - t_{j-1}$. A schedule S for TS can be described by an $n \times p$ scheduling matrix SM_S such that $SM_S(i, j)$ contains the amount of processor time assigned to T_i in u_j . The scheduling matrix SM_S satisfies the following constraints: (1) $SM_S(i, j) \leq l_j$ for $1 \leq i \leq n$ and $1 \leq j \leq p$; and (2) $\sum_{i=1}^n SM_S(i, j) \leq m \times l_j$ for $1 \leq j \leq p$. Given a scheduling matrix SM_S , a schedule can easily be constructed by McNaughton's rule [6]. Throughout this chapter, we will describe a schedule by its scheduling matrix. An interval u_j is said to be *unsaturated* if $m \times l_j > \sum_{i=1}^n SM_S(i, j)$; otherwise, it is said to be *saturated*. A task T_i is said to be *available* in the time interval $[t, t']$ if $r_i \leq t \leq t' \leq \bar{d}_i$. A task T_i is said to be *fully scheduled* in u_j if $SM_S(i, j) = l_j$; otherwise, it is said to be *partially scheduled*. A task is said to be *precisely scheduled* in S if $\epsilon(T_i, S) = 0$; otherwise, it is said to be *imprecisely scheduled*.

One way to find a schedule with the minimum total w -weighted error is as follows (see Ref. 7 for more details). Let $\bar{w}_1 > \bar{w}_2 > \dots > \bar{w}_k$ be the k distinct w -weights of the task system TS . A task whose w -weight is \bar{w}_j will be called a j -task. We iterate the following process k times, from $j = 1$ until k . In the j th iteration, the optional execution time of every task, except the j -tasks, are set to zero. Then a schedule S with the minimum total error is obtained using the algorithms given in the previous chapter. (Note that the tasks with nonzero optional execution times have identical weights, namely, \bar{w}_j .) We then adjust the mandatory execution time of every j -task, say T_l , to $\alpha(T_l, S)$, before we proceed to the next iteration. At the end of the k th iteration, the mandatory execution time of every task is the amount of processor time assigned to each task in the final schedule. The basic idea of the algorithm is that it is always better (for the purpose of minimizing the total w -weighted error) to execute more of the tasks with higher w -weights than the ones with lower w -weights.

As it turns out, if the w -weights are identical, then there is a schedule that simultaneously minimizes the total w -weighted error and the maximum w' -weighted error. This property is stated below as Property P1.

Property P1

For any task system TS , if the w -weights are identical, then there is a feasible schedule S such that $\Upsilon_w(S) = \Upsilon_w(TS)$ and $\epsilon_w(S) = \epsilon_w(TS)$. In other words, $\Upsilon_{w'}^w(TS) = \Upsilon_{w'}(TS)$ and $\epsilon_{w'}^w(TS) = \epsilon_w(TS)$.

Owing to Property P1, we can modify the above algorithm for finding the minimum total w -weighted error to solve our problem. At each iteration, instead of finding a schedule with the minimum total error, we use Algorithm CTWE given in the previous section to find a schedule with the minimum total error, subject to the constraint that the maximum w' -weighted error is minimized. Since the tasks with nonzero optional execution times have identical weights at each iteration, Property P1 ensures that the schedule obtained also minimizes the maximum w' -weighted error.

Our algorithm, to be called Algorithm CMWE (Constrained Maximum Weighted Error), constructs a sequence of $k + 1$ task systems, $TS(j) = (\{T_i(j)\}, \{r_i(j)\}, \{\bar{d}_i(j)\}, \{m_i(j)\}, \{o_i(j)\}, \{w_i(j)\}, \{w'_i(j)\})$ for $0 \leq j \leq k$. Each task system consists of exactly n tasks. For each task T_i in TS , there is a task $T_i(j)$ in $TS(j)$ such that $r_i(j) = r_i$, $\bar{d}_i(j) = \bar{d}_i$, $w_i(j) = w_i$, and $w'_i(j) = w'_i$. For each $1 \leq j \leq k$, if $T_i(j)$ is a l -task, $1 \leq l \leq j$, then $m_i(j)$ is set to the total processor time assigned to T_i in the final schedule for TS and $o_i(j)$ is set to zero. Otherwise, they are set to m_i and 0, respectively. Finally, $m_i(k)$, $1 \leq i \leq n$, gives the total processor time assigned to T_i in the final schedule for TS .

Initially, $TS(0)$ is set to be the same as TS , except that $o_i(0)$ is set to zero for each $1 \leq i \leq n$. Algorithm CMWE proceeds in k phases. In the j th phase, $1 \leq j \leq k$, the algorithm sets $TS(j) = (\{T_i(j)\}, \{r_i(j)\}, \{\bar{d}_i(j)\}, \{m_i(j)\}, \{o_i(j)\}, \{w_i(j)\}, \{w'_i(j)\})$ to be the same as $TS(j - 1)$, except that $o_i(j)$ is set to o_i for each j -task $T_i(j)$. Algorithm CTWE given in the previous section is then invoked to generate a schedule $S(j)$ for $TS(j)$. The variable λ for storing the total w -weighted error of TS (which initially is set to zero) is increased by the total w -weighted error of $S(j)$. Also, the variable η for storing the maximum w' -weighted error of TS (which initially is set to zero) is updated to be the larger of $\Upsilon_{w'}(S(j))$ and the current value of η . Finally, for each j -task $T_i(j)$, the algorithm sets $m_i(j)$ to $\alpha(T_i(j), S(j))$ and $o_i(j)$ to zero. The algorithm then repeats the above process in the next phase. (Notice that the mandatory and optional execution times of a j -task are different at the beginning of the j th phase than at the end of the j th phase.) A formal description of Algorithm CMWE is given below.

Algorithm CMWE

1. Sort all tasks in nonincreasing order of the w -weights and let $\bar{w}_1 > \bar{w}_2 > \dots > \bar{w}_k$ be the k distinct w -weights in TS .
2. Initialize λ and η to zero; initialize $TS(0)$ to TS , except that $o_i(0)$ is set to zero for each $1 \leq i \leq n$.
3. For $j = 1, \dots, k$ do:
 - Let $TS(j)$ be the same as $TS(j - 1)$, except that $o_i(j)$ is set to o_i for each j -task $T_i(j)$.
 - Invoke Algorithm CTWE given in the previous section to construct a schedule $S(j)$ for $TS(j)$.
 - $\lambda \leftarrow \lambda + \epsilon_w(S(j))$.
 - If $\eta < \Upsilon_{w'}(S(j))$, then $\eta \leftarrow \Upsilon_{w'}(S(j))$.
 - For each j -task $T_i(j)$, let $m_i(j)$ be $\alpha(T_i(j), S(j))$ and $o_i(j)$ be zero.
4. Output $S(k)$ and stop.

Figure 8.1 shows a task system with two distinct w -weights. The task system $TS(1)$ constructed in the first phase is shown in Figure 8.2a. Figure 8.2b depicts the schedule $S(1)$ for $TS(1)$ obtained from Algorithm CTWE given in the previous section. The task system $TS(2)$ constructed in the second phase is shown in Figure 8.3a. Finally, Figure 8.3b depicts the schedule $S(2)$ for $TS(2)$, which is also the final schedule produced by Algorithm CMWE.

Theorem 8.2

Algorithm CMWE constructs a schedule that minimizes the maximum w' -weighted error, subject to the constraint that the total w -weighted error is minimum. Its time complexity is $O(kn^2)$ for a single processor,

$T_i(0)$	$T_1(0)$	$T_2(0)$	$T_3(0)$	$T_4(0)$	$T_5(0)$	$T_6(0)$	$T_7(0)$	$T_8(0)$
$r_i(0)$	0	0	0	6	6	2	6	6
$\bar{d}_i(0)$	6	6	5	9	9	10	9	10
$m_i(0)$	1	2	0	1	1	2	0	1
$o_i(0)$	7	6	4	5	7	5	5	3
$w_i(0)$	1	1	2	2	1	1	2	2
$w'_i(0)$	4	2	1	1	2	2	4	2

FIGURE 8.1 An example task system $TS(0)$.

(a)

$T_i(1)$	$T_1(1)$	$T_2(1)$	$T_3(1)$	$T_4(1)$	$T_5(1)$	$T_6(1)$	$T_7(1)$	$T_8(1)$
$r_i(1)$	0	0	0	6	6	2	6	6
$\bar{d}_i(1)$	6	6	5	9	9	10	9	10
$m_i(1)$	1	2	0	1	1	2	0	1
$o_i(1)$	0	0	4	5	0	0	5	3
$w_i(1)$	1	1	2	2	1	1	2	2
$w'_i(1)$	4	2	1	1	2	2	4	1

(b)

0	1	2	5	6	7	8	9	10
$T_1(1)$	$T_3(1)$			$T_6(1)$	$T_7(1)$			$T_8(1)$
$T_2(1)$						$T_4(1)$	$T_5(1)$	$T_8(1)$

FIGURE 8.2 (a) The task system $TS(1)$ and (b) the schedule $S(1)$.

(a)

$T_i(2)$	$T_1(2)$	$T_2(2)$	$T_3(2)$	$T_4(2)$	$T_5(2)$	$T_6(2)$	$T_7(2)$	$T_8(2)$
$r_i(2)$	0	0	0	6	6	2	6	6
$\bar{d}_i(2)$	6	6	5	9	9	10	9	10
$m_i(2)$	1	2	4	4	1	2	4	2
$o_i(2)$	7	6	0	0	7	5	0	0
$w_i(2)$	1	1	2	2	1	1	2	2
$w'_i(2)$	4	2	1	1	2	2	4	1

(b)

0	1	2	5	6	7	8	9	10
$T_1(2)$	$T_3(2)$			$T_6(2)$	$T_7(2)$			$T_8(2)$
$T_2(2)$		$T_1(2)$			$T_4(2)$	$T_5(2)$	$T_8(2)$	

FIGURE 8.3 (a) The task system $TS(2)$ and (b) the schedule $S(2)$.

$O(kn^3 \log^2 n)$ for parallel and identical processors, and $O(km^2 n^5 \log mn)$ for uniform processors, where k is the number of distinct w -weights.

Proof

The correctness of Algorithm CMWE is due to Property P1 whose proof is omitted (see Ref. 5 for more details). Algorithm CMWE calls Algorithm CTWE k times. Its time complexity follows immediately from Theorem 8.1.

8.4 0/1-Constraints

In this section, we will consider scheduling imprecise computation tasks with 0/1-constraint on a single processor. We will give two approximation algorithms. The first one is for minimizing the total error and the second one is for minimizing the number of imprecisely scheduled tasks.

We now define notations that will be used throughout this section. Let TS be a set of n imprecise computation tasks. We use $PO(TS)$ to denote the set of tasks with positive optional execution time. A task T_i is *eligible* in a given interval $[t', t'']$ if $r_i \leq t' \leq t'' \leq \bar{d}_i$. We use $\hat{\epsilon}(TS)$ to denote the minimum total error of TS under the 0/1-constraint; note that $\epsilon(TS)$ denotes the minimum total error of TS without any constraint. If H is a scheduling algorithm, we let (1) $E_H(TS)$ denote the set of precisely scheduled tasks produced by H ; i.e., $E_H(TS) = \{T_i : o_i > 0 \text{ and } T_i \text{ is precisely scheduled by } H\}$ and (2) $\tilde{E}_H(TS)$ denote the set of imprecisely scheduled tasks produced by H ; i.e., $\tilde{E}_H(TS) = PO(TS) - E_H(TS)$.

8.4.1 Feasibility Test

In this section, we will give a fast algorithm to test if a set of tasks is feasible on a single processor. Each task T_i has an execution time e_i , release time r_i , and deadline \bar{d}_i . The Boolean function Feasible will decide if the set of tasks is feasible on a single processor. Let $0 = \min\{r_i\} = u_0 < u_1 < \dots < u_p = \max\{\bar{d}_i\}$ be the $p+1$ distinct integers obtained from the multiset $\{r_1, \dots, r_n, \bar{d}_1, \dots, \bar{d}_n\}$. These $p+1$ integers divide the time frame into p segments: $[u_0, u_1]$, $[u_1, u_2]$, \dots , $[u_{p-1}, u_p]$. The algorithm assumes that the tasks have been sorted in descending order of release times. Below is a formal description of the function.

Boolean Function Feasible (TS)

- (1) For $i = 1, \dots, p$ do: $l_i \leftarrow u_i - u_{i-1}$.
- (2) For $i = 1, \dots, n$ do:
 - Find a satisfying $u_a = \bar{d}_i$ and b satisfying $u_b = r_i$.
 - For $j = a, a-1, \dots, b+1$ do:
 - $\delta \leftarrow \min\{l_j, e_i\}$.
 - $l_j \leftarrow l_j - \delta, e_i \leftarrow e_i - \delta$.
 - If $e_i = 0$ then “break”.
 - If $e_i > 0$ then return “False”.
- (3) Return “True”.

The algorithm schedules tasks in descending order of their release times. When a task is scheduled, it is assigned from the latest segment $[u_{a-1}, u_a]$ in which it can be scheduled until the earliest segment $[u_b, u_{b+1}]$. Let us examine the time complexity of the algorithm. The time it takes to sort the tasks in descending order of their release times as well as obtaining the set $\{u_0, u_1, \dots, u_p\}$ is $O(n \log n)$. Step 1 of the algorithm takes linear time and a straightforward implementation of Step 2 takes $O(n^2)$ time. Thus, it appears that the running time of the algorithm is $O(n^2)$. However, observe that whenever a segment is scheduled, either all the units of a task are scheduled or the segment is saturated, or both. Hence, at most $O(n)$ segments have positive values. Thus, if we can avoid scanning those segments that have zero values,

then Step 2 takes only linear time. As it turns out, this can be done by the special UNION-FIND algorithm owing to Gabow and Tarjan [8].

As we will see later, both of our approximation algorithms make n calls to function Feasible with the same set of tasks but different values of execution times. Since each call takes linear time, the overall running time of our approximation algorithm is $O(n^2)$.

8.4.2 Total Error

In this section, we will give a fast approximation algorithm to minimize the total error. The algorithm works as follows. Let TS be a set of n tasks and let $n' = |PO(TS)|$. First, the tasks are sorted in descending order of their optional execution times. Then, the set of precisely scheduled tasks is initialized to the empty set and the following process is iterated n' times. At the i th iteration, we set the execution times of T_i and all the precisely scheduled tasks to its mandatory plus optional execution times, and the execution times of all other tasks to its mandatory execution times only. We then test if this set of tasks is feasible. If it is feasible, we include T_i into the set of precisely scheduled tasks; otherwise, T_i will not be included. A formal description of the algorithm is given below.

- (1) Sort the tasks in descending order of their optional execution times; i.e., $o_i \geq o_{i+1}$ for $1 \leq i < n$.
 $E_L(TS) \leftarrow \emptyset$.
- (2) For $i = 1, 2, \dots, n'$ do:
 Create a set of n tasks TS' with release times, deadlines, and execution times as follows: For each task T_j in $E_L(TS) \cup \{T_i\}$, create a task T'_j with the same release time and deadline as T_j , and execution time $e'_j = m_j + o_j$. For every other task T_k , create a task T'_k with the same release time and deadline as T_k , and execution time $e'_k = m_k$.
 If Feasible(TS') = "True", then $E_L(TS) \leftarrow E_L(TS) \cup \{T_i\}$.

The time complexity of the algorithm is $O(n^2)$, since the function Feasible is called n times and each call takes linear time. It is interesting to observe that the worst-case performance of the algorithm is unbounded if the tasks were sorted in ascending order, rather than descending order, of their optional execution times. Consider the two tasks T_1 and T_2 with $r_1 = 0$, $\bar{d}_1 = x$, $m_1 = 0$, $o_1 = x$, $r_2 = 0$, $\bar{d}_2 = 1$, $m_2 = 0$, and $o_2 = 1$. With the new ordering, T_2 will be scheduled precisely. However, the optimal solution will schedule T_1 precisely. Thus, the ratio becomes x , which can be made arbitrarily large. However, if the tasks were sorted in descending order of their optional execution times, the algorithm has a worst-case bound at most 3, as the following theorem shows.

Theorem 8.3

For any task system TS , we have $\epsilon_L(TS) \leq 3\hat{\epsilon}(TS)$, where $\epsilon_L(TS)$ is the total error produced by Algorithm Largest-Optional-Execution-Time-First. Moreover, the bound can be achieved asymptotically.

We first give a task system showing that the bound can be achieved asymptotically. Consider four tasks T_1, T_2, T_3 , and T_4 with $r_1 = x - \delta$, $\bar{d}_1 = 2x - \delta$, $r_2 = 0$, $\bar{d}_2 = x$, $r_3 = 2x - 2\delta$, $\bar{d}_3 = 3x - 2\delta$, $r_4 = x$, $\bar{d}_4 = 2x - 2\delta$, $m_1 = m_2 = m_3 = m_4 = 0$, $o_1 = o_2 = o_3 = x$, and $o_4 = x - 2\delta$. It is clear that Algorithm Largest-Optional-Execution-Time-First will schedule T_1 precisely, while the optimal solution will schedule T_2, T_3 , and T_4 precisely. Thus, the ratio approaches 3 as δ approaches 0.

We will prove Theorem 8.3 in the remainder of this section. First, we will state an assertion whose proof will be omitted; it can be found in Ref. 3. Call a task *improper* if it is precisely scheduled by Algorithm Largest-Optional-Execution-Time-First, but not by the optimal algorithm.

Assertion A1

Let T_s be an improper task in the task system TS and let \tilde{TS} be obtained from TS by setting the optional execution time of T_s to 0. Then we have $\epsilon_L(\tilde{TS}) > \epsilon_L(TS) - 3o_s$.

With Assertion A1, we will prove the bound by contradiction. Let TS be the smallest task system, in terms of n' , that violates the bound. We will characterize the nature of TS in the next two lemmas.

Lemma 8.1

$E_L(TS) \cap E_O(TS) = \emptyset$ and $E_L(TS) \cup E_O(TS) = PO(TS)$.

Proof

If $E_L(TS) \cap E_O(TS) \neq \emptyset$, let $T_i \in E_L(TS) \cap E_O(TS)$. Consider the task system \tilde{TS} obtained from TS by setting the mandatory and optional execution times of T_i to $m_i + o_i$ and 0, respectively. It is clear that $|PO(\tilde{TS})| < |PO(TS)|$, $\epsilon_L(\tilde{TS}) = \epsilon_L(TS)$, and $\hat{e}(\tilde{TS}) = \hat{e}(TS)$. Thus, \tilde{TS} is a smaller task system violating the bound, contradicting the assumption that TS is the smallest.

If $E_L(TS) \cup E_O(TS) \neq PO(TS)$, let T_i be a task such that $o_i > 0$ and $T_i \notin E_L(TS) \cup E_O(TS)$. Consider the task system \tilde{TS} obtained from TS by setting the optional execution time of T_i to be 0. Clearly, $|PO(\tilde{TS})| < |PO(TS)|$, $\epsilon_L(\tilde{TS}) = \epsilon_L(TS) - o_i$, and $\hat{e}(\tilde{TS}) = \hat{e}(TS) - o_i$. Thus, $\epsilon_L(\tilde{TS}) = \epsilon_L(TS) - o_i > 3\hat{e}(TS) - o_i = 3\hat{e}(\tilde{TS}) + 2o_i > 3\hat{e}(\tilde{TS})$, and hence \tilde{TS} is a smaller task system violating the bound.

Lemma 8.2

$E_L(TS) = \emptyset$.

Proof

If $E_L(TS) \neq \emptyset$, let $E_L(TS) = \{T_{i_1}, \dots, T_{i_m}\}$, where $m \geq 1$. By Lemma 8.1, each task in $E_L(TS)$ is an improper-task. Consider the task system \tilde{TS} obtained from TS by setting the optional execution time of T_{i_m} to 0. Clearly, $|PO(\tilde{TS})| < |PO(TS)|$. By Assertion A1, we have

$$\epsilon_L(\tilde{TS}) > \epsilon_L(TS) - 3o_{i_m} \quad (8.1)$$

Since $\epsilon_L(TS) > 3\hat{e}(TS)$, we have

$$\epsilon_L(\tilde{TS}) > 3(\hat{e}(TS) - o_{i_m}) \quad (8.2)$$

Since $T_{i_m} \notin E_O(TS)$, we have

$$\hat{e}(\tilde{TS}) = \hat{e}(TS) - o_{i_m} \quad (8.3)$$

From Equations 8.2 and 8.3, we have $\epsilon_L(\tilde{TS}) > 3\hat{e}(\tilde{TS})$, contradicting the assumption that TS is the smallest task system violating the bound.

We can now prove Theorem 8.3. Lemma 8.2 implies that Algorithm Largest-Optional-Execution-Time-First cannot schedule *any* tasks precisely. However, Lemma 8.1 implies that the optimal algorithm schedules *every* task precisely. These two facts lead to impossibility.

8.4.3 Number of Imprecisely Scheduled Tasks

In this section, we will give a fast approximation algorithm for minimizing the number of imprecisely scheduled tasks. The algorithm, to be called Algorithm Smallest-Optional-Execution-Time-First, works exactly like Algorithm Largest-Optional-Execution-Time-First, except that tasks are sorted in ascending order of their optional execution times. A formal description of the algorithm is given below.

Algorithm Smallest-Optional-Execution-Time-First

- (1) Sort the tasks in $PO(TS)$ from 1 to n' such that $o_i \leq o_{i+1}$ for $1 \leq i < n'$. Index the remaining tasks from $n' + 1$ to n . $E_S(TS) \leftarrow \emptyset$.

(2) For $i = 1, 2, \dots, n'$ do:

Create a set of n tasks TS' with release times, deadlines, and execution times as follows: For each task T_j in $E_S(TS) \cup \{T_i\}$, create a task T'_j with the same release time and deadline as T_j , and execution time $e'_j = m_j + o_j$. For every other task T_k , create a task T'_k with the same release time and deadline as T_k , and execution time $e'_k = m_k$.

If $\text{Feasible}(TS') = \text{"True"}$, then $E_S(TS) \leftarrow E_S(TS) \cup \{T_i\}$.

Again, the time complexity of Algorithm Smallest-Optional-Execution-Time-First is $O(n^2)$. In the last section, we showed that if Algorithm Smallest-Optional-Execution-Time-First were used to minimize the total error, then it would give an unbounded performance. As it turns out, if Algorithm Largest-Optional-Execution-Time-First were used to minimize the number of imprecisely scheduled tasks, then it would also give an unbounded performance. Consider the task system consisting of $n + 1$ tasks: For $1 \leq i \leq n$, $r_i = (i - 1)x$, $\bar{d}_i = ix$, $m_i = 0$, and $o_i = x$; $r_{n+1} = 0$, $\bar{d}_{n+1} = nx$, $m_{n+1} = 0$, $o_{n+1} = nx$, where $x > 0$. It is clear that Algorithm Largest-Optional-Execution-Time-First schedules T_{n+1} precisely, while the optimal algorithm schedules T_1, T_2, \dots, T_n precisely. Thus, the ratio can be made arbitrarily large by taking n large enough. However, Algorithm Smallest-Optional-Execution-Time-First gives a much better performance as the next theorem shows.

Theorem 8.4

For any task system TS , we have $|\tilde{E}_S(TS)| \leq 2|\tilde{E}_O(TS)|$. Moreover, the bound is tight.

We first give a task system showing that the bound is tight. Consider the three tasks T_1, T_2 , and T_3 with $r_1 = x - \delta$, $\bar{d}_1 = 2x - \delta$, $r_2 = 0$, $\bar{d}_2 = x$, $r_3 = 2(x - \delta)$, $\bar{d}_3 = 3x - 2\delta$, $m_1 = m_2 = m_3 = 0$, $o_1 = o_2 = o_3 = x$, where $x > \delta$. Algorithm Smallest-Optional-Execution-Time-First schedules T_1 precisely, while the optimal algorithm schedules T_2 and T_3 precisely. Thus, the ratio is 2. In the following we will prove the upper bound. First, we will state an assertion whose proof will be omitted; it can be found in Ref. 3. Call a task *improper* if it is precisely scheduled by Algorithm Smallest-Optional-Execution-Time-First, but not by the optimal algorithm.

Assertion A2

Let T_s be an improper task in the task system TS and let \tilde{TS} be obtained from TS by setting the optional execution time of T_s to 0. Then we have $|\tilde{E}_S(\tilde{TS})| \geq |\tilde{E}_S(TS)| - 2$.

With Assertion A2, we will prove the upper bound by contradiction. Let TS be the smallest task system, in terms of n' , that violates the bound. The next two lemmas, which are counterparts of Lemmas 8.1 and 8.2, characterize the nature of TS .

Lemma 8.3

$E_S(TS) \cap E_O(TS) = \emptyset$ and $E_S(TS) \cup E_O(TS) = PO(TS)$.

Proof

If $E_S(TS) \cap E_O(TS) \neq \emptyset$, let $T_i \in E_S(TS) \cap E_O(TS)$. Consider the task system \tilde{TS} obtained from TS by setting the mandatory and optional execution times of T_i to $m_i + o_i$ and 0, respectively. It is clear that $|PO(\tilde{TS})| < |PO(TS)|$, $|\tilde{E}_S(\tilde{TS})| = |\tilde{E}_S(TS)|$, and $|\tilde{E}_O(\tilde{TS})| = |\tilde{E}_O(TS)|$. Thus, \tilde{TS} is a smaller task system violating the bound, contradicting the assumption that TS is the smallest.

If $E_S(TS) \cup E_O(TS) \neq PO(TS)$, let T_i be a task such that $o_i > 0$ and $T_i \notin E_S(TS) \cup E_O(TS)$. Consider the task system \tilde{TS} obtained from TS by setting the optional execution time of T_i to 0. Clearly, $|PO(\tilde{TS})| < |PO(TS)|$, $|\tilde{E}_S(\tilde{TS})| = |\tilde{E}_S(TS)| - 1$, and $|\tilde{E}_O(\tilde{TS})| = |\tilde{E}_O(TS)| - 1$. Thus, $|\tilde{E}_S(\tilde{TS})| = |\tilde{E}_S(TS)| - 1 > 2|\tilde{E}_O(TS)| - 1 = 2|\tilde{E}_O(\tilde{TS})| + 1 > 2|\tilde{E}_O(\tilde{TS})|$, and hence \tilde{TS} is a smaller task system violating the bound.

Lemma 8.4

$E_S(TS) = \emptyset$.

Proof

If $E_S(TS) \neq \emptyset$, let $E_S(TS) = \{T_{i_1}, \dots, T_{i_m}\}$, where $m \geq 1$. By Lemma 8.3, each task in $E_S(TS)$ is an inappropriate task in TS . Consider the task system \tilde{TS} obtained from TS by setting the optional execution time of T_{i_m} to 0. Clearly, $|PO(\tilde{TS})| < |PO(TS)|$. By Assertion A2, we have

$$|\tilde{E}_S(\tilde{TS})| \geq |\tilde{E}_S(TS)| - 2 \quad (8.4)$$

Since $|\tilde{E}_S(TS)| > 2|\tilde{E}_O(TS)|$, we have

$$|\tilde{E}_S(\tilde{TS})| > 2|\tilde{E}_O(TS)| - 2 \quad (8.5)$$

Since $|\tilde{E}_O(\tilde{TS})| = |\tilde{E}_O(TS)| - 1$, we have

$$|\tilde{E}_S(\tilde{TS})| > 2|\tilde{E}_O(\tilde{TS})| \quad (8.6)$$

contradicting our assumption that TS is the smallest task system violating the bound.

Lemma 8.4 implies that Algorithm Smallest-Optional-Execution-Time-First cannot schedule *any* tasks in $PO(TS)$ precisely, while Lemma 8.3 implies that the optimal algorithm schedules *every* task in $PO(TS)$ precisely. These two facts lead to an impossibility, which proves Theorem 8.4.

While Theorem 8.4 gives a relationship between $|\tilde{E}_S(TS)|$ and $|\tilde{E}_O(TS)|$, it does not give a meaningful relationship between $|E_S(TS)|$ and $|E_O(TS)|$. The next theorem shows that they are also related by the same multiplicative factor.

Theorem 8.5

For any task system TS , $|E_O(TS)| \leq 2|E_S(TS)|$. Moreover, the bound is tight.

Proof

The task system TS given in the proof of Theorem 8.4 also shows that $|E_O(TS)| = 2|E_S(TS)|$. The upper bound is proved by contradiction. Let TS be the smallest task system, in terms of $|PO(TS)|$, that violates the bound. It is easy to verify that Lemma 8.3 also holds for TS . Thus, $E_S(TS) = \tilde{E}_O(TS)$ and $E_O(TS) = \tilde{E}_S(TS)$. Hence, $|E_O(TS)| = |\tilde{E}_S(TS)| \leq 2|\tilde{E}_O(TS)| = 2|E_S(TS)|$, contradicting our assumption that TS violates the bound.

8.5 Conclusions

In this chapter, we have given an algorithm for finding a schedule that minimizes the total *w-weighted* error, subject to the constraint that the maximum *w'-weighted* error is minimum. As well, we gave an algorithm for finding a schedule that minimizes the maximum *w'-weighted* error, subject to the constraint that the total *w-weighted* error is minimum.

We have also considered the problem of preemptively scheduling a set of imprecise computation tasks on a single processor with the added constraint that each optional subtask is either fully executed or not executed at all. We gave an $O(n^2)$ -time approximation algorithm, Algorithm Largest-Optional-Execution-Time-First, for minimizing the total error, and showed that it has a tight bound of 3. We also gave an $O(n^2)$ -time approximation algorithm, Algorithm Smallest-Optional-Execution-Time-First, for minimizing the number of imprecisely scheduled tasks, and showed that it has a tight bound of 2. Interestingly, the number

of precisely scheduled tasks in an optimal schedule is also bounded above by two times the number of precisely scheduled tasks in Algorithm Smallest-Optional-Execution-Time-First.

For future research, it will be interesting to see if there are any fast approximation algorithms for the two problems with better performance bounds than those of Algorithm Largest-Optional-Execution-Time-First and Algorithm Smallest-Optional-Execution-Time-First. Also, the case of parallel and identical processors and the case of uniform processors are totally unexplored.

References

1. E. L. Lawler, A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs, *Annals of Operations Research*, 26, 125–133, 1990.
2. W.-K. Shih, J. W. S. Liu and J.-Y. Chung, Algorithms for scheduling imprecise computations with timing constraints, *SIAM Journal on Computing*, 20, 537–552, 1991.
3. K. I-J. Ho, J. Y-T. Leung and W.-D. Wei, Scheduling imprecise computation tasks with 0/1-constraint, *Discrete Applied Mathematics*, 78, 117–132, 1997.
4. K. I-J. Ho, J. Y-T. Leung, and W.-D. Wei, Minimizing maximum weighted error for imprecise computation tasks, *Journal of Algorithms*, 16, 431–452, 1994.
5. K. I-J. Ho and J. Y-T. Leung, A dual criteria preemptive scheduling problem for minimax error of imprecise computation tasks, *International Journal of Foundations of Computer Science*, 15, 717–731, 2004.
6. R. McNaughton, Scheduling with deadlines and loss functions, *Management Science*, 6, 1–12, 1959.
7. J. Y-T. Leung, V. K. M. Yu and W.-D. Wei, Minimizing the weighted number of tardy task units, *Discrete Applied Mathematics*, 51, 307–316, 1994.
8. H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *Journal of Computer and System Science*, 30, 209–221, 1985.

Stochastic Analysis of Priority-Driven Periodic Real-Time Systems*

José Luis Díaz
Universidad de Oviedo

Kanghee Kim
Seoul National University

José María López
Universidad de Oviedo

Lucia Lo Bello
Università di Catania

Daniel F. García
Universidad de Oviedo

Chang-Gun Lee
Ohio State University

Sang Lyul Min
Seoul National University

Orazio Mirabella
Università di Catania

9.1	Introduction	9-1
9.2	Related Work	9-2
9.3	System Model	9-3
9.4	Stochastic Analysis Framework	9-4
	Overview • Backlog Analysis Algorithm • Interference Analysis Algorithm • Backlog Dependency Tree • Extension to Dynamic-Priority and Fixed-Priority Systems	
9.5	Steady-State Backlog Analysis	9-12
	Existence of the Stationary Backlog Distribution • Exact Solution • Approximated Solutions • Safe Truncation of the Exact Solution	
9.6	Computational Complexity	9-15
	Complexity of the Backlog and Interference Analysis • Complexity of the Steady-State Backlog Analysis	
9.7	Experimental Results	9-17
	Comparison between the Solution Methods • Complexity Evaluation of the Backlog and Interference Analysis	
9.8	Conclusions and Future Work	9-22

9.1 Introduction

Most recent research on hard real-time systems has used the periodic task model [1] in analyzing the schedulability of a given task set in which tasks are released periodically. Based on this periodic task model, various schedulability analysis methods for priority-driven systems have been developed to provide a deterministic guarantee that all the instances, called *jobs*, of every task in the system meet their deadlines, assuming that every job in a task requires its worst-case execution time [1–3].

Although this deterministic timing guarantee is needed in hard real-time systems, it is too stringent for soft real-time applications that only require a probabilistic guarantee that the deadline miss ratio of a task is below a given threshold. For soft real-time applications, we need to relax the assumption that every instance of a task requires the worst-case execution time to improve the system utilization. This is also

*An earlier version of the manuscript was published in the *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1460–1466, 2005.

needed for probabilistic hard real-time systems [4] in which a probabilistic guarantee close to 0% suffices, i.e., the overall deadline miss ratio (DMR) of the system should be below a hardware failure ratio.

Progress has recently been made in the analysis of real-time systems under the stochastic assumption that jobs from a task require variable execution times. Research in this area can be categorized into two groups depending on the approach used to facilitate the analysis. The methods in the first group introduce a worst-case assumption to simplify the analysis (e.g., the critical instant assumption in probabilistic time demand analysis (PTDA) [5] and stochastic time demand analysis [STDA] [6,7]) or a restrictive assumption (e.g., the heavy traffic condition in the real-time queueing theory [8,9]). Those in the second group, however, assume a special scheduling model that provides isolation between tasks so that each task can be analyzed independently of the other tasks in the system (e.g., the reservation-based system addressed in Ref. 10 and statistical rate monotonic scheduling [11]).

In this chapter, we describe a stochastic analysis framework that does not introduce any worst-case or restrictive assumptions into the analysis and is applicable to general priority-driven real-time systems. The proposed framework builds upon STDA in that the techniques used in the framework to compute the response time distributions of tasks are largely borrowed from the STDA. However, unlike the STDA, which focuses on particular execution scenarios starting at a critical instant, the proposed framework considers all possible execution scenarios to obtain the exact response time distributions of the tasks. Moreover, while the STDA addresses only fixed-priority systems such as rate monotonic (RM) [1] and deadline monotonic (DM) [12], our framework extends to dynamic-priority systems such as earliest deadline first (EDF) [1]. The contributions of the framework can be summarized as follows:

- The framework gives the *exact* response time distributions of the tasks. It assumes neither a particular execution scenario of the tasks, such as critical instants, nor a particular system condition, such as heavy traffic, to obtain accurate analysis results considering all possible execution scenarios for a wide range of system conditions.
- The framework provides a *unified* approach to addressing general priority-driven systems, including both fixed-priority systems, such as RM and DM, and dynamic-priority systems, such as EDF. We neither modify the conventional rules of priority-driven scheduling nor introduce other additional scheduling rules, such as reservation scheduling, to analyze the priority-driven system as it is.

In our framework, to consider all possible execution scenarios in the system, we analyze a whole hyperperiod of the given task set (which is defined as a period whose length is equal to the least common multiple of the periods of all the tasks). In particular, to handle even cases where one hyperperiod affects the next hyperperiod, which occurs when the maximum utilization of the system is greater than 1, we take the approach of modeling the system as a Markov process over an infinite sequence of hyperperiods. This modeling leads us to solve an infinite number of linear equations, so we present three different methods to solve it: one method gives the exact solution, and the other two give approximated solutions. We compare all these methods in terms of analysis complexity and accuracy through experiments. It should be noted that our framework subsumes the conventional deterministic analysis in the sense that, by modeling the worst-case execution times as single-valued distributions, it always produces the same result as the deterministic analysis on whether a task set is schedulable or not.

The rest of the chapter is organized as follows. In Section 9.2, the related work is described in detail. In Section 9.3, the system model is explained. Sections 9.4 and 9.5 describe the stochastic analysis framework including the exact and the approximation methods. In Section 9.6, the complexity of the methods is analyzed, and in Section 9.7, a comparison between the solutions obtained by the methods is given, together with other analysis methods proposed in literature. Finally, in Section 9.8, we conclude with directions for future research.

9.2 Related Work

Several studies have addressed the variability of task execution times in analyzing the schedulability of a given task set. Research in this area can be categorized into two groups depending on the approach

taken to make the analysis possible. The methods in the first group [5–9,13,14] introduce a worst-case or restrictive assumption to simplify the analysis. Those in the second group [10,11] assume a special scheduling model that provides isolation between tasks so that each task can be analyzed independently of other tasks in the system.

Examples of analysis methods in the first group include PTDA [5] and STDA [6,7], both of which target fixed-priority systems with tasks having arbitrary execution time distributions. PTDA is a stochastic extension of the time demand analysis [2] and can only deal with tasks with relative deadlines smaller than or equal to the periods. In contrast, STDA, which is a stochastic extension of general time demand analysis [3], can handle tasks with relative deadlines greater than the periods. Like the original time demand analysis, both methods assume the critical instant where the task being analyzed and all the higher priority tasks are released or arrive at the same time. Although this worst-case assumption simplifies the analysis, it only results in an upper bound on the deadline miss probability (DMP), the conservativeness of which depends on the number of tasks and the average utilization of the system. Moreover, both analyses are valid only when the maximum utilization of the system does not exceed 1.

Other examples of analysis methods in the first group are the methods proposed by Manolache et al. [13], which addresses only uniprocessor systems, and the one proposed by Leulseged and Nissanke [14], which extends to multiprocessor systems. These methods, like the one presented in this chapter, cover general priority-driven systems including both fixed-priority and dynamic-priority systems. However, to limit the scope of the analysis to a single hyperperiod, both methods assume that the relative deadlines of tasks are shorter than or equal to their periods and that all the jobs that miss the deadlines are dropped. Moreover, in Ref. 13, all the tasks are assumed to be nonpreemptable to simplify the analysis.

The first group also includes the real-time queueing theory [8,9], which extends the classical queueing theory to real-time systems. This analysis method is flexible, in that it is not limited to a particular scheduling algorithm and can be extended to real-time queueing networks. However, it is only applicable to systems where the heavy traffic assumption (i.e., the average system utilization is close to 1) holds. Moreover, it only considers one class of tasks such that the interarrival times and execution times are identically distributed.

Stochastic analysis methods in the second group include the one proposed by Abeni and Buttazzo [10], and the method with statistical rate monotonic scheduling (SRMS) [11]. Both assume reservation-based scheduling algorithms so that the analysis can be performed as if each task had a dedicated (virtual) processor. That is, each task is provided with a guaranteed budget of processor time in every period [10] or superperiod (the period of the next low-priority task, which is assumed to be an integer multiple of the period of the task in SRMS) [11]. So, the DMP of a task can be analyzed independently of the other tasks, assuming the guaranteed budget. However, these stochastic analysis methods are not applicable to general priority-driven systems due to the modification of the original priority-driven scheduling rules or the use of reservation-based scheduling algorithms.

9.3 System Model

We assume a uniprocessor system that consists of n -independent periodic tasks $S = \{\tau_1, \dots, \tau_n\}$, each task τ_i ($1 \leq i \leq n$) being modeled by the tuple (T_i, Φ_i, C_i, D_i) , where T_i is the period of the task, Φ_i its initial phase, C_i its execution time, and D_i its relative deadline. The execution time is a discrete random variable* with a given probability mass function (PMF), denoted by $f_{C_i}(\cdot)$, where $f_{C_i}(c) = \mathbb{P}\{C_i = c\}$. The execution time PMF can be given by a measurement-based analysis such as automatic tracing analysis [15] and stored as a finite vector, whose indices are possible values of the execution time and the stored values are their probabilities. The indices range from a minimum execution time C_i^{\min} to a maximum execution time

*Throughout this chapter, we use a calligraphic typeface to denote random variables, e.g., \mathcal{C} , \mathcal{W} , and \mathcal{R} , and a noncalligraphic typeface to denote deterministic variables, e.g., C , W , and R .

C_i^{\max} . Without loss of generality, the phase Φ_i of each task τ_i is assumed to be smaller than T_i . The relative deadline D_i can be smaller than, equal to, or greater than T_i .

Associated with the task set, the system utilization is defined as the sum of the utilizations of all the tasks. Owing to the variability of task execution times, the minimum U^{\min} , the maximum U^{\max} , and the average system utilization \bar{U} are defined as $\sum_{i=1}^n C_i^{\min}/T_i$, $\sum_{i=1}^n C_i^{\max}/T_i$, and $\sum_{i=1}^n \bar{C}_i/T_i$, respectively. In addition, a hyperperiod of the task set is defined as a period of length T_H , which is equal to the least common multiple of the task periods, i.e., $T_H = \text{lcm}_{1 \leq i \leq n}\{T_i\}$.

Each task gives rise to an infinite sequence of jobs whose release times are deterministic. If we denote the j th job of task τ_i by $J_{i,j}$, its release time $\lambda_{i,j}$ is equal to $\Phi_i + (j-1)T_i$. Each job $J_{i,j}$ requires an execution time, which is described by a random variable following the given PMF $f_{C_i}(\cdot)$ of the task τ_i and is assumed to be independent of other jobs of the same task and those of other tasks. However, throughout the chapter we use a single index j for the job subscript, since the task that the job belongs to is not important in describing our analysis framework. However, we sometimes additionally use a superscript for the job notation to express the hyperperiod that the job belongs to. That is, we use $J_j^{(k)}$ to refer to the j th job in the k th hyperperiod.

The scheduling model we assume is a general priority-driven preemptive one that covers both fixed-priority systems, such as RM and DM, and dynamic-priority systems, such as EDF. The only limitation is that once a priority is assigned to a job it never changes, which is called a job-level fixed-priority model [16]. According to the priority, all the jobs are scheduled in such a way that, at any time, the job with the highest priority is always served first. If two or more jobs with the same priority are ready at the same time, the one that arrived first is scheduled first. We denote the priority of job J_j by a priority value p_j . Note that a higher priority value means a lower priority.

The response time for each job J_j is represented by \mathcal{R}_j and its PMF by $f_{\mathcal{R}_j}(r) = \mathbb{P}\{\mathcal{R}_j = r\}$. From the job response time PMFs, we can obtain the response time PMF for any task by averaging those of all the jobs belonging to the task. The task response time PMFs provide the analyst with significant information about the stochastic behavior of the system. In particular, the PMFs can be used to compute the probability of deadline misses for the tasks. The deadline miss probability DMP_i of task τ_i can be computed as follows:

$$\text{DMP}_i = \mathbb{P}\{\mathcal{R}_i > D_i\} = 1 - \mathbb{P}\{\mathcal{R}_i \leq D_i\} \quad (9.1)$$

9.4 Stochastic Analysis Framework

9.4.1 Overview

The goal of the proposed analysis framework is to accurately compute the *stationary* response time distributions of all the jobs, when the system is in the steady state. The stationary response time distribution of job J_j can be defined as follows:

$$\lim_{k \rightarrow \infty} f_{\mathcal{R}_j}^{(k)} = f_{\mathcal{R}_j}^{(\infty)}$$

where $f_{\mathcal{R}_j}^{(k)}$ is the response time PMF of $J_j^{(k)}$, the j th job in the k th hyperperiod. In this section, we will describe how to compute the response time distributions of all the jobs in an arbitrary hyperperiod k , and then, in the following section, explain how to compute the stationary distributions, which are obtained when $k \rightarrow \infty$. We start our discussion by explaining how the response time \mathcal{R}_j of a job J_j is determined.

The response time of a job J_j is determined by two factors. One is the pending workload that delays the execution of J_j , which is observed immediately prior to its release time λ_j . We call this pending workload *backlog*. The other is the workload of jobs that may preempt J_j , which is released after J_j . We call this

workload *interference*. Since both the backlog and the interference for J_j consist of jobs with a priority higher than that of J_j (i.e., with a priority value smaller than the priority value p_j of J_j), we can elaborate the two terms to p_j -backlog and p_j -interference, respectively. Thus, the response time of J_j can be expressed by the following equation:

$$\mathcal{R}_j = \mathcal{W}_{p_j}(\lambda_j) + \mathcal{C}_j + \mathcal{I}_{p_j} \quad (9.2)$$

where $\mathcal{W}_{p_j}(\lambda_j)$ is the p_j -backlog observed at time λ_j , \mathcal{C}_j the execution time of J_j , and \mathcal{I}_{p_j} the p_j -interference occurring after time λ_j .

In our framework, we compute the distribution of the response time \mathcal{R}_j in two steps: *backlog analysis* and *interference analysis*. In the backlog analysis, the stationary p_j -backlog distributions $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$ of all the jobs in a hyperperiod are computed. Then in the interference analysis, the stationary response time distributions $f_{\mathcal{R}_j}(\cdot)$ of the jobs are determined by introducing the associated execution time distribution $f_{\mathcal{C}_j}(\cdot)$ and the p_j -interference effect \mathcal{I}_{p_j} into each stationary p_j -backlog distribution $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$.

9.4.2 Backlog Analysis Algorithm

For the backlog analysis, we assume a job sequence $\{J_1, \dots, J_j\}$ in which all the jobs have a priority value smaller than or equal to p_j . It is also assumed that the stationary p_j -backlog distribution observed immediate prior to the release time of the first job J_1 , i.e., $f_{\mathcal{W}_{p_j}(\lambda_1)}(\cdot)$, is given. In Section 9.5, it will be explained how the assumed stationary backlog distribution can be computed. Then the p_j -backlog distribution $f_{\mathcal{W}_{p_j}(\lambda_j)}(\cdot)$ immediate prior to the release time of J_j can be computed from $f_{\mathcal{W}_{p_j}(\lambda_1)}(\cdot)$ by the algorithm described in this subsection. For the sake of brevity, we will simplify the notation $\mathcal{W}_{p_j}(\lambda_j)$ to $\mathcal{W}(\lambda_j)$, i.e., without the subscript denoting the priority level p_j .

Let us first consider how to compute the backlog when the execution times of all the jobs are given as deterministic values. In this deterministic scenario, the backlog $W(\lambda_k)$ immediate prior to the release time of each job J_k ($1 \leq k < j$) can be expressed as follows:

$$W(\lambda_{k+1}) = \max\{W(\lambda_k) + C_k - (\lambda_{k+1} - \lambda_k), 0\} \quad (9.3)$$

So, once the backlog $W(\lambda_1)$ for the first job J_1 is given, the series of the backlog $\{\mathcal{W}(\lambda_2), \mathcal{W}(\lambda_3), \dots, \mathcal{W}(\lambda_j)\}$ can be calculated by repeatedly applying Equation 9.3 along the job sequence.

Then we can explain our backlog analysis algorithm as a stochastic extension of Equation 9.3. Deterministic variables $W(\lambda_k)$ and C_k are translated into random variables $\mathcal{W}(\lambda_k)$ and \mathcal{C}_k , and Equation 9.3 is translated into a numerical procedure on the associated PMFs. This procedure can be summarized in the following three steps:

1. The expression “ $\mathcal{W}(\lambda_k) + \mathcal{C}_k$ ” is translated into convolution between the two PMFs of the random variables $\mathcal{W}(\lambda_k)$ and \mathcal{C}_k , respectively.

$$f_{\mathcal{W}(\lambda_k) + \mathcal{C}_k}(\cdot) = (f_{\mathcal{W}(\lambda_k)} \otimes f_{\mathcal{C}_k})(\cdot)$$

In Figure 9.1, for example, the arrow annotated with “Convolve” shows such a convolution operation.

2. The expression “ $\mathcal{W}(\lambda_k) + \mathcal{C}_k - (\lambda_{k+1} - \lambda_k)$ ” is translated into shifting the PMF $f_{\mathcal{W}(\lambda_k) + \mathcal{C}_k}(\cdot)$ obtained above by $(\lambda_{k+1} - \lambda_k)$ units to the left. In the example shown in Figure 9.1, the amount of the shift is six time units.
3. The expression “ $\max\{\mathcal{W}(\lambda_k) + \mathcal{C}_k - (\lambda_{k+1} - \lambda_k), 0\}$ ” is translated into summing up all the probability values in the negative range of the PMF obtained above and adding the sum to the probability of the backlog equal to zero. In the above example, the probability sum is $20/54$.

These three steps exactly describe how to obtain the backlog PMF $f_{\mathcal{W}(\lambda_{k+1})}(\cdot)$ from the preceding backlog PMF $f_{\mathcal{W}(\lambda_k)}(\cdot)$. So, starting from the first job in the given sequence for which the stationary backlog PMF

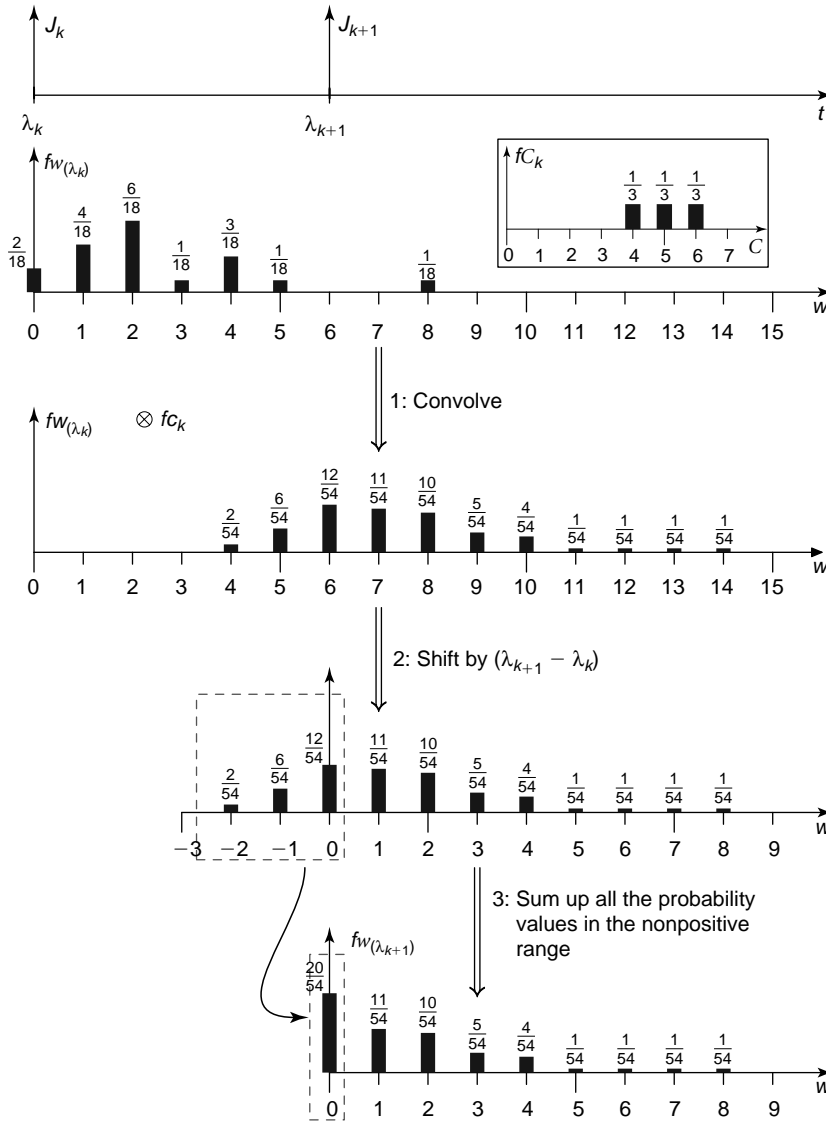


FIGURE 9.1 An example of backlog analysis using the convolve-shrink procedure.

$f_{W(\lambda_1)}(\cdot)$ is assumed to be known, we can compute the stationary backlog PMF of the last job J_j by repeatedly applying the above procedure along the sequence. We refer to this procedure as “convolve-shrink.”

9.4.3 Interference Analysis Algorithm

Once the p_j -backlog PMF is computed for each job J_j at its release time by the backlog analysis algorithm described above, we can easily obtain the response time PMF of the job J_j by convolving the p_j -backlog PMF $f_{W_{p_j}(\lambda_j)}(\cdot)$ and the execution time PMF $f_{C_j}(\cdot)$. This response time PMF is correct if the job J_j is nonpreemptable. However, if J_j is preemptable and there exist higher-priority jobs following J_j , we have to further analyze the p_j -interference for J_j , caused by all the higher-priority jobs, to obtain the complete response time PMF.

For the interference analysis, we have to identify all the higher-priority jobs following J_j . These higher-priority jobs can easily be found by searching for all the jobs released later than J_j and comparing their

priorities with that of J_j . For the sake of brevity, we represent these jobs with $\{J_{j+1}, J_{j+2}, \dots, J_{j+k}, \dots\}$, while slightly changing the meaning of the notation λ_{j+k} from the absolute release time to the release time relative to λ_j , i.e., $\lambda_{j+k} \leftarrow (\lambda_{j+k} - \lambda_j)$.

As in the case of the backlog analysis algorithm, let us first consider how to compute the response time R_j of J_j when the execution times of all the jobs are given as deterministic values. In this deterministic scenario, the response time R_j of J_j can be computed by the following algorithm:

$$\begin{aligned} R_j &= W_{p_j}(\lambda_j) + C_j; \quad k = 1 \\ \text{while } R_j &> \lambda_{j+k} \\ R_j &= R_j + C_{j+k}; \quad k = k + 1 \end{aligned} \tag{9.4}$$

The total number k of iterations of the “while” loop is determined by the final response time that does not reach the release time of the next higher-priority job J_{j+k+1} . For an arbitrary value k , the final response time R_j is given as $W_{p_j}(\lambda_j) + C_j + \sum_{l=1}^k C_{j+l}$.

We can explain our interference analysis algorithm as a stochastic extension of Algorithm (9.4). We treat deterministic variables R_j and C_j as random variables \mathcal{R}_j and \mathcal{C}_j , and translate Algorithm (9.4) into a numerical procedure on the associated PMFs as follows:

1. The expression “ $\mathcal{R}_j = W_{p_j}(\lambda_j) + C_j$ ” is translated into $f_{\mathcal{R}_j}(\cdot) = (f_{W_{p_j}(\lambda_j)} \otimes f_{C_j})(\cdot)$. This response time PMF is valid in the interval $(0, \lambda_{j+1}]$. For example, in Figure 9.2, the first convolution \otimes shows the corresponding operation.
2. While $\mathbb{P}\{\mathcal{R}_j > \lambda_{j+k}\} > 0$, the expression “ $\mathcal{R}_j = \mathcal{R}_j + C_{j+k}$ ” is translated into convolution between the partial PMF defined in the range (λ_{j+k}, ∞) of the response time PMF $f_{\mathcal{R}_j}(\cdot)$ calculated in the previous iteration and the execution time PMF $f_{C_{j+k}}(\cdot)$. The resulting PMF is valid in the range $(\lambda_{j+k}, \lambda_{j+k+1}]$. When $\mathbb{P}\{\mathcal{R}_j > \lambda_{j+k}\} = 0$, the loop is terminated. In the example shown in Figure 9.2, this procedure is described by the two successive convolutions, where only two higher-priority jobs J_{j+1} and J_{j+2} are assumed (In this case, all three jobs are assumed to have the same execution time distribution).

Note that in the above procedure the number of higher-priority jobs we have to consider in a real system can be infinite. However, in practice, since we are often interested only in the probability of job J_j missing the deadline D_j , the set of interfering jobs we have to consider can be limited to the jobs released in the time interval $(\lambda_j, \lambda_j + D_j)$. This is because we can compute the DMP, i.e., $\mathbb{P}\{\mathcal{R}_j > D_j\}$, from the partial response time distribution defined in the range $[0, D_j]$, i.e., $\mathbb{P}\{\mathcal{R}_j > D_j\} = 1 - \mathbb{P}\{\mathcal{R}_j \leq D_j\}$. Thus, we can terminate the “while” loop of Algorithm (9.4) when λ_{j+k} is greater than D_j . For the example in Figure 9.2, if the relative deadline D_j of J_j is 7, the DMP will be $\mathbb{P}\{\mathcal{R}_j > D_j\} = 1 - 11/16 = 5/16$.

We will refer to the above procedure as “split–convolve–merge,” since at each step the response time PMF being computed is split, the resulting tail is convolved with the associated execution time distribution, and this newly made tail and the original head are merged.

9.4.4 Backlog Dependency Tree

In the backlog analysis algorithm, for a given job J_j , we assumed that a sequence of preceding jobs with a priority higher than or equal to that of J_j and the stationary backlog distribution of the first job in the sequence were given. In this subsection, we will explain how to derive such a job sequence for each job in a hyperperiod. As a result, we give a *backlog dependency tree* in which p_j -backlog distributions of all the jobs in the hyperperiod can be computed by traversing the tree while applying the convolve–shrink procedure. This backlog dependency tree greatly simplifies the steady-state backlog analysis for the jobs, since it reduces the problem to computing only the stationary backlog distribution of the root job of the tree. In Section 9.5, we will address how to compute the stationary backlog distribution for the root job.

To show that there exist dependencies between the p_j -backlogs, we first classify all the jobs in a hyperperiod into *ground jobs* and *nonground jobs*. A ground job is defined as a job that has a lower priority

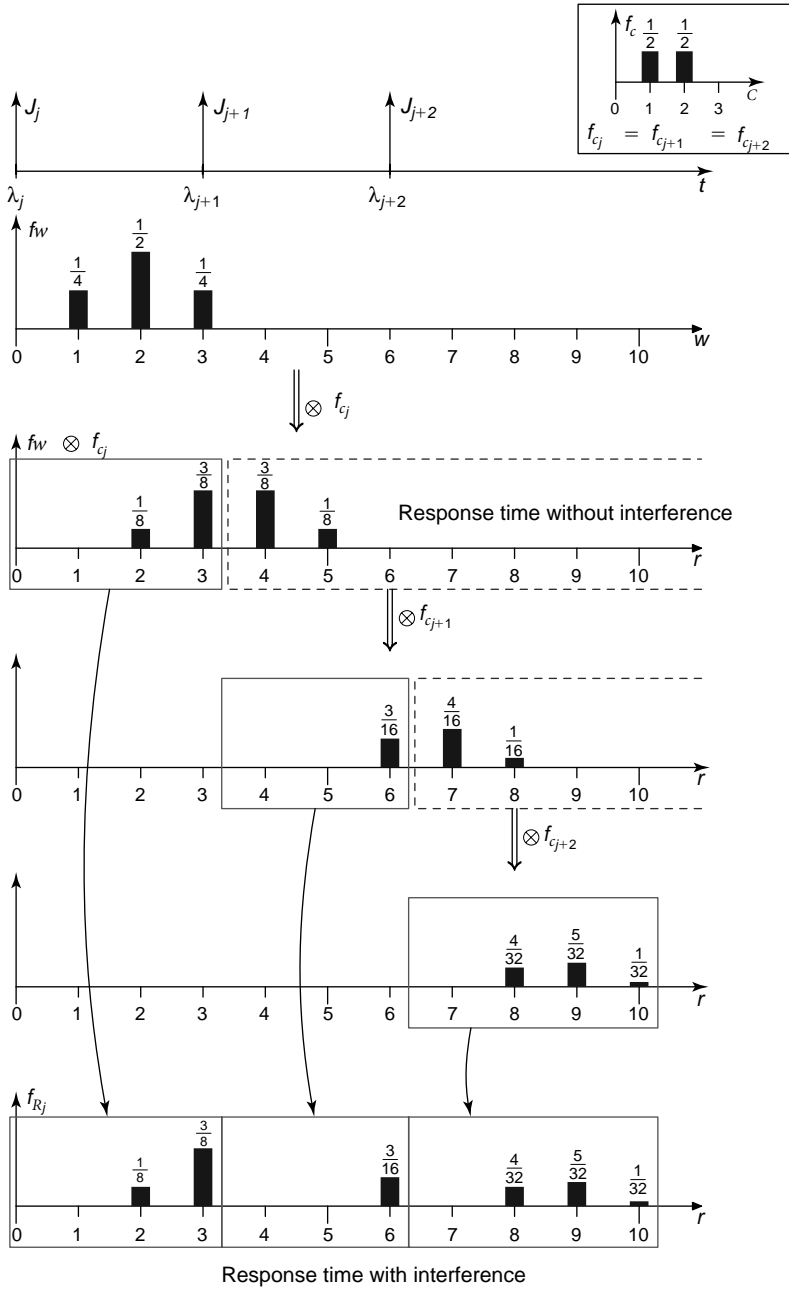


FIGURE 9.2 An example of interference analysis using the split-convolve-merge procedure.

than those of all the jobs previously released. That is, J_j is a ground job if and only if $p_k \leq p_j$ for all jobs J_k such that $\lambda_k < \lambda_j$. A nonground job is a job that is not a ground job. One important implication from the ground job definition is that the p_j -backlog of a ground job is always equal to the total backlog in the system observed at its release time. We call the total backlog as *system backlog* and denote it by $\mathcal{W}(t)$, i.e., without the subscript p_j denoting the priority level. So, for a ground job J_j , $\mathcal{W}_{p_j}(\lambda_j) = \mathcal{W}(\lambda_j)$.

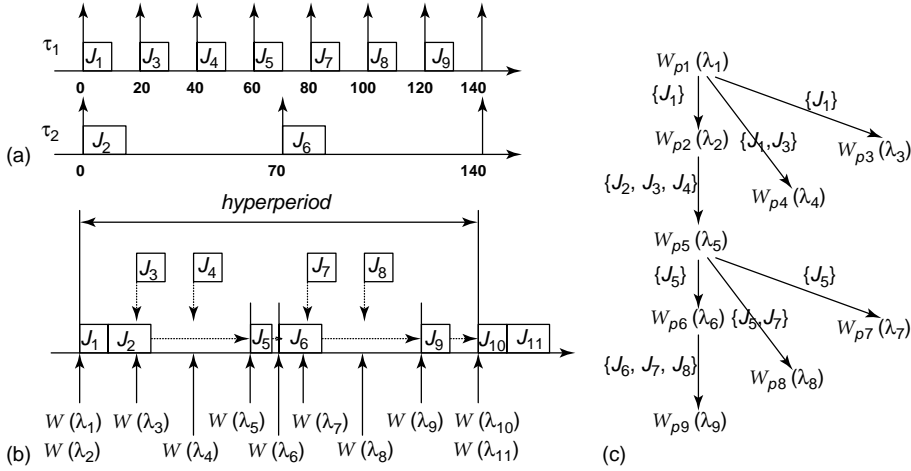


FIGURE 9.3 An example of backlog dependency tree generation. (a) Task set, (b) ground jobs and nonground jobs, and (c) backlog dependency tree.

Let us consider the task set example shown in Figure 9.3a. This task set consists of two tasks τ_1 and τ_2 with the relative deadlines equal to the periods 20 and 70, respectively. The phases Φ_i of both tasks are zero. We assume that these tasks are scheduled by EDF.

In this example, there are five ground jobs J_1, J_2, J_5, J_6 , and J_9 , and four nonground jobs J_3, J_4, J_7 , and J_8 as shown in Figure 9.3b. That is, regardless of the actual execution times of the jobs, $\mathcal{W}_{p1}(\lambda_1) = \mathcal{W}(\lambda_1)$, $\mathcal{W}_{p2}(\lambda_2) = \mathcal{W}(\lambda_2)$ (which is under the assumption that $\mathcal{W}(\lambda_2)$ includes the execution time of J_1 while $\mathcal{W}(\lambda_1)$ does not), $\mathcal{W}_{p5}(\lambda_5) = \mathcal{W}(\lambda_5)$, $\mathcal{W}_{p6}(\lambda_6) = \mathcal{W}(\lambda_6)$, and $\mathcal{W}_{p9}(\lambda_9) = \mathcal{W}(\lambda_9)$. On the contrary, for any of the nonground jobs J_j , $\mathcal{W}_{pj}(\lambda_j) \neq \mathcal{W}(\lambda_j)$. For example, $\mathcal{W}_{p4}(\lambda_4) \neq \mathcal{W}(\lambda_4)$ if J_2 is still running until J_4 is released, since the system backlog $\mathcal{W}(\lambda_4)$ includes the backlog left by J_2 while the p_4 -backlog $\mathcal{W}_{p4}(\lambda_4)$ does not.

We can capture backlog dependencies between the ground and nonground jobs. For each nonground job J_j , we search for the last ground job that is released before J_j and has a priority higher than or equal to that of J_j . Such a ground job is called the *base job* for the nonground job. From this relation, we can observe that the p_j -backlog of the nonground job J_j directly depends on that of the base job. For example, for the task set shown above, the base job of J_3 and J_4 is J_1 , and that of J_7 and J_8 is J_5 . We can see that for the nonground job J_3 , the p_3 -backlog can be directly computed from that of the ground job J_1 by considering only the execution time of J_1 . In this computation, the existence of J_2 is ignored because J_2 has a lower priority than J_3 . Likewise, for the nonground job J_4 , the p_4 -backlog can also be directly computed from that of the ground job J_1 in the same manner, except for the fact that in this case, we have to take into account the arrival of J_3 in between λ_1 and λ_4 (since J_3 has a higher priority than J_4).

Note that such backlog dependencies exist even between ground jobs and can still be captured under the concept of the base job. The base job of J_2 is J_1 , that of J_5 is J_2 , and so on. As a result, all the backlog dependencies among the jobs can be depicted with a tree, as shown in Figure 9.3c. In this figure, each node represents the p_j -backlog $\mathcal{W}_{pj}(\lambda_j)$ of J_j , each link $\mathcal{W}_{pk}(\lambda_k) \rightarrow \mathcal{W}_{pj}(\lambda_j)$ represents the dependency between $\mathcal{W}_{pk}(\lambda_k)$ and $\mathcal{W}_{pj}(\lambda_j)$, and the label on each link represents the set of jobs that should be taken into account to compute $\mathcal{W}_{pj}(\lambda_j)$ from $\mathcal{W}_{pk}(\lambda_k)$.

It is important to understand that this backlog dependency tree completely encapsulates all the job sequences required in computing the p_j -backlog's $\mathcal{W}_{pj}(\lambda_j)$ of all the jobs in the hyperperiod. For example, let us consider the path from $\mathcal{W}_{p1}(\lambda_1)$ to $\mathcal{W}_{p8}(\lambda_8)$. We can see that the set of labels found in the path represents the exact sequence of jobs that should be considered in computing $\mathcal{W}_{p8}(\lambda_8)$ from $\mathcal{W}_{p1}(\lambda_1)$. That is, the job sequence $\{J_1, J_2, J_3, J_4, J_5, J_7\}$ includes all the jobs with a priority higher than or equal to that of J_8 , among all the jobs preceding J_8 . This property is applied for every node in the tree. Therefore, given

the stationary root backlog distribution, i.e., $f_{\mathcal{W}_{p_1}(\lambda_1)}(\cdot)$, we can compute the stationary p_j -backlog distributions of all the other jobs in the hyperperiod by traversing the tree while applying the convolve-shrink procedure.

Finally, note that there is one optimization issue in the dependency tree. In the cases of computing $\mathcal{W}_{p_3}(\lambda_3)$ and computing $\mathcal{W}_{p_4}(\lambda_4)$, the associated job sequences are $\{J_1\}$ and $\{J_1, J_3\}$, and the former is a subsequence of the latter. In this case, since we can obtain $\mathcal{W}_{p_3}(\lambda_3)$ while computing $\mathcal{W}_{p_4}(\lambda_4)$ with the sequence $\{J_1, J_3\}$, i.e., $\mathcal{W}_{p_3}(\lambda_3) = \mathcal{W}_{p_4}(\lambda_3)$, the redundant computation for $\mathcal{W}_{p_3}(\lambda_3)$ with the sequence $\{J_1\}$ can be avoided. This observation is also applied to the case of nonground jobs J_7 and J_8 . It suffices to note that such redundancies can easily be removed by certain steps of tree manipulation.

9.4.5 Extension to Dynamic-Priority and Fixed-Priority Systems

In this subsection, we will prove the existence of ground jobs for the job-level fixed-priority scheduling model [16]. We will also prove the existence of base jobs while distinguishing between fixed-priority and dynamic-priority systems.

Theorem 9.1

Let $S = \{\tau_1, \dots, \tau_n\}$ be a periodic task set in which each task generates a sequence of jobs with a deterministic period T_i and phase Φ_i . Also, let $T_H = \text{lcm}_{1 \leq i \leq n} \{T_i\}$, i.e., the length of a hyperperiod. Consider a sequence of hyperperiods, the first of which starts at time t ($0 \leq t < T_H$). Then, for any t , if the relative priorities of all jobs in a hyperperiod $[t + kT_H, t + (k+1)T_H)$ coincide with those of all jobs in the next hyperperiod $[t + (k+1)T_H, t + (k+2)T_H)$ ($k = 0, 1, \dots$), it follows that

- (a) *At least one ground job exists in any hyperperiod.*
- (b) *The same set of ground jobs are found for all the hyperperiods.*

Proof

(a) Assume that all the jobs have distinct priority values. If there exist jobs with the same priority value, they can always be reassigned distinct priority values while respecting the FCFS (first come first serve) principle or a user-defined principle. Then for any hyperperiod k , i.e., $[t + kT_H, t + (k+1)T_H)$, we can find a job J_j with the maximum priority value p_{\max} in the hyperperiod. This guarantees that J_j has a higher priority value (or a lower priority) than all the preceding jobs released in $[t + kT_H, \lambda_j)$. Then, since the previous instance of J_j released at time $\lambda_j - T_H$ has a lower priority value than J_j , and any job released in $[\lambda_j - T_H, t + kT_H)$ has a lower priority value than the previous instance, it follows that J_j even has a higher priority value than all the jobs released in $[\lambda_j - T_H, \lambda_j)$. Likewise, it can be shown that J_j has a higher priority value than all the jobs released in $[\lambda_j - 2T_H, \lambda_j)$, $[\lambda_j - 3T_H, \lambda_j)$, and so on. Therefore, J_j is a ground job, and for any hyperperiod, there exists at least one ground job.

(b) This is straightforward from the proof of (a).

The key point of the proof is that, in any hyperperiod, a job with the maximum priority value always has a lower priority than any preceding jobs. From this, it is easy to devise an algorithm to find all the ground jobs in a hyperperiod. First, we take an arbitrary hyperperiod and simply find the job J_j with the maximum priority value. This is a ground job. After that, we find all the other ground jobs by searching the single hyperperiod starting at the release time of the ground job, i.e., $[\lambda_j, \lambda_j + T_H)$. In this search, we simply have to check whether a job J_l in the hyperperiod has a greater priority value than all the preceding jobs released in the hyperperiod, i.e., $\{J_j, \dots, J_{l-1}\}$.

In the following, we will address the existence of the base jobs for dynamic-priority systems such as EDF.

Theorem 9.2

For a system defined in Theorem 9.1, if the priority value $p_j^{(k)}$ of every job $J_j^{(k)}$ in the hyperperiod k (≥ 2) can be expressed as $p_j^{(k)} = p_j^{(k-1)} + \Delta$, where Δ is an arbitrary positive constant, any job in a hyperperiod can always find its base job among the preceding ground jobs in the same hyperperiod or a preceding hyperperiod.

Proof

Since it is trivial to show that the base job of a ground job can always be found among the preceding ground jobs (actually the base job is the immediately preceding ground job), we focus only on the base job for a nonground job.

Let us assume a case where the base job for a nonground job $J_j^{(k)}$ is not found in the same hyperperiod k , and let $J_i^{(k)}$ be a ground job in the hyperperiod that has a higher priority value than the job $J_j^{(k)}$. That is, $J_i^{(k)}$ is not the base job of $J_j^{(k)}$. Then we can always find a previous instance $J_i^{(h)}$ of $J_i^{(k)}$ in a preceding hyperperiod $h (< k)$ such that $p_i^{(h)} \leq p_j^{(k)}$, by choosing an appropriate value h that satisfies the inequality $k - h \geq (p_i^{(k)} - p_j^{(k)})/\Delta$. Since $p_i^{(k)} = p_i^{(h)} + (k - h)\Delta$, such a value h always satisfies $p_i^{(h)} \leq p_j^{(k)}$. Then, since $J_i^{(h)}$ is also a ground job (recall Theorem 9.1b), it can be taken as the base job of $J_j^{(k)}$ if no other eligible ground job is found. Therefore, for any nonground job J_j , we can always find the base job among the preceding ground jobs.

For EDF, $\Delta = T_H$, since the priority value assigned to each job is the absolute deadline.

Note that in our analysis framework it does not matter whether the base job J_i is found in the same hyperperiod (say k) the nonground job J_j belongs to, or a preceding hyperperiod (say h), since the case where the base job J_i is found in a preceding hyperperiod simply means that the corresponding job sequence from J_i to J_j spans over the multiple hyperperiods from the hyperperiod h to k . Even in this case, since it is possible to compute the stationary backlog distribution for the root of the backlog dependency tree that originates from the hyperperiod h , through the steady-state analysis in Section 9.5, the backlog distribution of such a nonground job J_j can be computed along the derived job sequence.

The next possible question will be whether there exists a bound on the search range for the base jobs. Theorem 9.3 addresses this problem for EDF.

Theorem 9.3

For EDF, it is always possible to find the base job of any nonground job J_j in the time window $[\lambda_j - (D^{\max} + T_H), \lambda_j]$, where $D^{\max} = \max_{1 \leq i \leq n} D_i$. That is, the search range for the base job is bounded by $D^{\max} + T_H$.

Proof

Let τ_i be the task with $D_i = D^{\max}$, and J_k an instance of τ_i . Then J_k is a ground job, since the priority value p_k is $\lambda_k + D^{\max}$, and all the previously released jobs have lower priority values. Let J_j be a nonground job arriving at the beginning of the hyperperiod $[\lambda_k + D^{\max}, \lambda_k + D^{\max} + T_H]$. Then J_k can be taken as the base job of J_j in the worst case, since J_k is a preceding ground job that has a lower priority value than J_j . Even if we assume that the nonground job J_j arrives at the end of the hyperperiod, i.e., at time $\lambda_k + D^{\max} + T_H$, J_k can still be taken as the base job of J_j in the worst case. Therefore, the maximum distance between any nonground job J_k and its base job cannot be greater than $D^{\max} + T_H$.

Note that if we consider a case where $D^{\max} < T_H$ (since the opposite case is rare in practice), Theorem 9.3 means that it is sufficient to search at most one preceding hyperperiod to find the base jobs of all the nonground jobs in a hyperperiod.

On the contrary, in fixed-priority systems such as RM and DM, the base jobs of the nonground jobs do not exist among the ground jobs (recall that for such systems Theorem 9.2 does not hold, since $\Delta = 0$). In such systems, all jobs from the lowest-priority task τ_n are classified as ground jobs while all jobs from the other tasks are nonground jobs. In this case, since any ground job always has a lower priority than any nonground job we cannot find the base job for any nonground job (even if all the preceding hyperperiods are searched).

Note, however, that this special case does not compromise our analysis framework. It is still possible to compute the backlog distributions of all the jobs by considering each possible priority level. That is, we can consider a subset of tasks $\{\tau_1, \dots, \tau_i\}$ for each priority level $i = 1, \dots, n$, and compute the backlog

distributions of all the jobs from task τ_i , since the jobs from τ_i are all ground jobs in the subset of the tasks, and there always exist backlog dependencies between the ground jobs.

Therefore, the only difference between dynamic-priority and fixed-priority systems is that for the former the backlog distributions of all the jobs are computed at once with the single backlog dependency tree, while for the latter they are computed by iterative analysis over the n priority levels, which results in n backlog dependency lists.

9.5 Steady-State Backlog Analysis

In this section, we will explain how to analyze the steady-state backlog of a ground job, which is used as the root of the backlog dependency tree or the head of the backlog dependency list. In this analysis, for the ground job J_j , we have to consider an infinite sequence of all the jobs released before J_j , i.e., $\{\dots, J_{j-3}, J_{j-2}, J_{j-1}\}$, since all the preceding jobs contribute to the “system backlog” observed by J_j .

In Section 9.5.1, we will prove the existence of the stationary system backlog distribution (SSBD), and in Sections 9.5.2 and 9.5.3, explain the exact and the approximation methods to compute the stationary distribution. Finally, in Section 9.5.4, it will be discussed how to safely truncate the exact solution, which is infinite, to use it as the root of the backlog dependency tree.

9.5.1 Existence of the Stationary Backlog Distribution

The following theorem states that there exists a *stationary* (or *limiting*) system backlog distribution, as long as the average system utilization \bar{U} is less than 1.

Theorem 9.4

Let us assume an infinite sequence of hyperperiods, the first of which starts at the release time λ_j of the considered ground job J_j . Let $f_{\mathcal{B}_k}(\cdot)$ be the distribution of the system backlog \mathcal{B}_k observed immediately prior to the release time of the ground job $J_j^{(k)}$, i.e., at the beginning of hyperperiod k . Then, if the average system utilization \bar{U} is less than 1, there exists a stationary (or limiting) distribution $f_{\mathcal{B}_\infty}(\cdot)$ of the system backlog \mathcal{B}_k such that $\lim_{k \rightarrow \infty} f_{\mathcal{B}_k} = f_{\mathcal{B}_\infty}$.

Proof

The proof can be found in Ref. 17.

For the special case where $U^{\max} \leq 1$, the system backlog distributions $f_{\mathcal{B}_k}(\cdot)$ of all the hyperperiods are identical. That is, $f_{\mathcal{B}_1} = \dots = f_{\mathcal{B}_k} = \dots = f_{\mathcal{B}_\infty}$. In this case, the stationary backlog distribution $f_{\mathcal{B}_\infty}(\cdot)$ can easily be computed by considering only the finite sequence of the jobs released before the release time of the ground job J_j . That is, we simply have to apply the convolve–shrink procedure along the finite sequence of jobs released in $[0, \lambda_j)$, assuming that the system backlog at time 0 is 0 (i.e., $\mathbb{P}\{\mathcal{W}(0) = 0\} = 1$). Therefore, for the special case where $U^{\max} \leq 1$, the following steady-state backlog analysis is not needed.

9.5.2 Exact Solution

For a general case where $U^{\max} > 1$, to compute the exact solution for the stationary backlog distribution $f_{\mathcal{B}_\infty}(\cdot)$, we show that the stochastic process defined with the sequence of random variables $\{\mathcal{B}_0, \mathcal{B}_1, \dots, \mathcal{B}_k, \dots\}$ is a Markov chain. To do this, let us express the PMF of \mathcal{B}_k in terms of the PMF of \mathcal{B}_{k-1} using the concept of conditional probabilities.

$$\mathbb{P}\{\mathcal{B}_k = x\} = \sum_y \mathbb{P}\{\mathcal{B}_{k-1} = y\} \mathbb{P}\{\mathcal{B}_k = x \mid \mathcal{B}_{k-1} = y\} \quad (9.5)$$

Then we can see that the conditional probabilities $\mathbb{P}\{\mathcal{B}_k = x \mid \mathcal{B}_{k-1} = y\}$ do not depend on k , since all hyperperiods receive the same sequence of jobs with the same execution time distributions. That is, $\mathbb{P}\{\mathcal{B}_k = x \mid \mathcal{B}_{k-1} = y\} = \mathbb{P}\{\mathcal{B}_1 = x \mid \mathcal{B}_0 = y\}$. This leads us to the fact that the PMF of \mathcal{B}_k depends only on that of \mathcal{B}_{k-1} and not on those of $\{\mathcal{B}_{k-2}, \mathcal{B}_{k-3}, \dots\}$. Thus, the stochastic process is a Markov chain. We can rewrite Equation 9.5 in matrix form as follows:

$$\mathbf{b}_k = \mathbf{P} \mathbf{b}_{k-1} \quad (9.6)$$

where \mathbf{b}_k is a column vector $[\mathbb{P}\{\mathcal{B}_k = 0\}, \mathbb{P}\{\mathcal{B}_k = 1\}, \dots]^\top$, i.e., the PMF of \mathcal{B}_k , and \mathbf{P} the Markov matrix, which consists of the transition probabilities $\mathbf{P}(x, y)$ defined as

$$\mathbf{P}(x, y) = b_y(x) = \mathbb{P}\{\mathcal{B}_k = x \mid \mathcal{B}_{k-1} = y\} = \mathbb{P}\{\mathcal{B}_1 = x \mid \mathcal{B}_0 = y\}$$

Thus, the problem of computing the exact solution $\boldsymbol{\pi}$ for the stationary backlog distribution, i.e., $[\mathbb{P}\{\mathcal{B}_\infty = 0\}, \mathbb{P}\{\mathcal{B}_\infty = 1\}, \dots]^\top$, is equivalent to solving the equilibrium equation $\boldsymbol{\pi} = \mathbf{P} \boldsymbol{\pi}$.

However, the equilibrium equation $\boldsymbol{\pi} = \mathbf{P} \boldsymbol{\pi}$ cannot be directly solved, since the number of linear equations obtained from it is infinite. Theoretically, when $k \rightarrow \infty$, the system backlog can be arbitrarily long, since $U^{\max} > 1$. This means that the exact solution $\boldsymbol{\pi}$ has an infinite length, and the Markov matrix is therefore also of infinite size. We address this problem by deriving a finite set of linear equations that is equivalent to the original infinite set of linear equations. This is possible due to the regular structure of the Markov matrix proven below.

$$\mathbf{P} = \begin{pmatrix} b_0(0) & b_1(0) & b_2(0) & \dots & b_r(0) & 0 & 0 & 0 \\ b_0(1) & b_1(1) & b_2(1) & \dots & b_r(1) & b_r(0) & 0 & 0 \\ b_0(2) & b_1(2) & b_2(2) & \dots & b_r(2) & b_r(1) & b_r(0) & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & b_r(2) & b_r(1) & \ddots \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots & b_r(2) & \ddots \\ b_0(m_r) & b_1(m_r) & b_2(m_r) & \dots & b_r(m_r) & \vdots & \vdots & \ddots \\ 0 & 0 & 0 & \dots & 0 & b_r(m_r) & \vdots & \ddots \\ 0 & 0 & 0 & \dots & 0 & 0 & b_r(m_r) & \ddots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \ddots \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Each column y in the Markov matrix \mathbf{P} is the backlog PMF observed at the end of a hyperperiod when the amount of the backlog at the beginning of the hyperperiod is y . The backlog PMF of the column y can be calculated by applying the convolve-shrink procedure (in Section 9.4.2) along the whole sequence of jobs in the hyperperiod, assuming that the initial backlog is equal to y . So, the regular structure found in the Markov matrix, i.e., the columns $r, r+1, r+2, \dots$, with the same backlog PMF only shifted down by one position, means that there exists a value r for the initial backlog from which onward the backlog PMF observed at the end of the hyperperiod is always the same, only shifted one position to the right in the system. The value r is the maximum sum of all the possible idle times occurring in a hyperperiod. It is equal to

$$r = T_H(1 - U^{\min}) + W^{\min} \quad (9.7)$$

where W^{\min} is the system backlog observed at the end of the hyperperiod when the initial system backlog is zero and all the jobs have minimum execution times (W^{\min} is usually zero unless most of the workload is concentrated at the end of the hyperperiod). If the initial backlog is r , the whole hyperperiod is busy,

and thus the backlog PMF observed at the end of the hyperperiod is simply the result of convolving the execution time distributions of all the jobs, shifted $(T_H - r)$ units to the left. The length of the backlog PMF is $(m_r + 1)$, where m_r is the index of the last nonzero element in column r . This observation is analogously applied to all cases where the initial backlog is larger than r .

Using the above regularity, we can derive the equivalent finite set of linear equations as follows. First, we take the first $(m_r + 1)$ linear equations from $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$, which correspond to rows 0 to m_r in the Markov matrix. The number of unknowns appearing in the $(m_r + 1)$ linear equations is $(r + m_r + 1)$, i.e., $\{\pi_0, \pi_1, \dots, \pi_{r+m_r}\}$. Next, we derive r additional equations from the fact that $\pi_x \rightarrow 0$ when $x \rightarrow \infty$, to complete the finite set of linear equations, i.e., $(r + m_r + 1)$ linear equations with the $(r + m_r + 1)$ unknowns. For this derivation, from rows $(m_r + 1)$, $(m_r + 2)$, ... in the Markov matrix we extract the following equation:

$$\mathbf{Q}_{x+1} = \mathbf{A} \mathbf{Q}_x \quad x \geq m_r + 1 \quad (9.8)$$

where

$$\mathbf{A} = \begin{pmatrix} \mathbf{Q}_x = [\pi_{x-d}, \pi_{x-d+1}, \dots, \pi_{x-1}, \pi_x, \pi_{x+1}, \dots, \pi_{x-d+m_r-1}]^T, & d = m_r - r \\ \begin{matrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -\frac{b_r(m_r)}{b_r(0)} & -\frac{b_r(m_r-1)}{b_r(0)} & \dots & -\frac{b_r(d+1)}{b_r(0)} & \frac{1-b_r(d)}{b_r(0)} & -\frac{b_r(d-1)}{b_r(0)} & \dots & -\frac{b_r(1)}{b_r(0)} \end{matrix} \end{pmatrix}$$

Then, by diagonalizing the companion-form matrix \mathbf{A} , it can be shown that the general form of π_x is expressed as follows:

$$\pi_x = \sum_{k=1}^{m_r} a_k \lambda_k^{x-m_r-1} \quad (9.9)$$

where $\{\lambda_1, \lambda_2, \dots, \lambda_{m_r}\}$ are the eigenvalues obtained by the matrix diagonalization, and the associated coefficients a_k are a linear combination of $\{\pi_{r+1}, \pi_{r+2}, \dots, \pi_{r+m_r}\}$. Since it has already been proved in Ref. 17 that in Equation 9.9 there exist $(r + 1)$ eigenvalues λ_k such that $|\lambda_k| \geq 1$, the associated coefficients a_k are equated to 0 because the condition that $\pi_x \rightarrow 0$ when $x \rightarrow \infty$ is met only in this case. As a result, $(r + 1)$ additional linear equations are obtained, but since one linear equation is always degenerate, r linear equations remain.

Therefore, the complete set of linear equations is composed of the first $(m_r + 1)$ linear equations taken from $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$, and the r linear equations obtained by equating to 0 all the coefficients a_k such that $|\lambda_k| \geq 1$. Then the set of $(r + m_r + 1)$ linear equations with the $(r + m_r + 1)$ unknowns can be solved with a numerical method, and as a result the solution for $\{\pi_0, \pi_1, \dots, \pi_{r+m_r}\}$ is obtained. Once the solution is given, we can complete the general form of π_x , since all the other unknown coefficients a_k , such that $|\lambda_k| < 1$, are calculated from the solution. Therefore, we can finally generate the infinite stationary backlog distribution with the completed general form. For more information about the above process, the reader is referred to Refs. 17 and 18.

9.5.3 Approximated Solutions

Markov matrix truncation method: One possible approximation of the exact solution is to truncate the Markov matrix \mathbf{P} to a finite square matrix \mathbf{P}' . That is, we approximate the problem of $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$ to $\boldsymbol{\pi}' = \mathbf{P}'\boldsymbol{\pi}'$,

where $\boldsymbol{\pi}' = [\pi'_0, \pi'_1, \pi'_2, \dots, \pi'_p]$ and \mathbf{P}' is a square matrix of size $(p+1)$, which consists of the elements $\mathbf{P}(x, y)$ ($0 \leq x, y \leq p$) of the Markov matrix \mathbf{P} . The resulting equation is an eigenvector problem from which we can calculate the approximated solution $\boldsymbol{\pi}'$ with a numerical method. Among the calculated eigenvectors, we can choose as the solution an eigenvector whose eigenvalue is the closest to 1. To obtain a good approximation of the exact solution $\boldsymbol{\pi}$, the truncation point p should be increased as much as possible which makes the eigenvalue converge to 1.

Iterative method: Another approximation method, which does not require the Markov matrix derivation, is simple iteration of the backlog analysis algorithm for the system backlog \mathcal{B}_k over a sufficient number of hyperperiods. Since Theorem 9.4 guarantees that $f_{\mathcal{B}_k}(\cdot)$ converges toward $f_{\mathcal{B}_\infty}(\cdot)$, we can compute $f_{\mathcal{B}_1}, f_{\mathcal{B}_2}, \dots, f_{\mathcal{B}_k}$, in turn, until convergence occurs. That is, while monitoring the quadratic difference $\|f_{\mathcal{B}_k} - f_{\mathcal{B}_{k-1}}\|$ (def. $\|\mathbf{x} - \mathbf{y}\| = \sqrt[p]{\sum_i (x_i - y_i)^2}$), we can continue the computation of $f_{\mathcal{B}_k}(\cdot)$'s until the difference falls below a given threshold ϵ .

For both approximation methods, it is important to choose the associated control parameters appropriately, i.e., the truncation point and the number of iterations, respectively. In general, as \bar{U} approaches 1, a larger value should be used for the control parameters, since the probability values of the stationary backlog distribution spread more widely. Note that by choosing an appropriate value for the control parameters, we can achieve a trade-off between analysis accuracy and the computational overheads required to obtain the approximated solution. We will address this issue in Section 9.7.1.

9.5.4 Safe Truncation of the Exact Solution

As mentioned earlier, the stationary backlog distribution has an infinite length when $U^{\max} > 1$. So, in practice, to use the infinite solution obtained by the exact method as the root of the backlog dependency tree we have to truncate the solution at a certain point. In this subsection, we show that the use of such a truncated solution is safe and that it is “more pessimistic” than the original infinite solution, thus giving an upper bound on the DMP for each task.

Let $f'_{\mathcal{B}_\infty}(\cdot)$ be the solution obtained by truncating the original infinite solution at point M . That is, the truncated solution $f'_{\mathcal{B}_\infty}(\cdot)$ is expressed as follows:

$$f'_{\mathcal{B}_\infty}(w) = \begin{cases} f_{\mathcal{B}_\infty}(w) & w \leq M \\ 0 & w > M \end{cases}$$

The truncated solution is not a complete distribution, since the total sum of the nonzero probabilities is less than 1. In other words, the truncated solution has a “deficit” of $\sum_{w>M} f_{\mathcal{B}_\infty}(w)$. However, it is possible to say that $f'_{\mathcal{B}_\infty}(\cdot)$ is more pessimistic than $f_{\mathcal{B}_\infty}(\cdot)$ in the sense that

$$\sum_{w=0}^t f'_{\mathcal{B}_\infty}(w) \leq \sum_{w=0}^t f_{\mathcal{B}_\infty}(w) \quad \text{for any } t$$

This means that the use of the truncated distribution $f'_{\mathcal{B}_\infty}(\cdot)$ always produces results, which are more pessimistic than that of the original distribution $f_{\mathcal{B}_\infty}(\cdot)$. Thus, it leads to a higher DMP for each task than the original one.

9.6 Computational Complexity

In this section, we investigate the computational complexity of our analysis framework, dividing it into two parts: (1) the complexity of the backlog and interference analysis and (2) the complexity of the steady-state backlog analysis. In this complexity analysis, to make the analysis simple and safe we introduce two assumptions. One assumption is that we regard the deterministic releases of jobs in the hyperperiod as random releases that follow an interarrival time distribution. So, if the total number of jobs in the hyperperiod is

n , the interarrival time distribution is understood as a random distribution with a mean value $\bar{T} = T_H/n$. The other assumption is that all the jobs in the hyperperiod have execution time distributions of the same length m . This simplification is safe, since we can make execution time distributions of different lengths have the same length by zero-padding all the distributions other than the longest one.

9.6.1 Complexity of the Backlog and Interference Analysis

To safely analyze the complexity of the backlog analysis, we assume that, for any job J_j in the hyperperiod, all the preceding jobs $\{J_1, \dots, J_{j-1}\}$ are involved in computing the p_j -backlog distribution. That is, it is assumed that the p_j -backlog distribution can only be computed by applying the convolve–shrink procedure to the stationary backlog distribution of J_1 along the whole sequence of preceding jobs. This scenario is the worst case that can happen in computing the p_j -backlog distribution, since the set of jobs required to compute the p_j -backlog distribution does not necessarily cover all the preceding jobs. So, by assuming the worst-case scenario for every job J_j in the hyperperiod we can safely ignore the complex backlog dependencies among the jobs.

Without loss of generality, assuming that the truncated length M of the stationary backlog distribution of J_1 is expressed as a multiple of the execution time distribution length m , i.e., $s \times m$, let us consider the process of applying the convolve–shrink procedure to each job in the sequence $\{J_1, \dots, J_j\}$. Each convolution operation increases the length of the backlog distribution by $(m - 1)$ points, and each shrink operation reduces the length by \bar{T} points on average. Note that if $\bar{T} \approx (m - 1)$, the backlog distribution length remains constant on average, and thus the convolve–shrink procedure has the same cost for all the jobs in the sequence. However, if $\bar{T} \rightarrow 0$, which implies that U^{\max} becomes significantly high, the backlog distribution length always increases approximately by m points for each iteration. Assuming this pessimistic case for \bar{T} , the complexity of the j th iteration of the convolve–shrink procedure is $O((s + j - 1)m^2)$, since the j th iteration is accompanied by convolution between the backlog distribution of length $(s + j - 1)m$ and the associated execution time distribution of length m . So, the complexity of computing the single p_j -backlog distribution from the stationary backlog distribution is $sm^2 + (s + 1)m^2 + \dots + (s + j - 1)m^2$, i.e., $O(j^2m^2)$. Therefore, the total complexity* of computing the p_j -backlog distributions of all the jobs $\{J_1, \dots, J_n\}$ in the hyperperiod is $O(n^3m^2)$.

Likewise, the complexity of the interference analysis can be analyzed as follows. First, let us consider the complexity for a single job J_j . As explained above, the length of the p_j -backlog distribution of J_j for which the interference analysis is to be applied is $(s + j - 1)m$, so the initial response time distribution (without any interference) will have a length of $(s + j)m$. We can assume that there exists a constant value k (called *interference degree*) that represents the maximum number of interfering jobs, within the deadlines, for any job in the hyperperiod. Then the split–convolve–merge procedure is applied k times to the initial response time distribution of J_j . We can see that the convolution at the i th iteration of the technique has a complexity of $O((l_i - i\bar{T})m)$, where l_i is the length of the response time distribution produced by the $(i - 1)$ th iteration. This iteration increases the response time distribution by $(m - 1)$ points. So, assuming that $\bar{T} \rightarrow 0$, we can say that the i th iteration has a complexity of $O((s + j + i)m^2)$, since $l_i = (s + j + i - 1)m$. Thus, the complexity of applying the split–convolve–merge procedure k times to the initial response time distribution is $(s + j)m^2 + (s + j + 1)m^2 + \dots + (s + j + k - 1)m^2$, i.e., $O(k^2m^2)$. Therefore, if we consider all the n jobs in the hyperperiod the total complexity of the interference analysis is $O(nk^2m^2)$. In particular, by assuming that $k < n$, this complexity can be expressed as $O(n^3m^2)$. This assumption is reasonable, since the fact that $k \geq n$ means that every job in the hyperperiod has a relative deadline greater than or equal to the length of the hyperperiod, which is unrealistic in practice.

*In this analysis, we have assumed that the associated backlog dependency tree is completely built by considering all the jobs in a single hyperperiod only. However, if more than one hyperperiod were to be considered for the complete construction of the backlog dependency tree the term n in $O(n^3m^2)$ should be replaced with the total number of jobs in the multiple hyperperiods.

9.6.2 Complexity of the Steady-State Backlog Analysis

The complexity of the steady-state backlog analysis is different, depending on the solution method used to compute the stationary backlog distribution. First, let us investigate the complexity of the exact method. The exact method consists of three steps: Markov matrix \mathbf{P} derivation, companion-form matrix \mathbf{A} diagonalization, and solving a system of linear equations. The complexity of the Markov matrix derivation is equivalent to that of computing r times the system backlog distribution observed at the end of a hyperperiod from that assumed at the beginning of the hyperperiod, by applying the convolve–shrink procedure along the whole sequence of jobs $\{J_1, \dots, J_n\}$. So, the complexity is $O(rn^2m^2)$, since $O(n^2m^2)$ is the complexity of computing once the system backlog distribution observed at the end of the hyperperiod with n jobs. The complexity of the companion-form matrix \mathbf{A} diagonalization is $O(m_r^3)$, since the diagonalization of a matrix with size l has a complexity of $O(l^3)$ [19]. However, note that m_r is smaller than nm , since $(m_r + 1)$ denotes the length of the backlog distribution obtained by convolving n execution time distributions of length m . So, the complexity of diagonalizing the companion-form matrix \mathbf{A} can be expressed as $O(n^3m^3)$. Finally, the complexity of solving the system of linear equations is $O((m_r + r)^3)$, since a system of l linear equations can be solved in time $O(l^3)$ [20]. This complexity can also be expressed as $O((nm + r)^3)$, since $m_r < nm$. Therefore, the total complexity of the exact method is $O(rn^2m^2) + O(n^3m^3) + O((nm + r)^3)$. This complexity expression can be further simplified to $O(n^3m^3)$ by assuming that $r < nm$. This assumption is reasonable, since $r < T_H = n\bar{T}$ and we can assume that $\bar{T} < m$ when $\bar{T} \rightarrow 0$.

Next, let us consider the complexity of the Markov matrix truncation method. In this case, since the complexity also depends on the chosen truncation point p , let us assume that the value p is given. Then we can see that the complexity* of deriving the truncated Markov matrix \mathbf{P} is $O(pn^2m^2)$, and the complexity of solving the system of p linear equations through matrix diagonalization is $O(p^3)$. Thus, the total complexity is $O(pn^2m^2) + O(p^3)$.

Finally, let us consider the complexity of the iterative method. In this case, the complexity depends on the number of hyperperiods over which the backlog analysis is iterated for convergence. If the number of the hyperperiods is I , then the complexity is $O(I^2n^2m^2)$, since the convolve–shrink procedure should be applied to a sequence of In jobs.

However, we cannot directly compare the complexities of all the methods, since we do not know in advance the appropriate values for the control parameters p and I that can give solutions of the same accuracy. To obtain insight as to how the control parameters should be chosen, we have to investigate system parameters that can affect the accuracy of the approximation methods. This issue will be addressed in the following section.

9.7 Experimental Results

In this section, we will give experimental results obtained using our analysis framework. First, we compare all the proposed solution methods to compute the SSBD, in terms of analysis complexity and accuracy. In this comparison, we vary the system utilization to see its effect on each solution method, and also compare the results with those obtained by STDA [6,7]. Second, we evaluate the complexity of the backlog and interference analysis by experiments to corroborate the complexity asymptotically analyzed in the previous section. In these experiments, while varying n (the number of jobs), m (the maximum length of the execution time distributions), \bar{T} (the average interarrival time), and k (the interference degree) we investigate their effects on the backlog and interference analysis.

*Note that, when the truncation point p is larger than r , the complexity is reduced to $O(rn^2m^2)$, since the last $(p - r)$ columns in the Markov matrix can be replicated from the r th column.

TABLE 9.1 Task Sets Used in the Experiments

Task Set		T_i	D_i	Execution Times			Utilizations		
				C_i^{\min}	\bar{C}_i	C_i^{\max}	U^{\min}	\bar{U}	U^{\max}
A	τ_1	20	20	4	6	10	0.58	0.82	1.27
	τ_2	60	60	12	16	22			
	τ_3	90	90	16	23	36			
B	τ_1	20	20	4	6	10	0.58	0.87	1.27
	τ_2	60	60	12	17	22			
	τ_3	90	90	16	26	36			
C	τ_1	20	20	4	7	10	0.58	0.92	1.27
	τ_2	60	60	12	17	22			
	τ_3	90	90	16	26	36			
C1	τ_1	20	20	3	7	11	0.46	0.92	1.38
	τ_2	60	60	10	17	24			
	τ_3	90	90	13	26	39			
C2	τ_1	20	20	2	7	12	0.34	0.92	1.50
	τ_2	60	60	8	17	26			
	τ_3	90	90	10	26	42			

9.7.1 Comparison between the Solution Methods

To investigate the effect of system utilization on each solution method to compute the SSBD, we use the task sets shown in Table 9.1. All the task sets consist of three tasks with the same periods, the same deadlines, and null phases which result in the same backlog dependency tree for a given scheduling algorithm.

The only difference in the task sets is the execution time distributions. For task sets A, B, and C, the minimum and maximum execution times for each task do not change, while the average execution time is varied. In this case, since the time needed for the backlog and interference analysis is constant, if a system backlog distribution of the same length is used as the root of the backlog dependency tree, we can evaluate the effect of the average system utilization \bar{U} on the SSBD. However, for task sets C, C1, and C2, the average execution time of each task is fixed, while the whole execution time distribution is gradually stretched. In this case, we can evaluate the effect of the maximum system utilization U^{\max} on the SSBD, while fixing the average system utilization \bar{U} .

Table 9.2 summarizes the results of our stochastic analysis and, for the case of RM, also the results obtained by STDA. The table shows the DMP for each task obtained from the SSBD computed by each solution method (i.e., exact, Markov matrix truncation, and iterative), and the average DMR and standard deviation obtained from the simulations. For the truncation and iterative methods, the values used for the control parameters p and I are shown in Table 9.3 (this will be explained later). The average DMR is obtained by averaging the DMRs measured from 100 simulation runs of each task set performed during 5000 hyperperiods. To implement the exact method and the Markov matrix truncation method, we used the Intel linear algebra package called Math Kernel Library 5.2 [21].

From Table 9.2, we can see that our analysis results are almost identical to the simulation results, regardless of the solution method used. For the case of RM, the analysis results obtained by STDA are also given, but we can observe significant differences between the DMPs given by STDA and those obtained by our analysis. In the case of task τ_3 in task set A, the DMP given by STDA (39.3%) is more than four times that given by our analysis (9.4%). Moreover, as \bar{U} or U^{\max} increases the DMP computed by STDA gets even worse. This results from the critical instant assumption made in STDA.

In contrast, our implementation of the exact method could not provide a numerically valid result for task set C2 (in the case of RM, only for task τ_3). This is because the numerical package we used, which uses the 64-bit floating point type, may result in an ill-conditioned set of linear equations when a significantly small probability value $b_r(0)$ is used as the divisor in making the companion-form matrix **A**. (Recall from Section 9.5.2 that $b_r(0)$ is the probability that all the jobs in the hyperperiod have the minimum

TABLE 9.2 Analysis Accuracy Comparison between the Solution Methods (DMP)

Task Set		RM					EDF			
		Simulation	STDA	Exact	Truncation	Iterative	Simulation	Exact	Truncation	Iterative
A	τ_1	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0001 ± 0.0000	0.0001	0.0001	0.0001
	τ_2	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0000 ± 0.0000	0.0000	0.0000	0.0000
	τ_3	0.0940 ± 0.0025	0.3931	0.0940	0.0940	0.0940	0.0000 ± 0.0000	0.0000	0.0000	0.0000
B	τ_1	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0013 ± 0.0002	0.0013	0.0013	0.0013
	τ_2	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0005 ± 0.0002	0.0005	0.0005	0.0005
	τ_3	0.2173 ± 0.0033	0.6913	0.2170	0.2170	0.2170	0.0000 ± 0.0001	0.0000	0.0000	0.0000
C	τ_1	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0223 ± 0.0013	0.0224	0.0224	0.0224
	τ_2	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0168 ± 0.0014	0.0169	0.0169	0.0169
	τ_3	0.3849 ± 0.0052	0.9075	0.3852	0.3852	0.3852	0.0081 ± 0.0011	0.0081	0.0081	0.0081
C1	τ_1	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0626 ± 0.0031	0.0630	0.0627	0.0627
	τ_2	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.0604 ± 0.0038	0.0610	0.0607	0.0607
	τ_3	0.4332 ± 0.0065	0.9209	0.4334	0.4334	0.4334	0.0461 ± 0.0032	0.0466	0.0463	0.0463
C2	τ_1	0.0000 ± 0.0000	0.0000	0.0000	0.0000	0.0000	0.1248 ± 0.0058	N.A.	0.1250	0.1250
	τ_2	0.0002 ± 0.0001	0.0018	0.0002	0.0002	0.0002	0.1293 ± 0.0064	N.A.	0.1296	0.1296
	τ_3	0.4859 ± 0.0081	0.9339	N.A.	0.4860	0.4860	0.1136 ± 0.0063	N.A.	0.1138	0.1138

TABLE 9.3 Analysis Time Comparison between the Solution Methods

Task Set	SSBD Computation Time (s)						
	Exact	Truncation			Iterative		
		$\delta = 10^{-3}$	$\delta = 10^{-6}$	$\delta = 10^{-9}$	$\delta = 10^{-3}$	$\delta = 10^{-6}$	$\delta = 10^{-9}$
A	0.13	0.00	0.00	0.00	0.00	0.00	0.00
		$p = 2$	$p = 15$	$p = 25$	$I = 2$	$I = 2$	$I = 3$
B	0.13	0.00	0.00	0.01	0.00	0.00	0.01
		$p = 8$	$p = 23$	$p = 37$	$I = 2$	$I = 3$	$I = 6$
C	0.15	0.01	0.03	0.07	0.00	0.01	0.03
		$p = 29$	$p = 63$	$p = 96$	$I = 4$	$I = 12$	$I = 20$
C1	0.31	0.02	0.10	0.25	0.01	0.05	0.21
		$p = 54$	$p = 115$	$p = 173$	$I = 7$	$I = 20$	$I = 35$
C2	N.A.	0.07	0.31	0.82	0.02	0.23	0.88
		$p = 86$	$p = 181$	$p = 272$	$I = 10$	$I = 30$	$I = 52$

execution times.) In the case of C2, the probability value $b_r(0)$ was 5×10^{-17} . This is also the reason why a small difference is observed between the DMP computed by the exact method and those computed by the approximation methods for task set C1, scheduled by EDF. However, note that this precision problem can be overcome simply by using a numerical package with a higher precision.

Table 9.3 shows in the case of EDF the analysis time* required by each solution method to compute the SSBDs used to produce the results in Table 9.2. The analysis time does not include the time taken by the backlog dependency tree generation, which is almost 0, and the time required by the backlog and interference analysis, which is less than 10 ms. The table also shows the values of the control parameters, p and I , used for the truncation and iterative methods. For fair comparison between the two approximation methods, we define an accuracy level δ to be the quadratic difference between the exact solution of the stationary system backlog distribution $\text{SSBD}_{\text{exact}}$ and the approximated solution computed by either of

*The analysis time was measured with a Unix system call called `times()` on a personal computer equipped with a Pentium Processor IV 2.0 GHz and 256 MB main memory.

the methods $\text{SSBD}_{\text{approx}}$, i.e., $\delta = \|\text{SSBD}_{\text{exact}} - \text{SSBD}_{\text{approx}}\|$. In the evaluation of δ , however, due to the numerical errors that can be caused by our implementation of the exact method, we do not refer to the solution given by our implementation as $\text{SSBD}_{\text{exact}}$ but to the solution obtained by infinitely applying the iterative method to the corresponding task set until the resulting solution converges.

In Table 9.3, we can see both the SSBD computation time and the associated control parameters used to obtain solutions with the required accuracy levels $\delta = 10^{-3}$, 10^{-6} , and 10^{-9} . (The DMPs shown in Table 9.2 for the truncation and iterative methods were obtained at an accuracy level of $\delta = 10^{-6}$.) From the results shown for task sets A–C, we can see that, as \bar{U} increases, the analysis time also rapidly increases for the truncation and iterative methods, while it stays almost constant for the exact method. The reason for this is that as \bar{U} increases the probability values of the stationary backlog distribution spread more widely, so both approximation methods should compute the solution for a wider range of the backlog. That is, both methods should use a larger value for the associated control parameters, p and I , to achieve the required accuracy level. For the exact method, on the contrary, this spread of the stationary probability values does not affect the analysis time, since the method originally derives a general form solution from which the SSBD can be completely generated.

The above observation is analogously applied for the results from task sets C to C2. Owing to the increasing U^{\max} , the SSBD spreads even more widely, so the truncation and iterative methods should increase the associated control parameters even more to achieve the required accuracy level. We can see that the analysis time taken by the exact method also increases, but this is not because the stationary backlog distribution spreads, but because the size of the resulting companion-form matrix \mathbf{A} becomes large due to the increasing length of the execution time distributions.

In summary, if \bar{U} or U^{\max} is significantly high the approximation methods require a long computation time for high accuracy, possibly larger than that of the exact method. However, if \bar{U} is not close to 1, e.g., less than 0.8, the methods can provide highly accurate solutions at a considerably lower complexity.

9.7.2 Complexity Evaluation of the Backlog and Interference Analysis

To evaluate the complexity of the backlog and interference analysis, we generated synthetic systems by varying the system parameters n , m , and \bar{T} , while fixing \bar{U} . That is, each system generated is composed of n jobs with the same execution time distribution of length m and mean interarrival time \bar{T} . The shapes of the execution time distribution and the interarrival time distribution of the jobs are determined in such a way that the fixed average system utilization is maintained, even if they have no influence on the complexity of the backlog and interference analysis. (Recall that the backlog and interference analysis time is not affected by the actual values of the probabilities composing the distributions; the probability values may only affect the analysis time of the stationary system backlog distribution by changing the average system utilization \bar{U} .) We do not have to specify the interference degree k at the synthetic system generation stage, since it can be arbitrarily set prior to interference analysis of the resulting system.

For each system generated we perform backlog and interference analysis, assuming a null backlog at the beginning of the analysis. For each of the n jobs we measure the time taken by backlog analysis and interference analysis separately. In this measurement, the backlog analysis time for the j th job is defined as the time taken by applying the convolve–shrink procedure from the first job J_1 (with the null backlog) to job J_j .

Figure 9.4 shows the backlog analysis time measured for each job J_j in seconds, while varying m and \bar{T} . Note that both the x-axis and the y-axis are in logarithmic scale. From this figure we can see that the backlog analysis time for each job increases in polynomial order $O(j^2 m^2)$, as analyzed in the previous section. However, note that, owing to the backlog dependencies, the backlog analysis for the j th job may be efficiently performed in a real system by reusing the result of the backlog analysis for some close preceding job J_i ($i < j$). So, the backlog analysis time for real jobs may be significantly lower than that expected from the figure. Moreover, also note that in the case where $\bar{T} = m$ the backlog analysis time slowly increases as the value of j increases, since the backlog distribution length rarely grows due to the large interarrival times of the jobs.

Figure 9.5a shows the interference analysis times measured for the 100th, 250th, 500th, and 1000th jobs in seconds, while only varying the interference degree k . Note that both the x-axis and the y-axis are

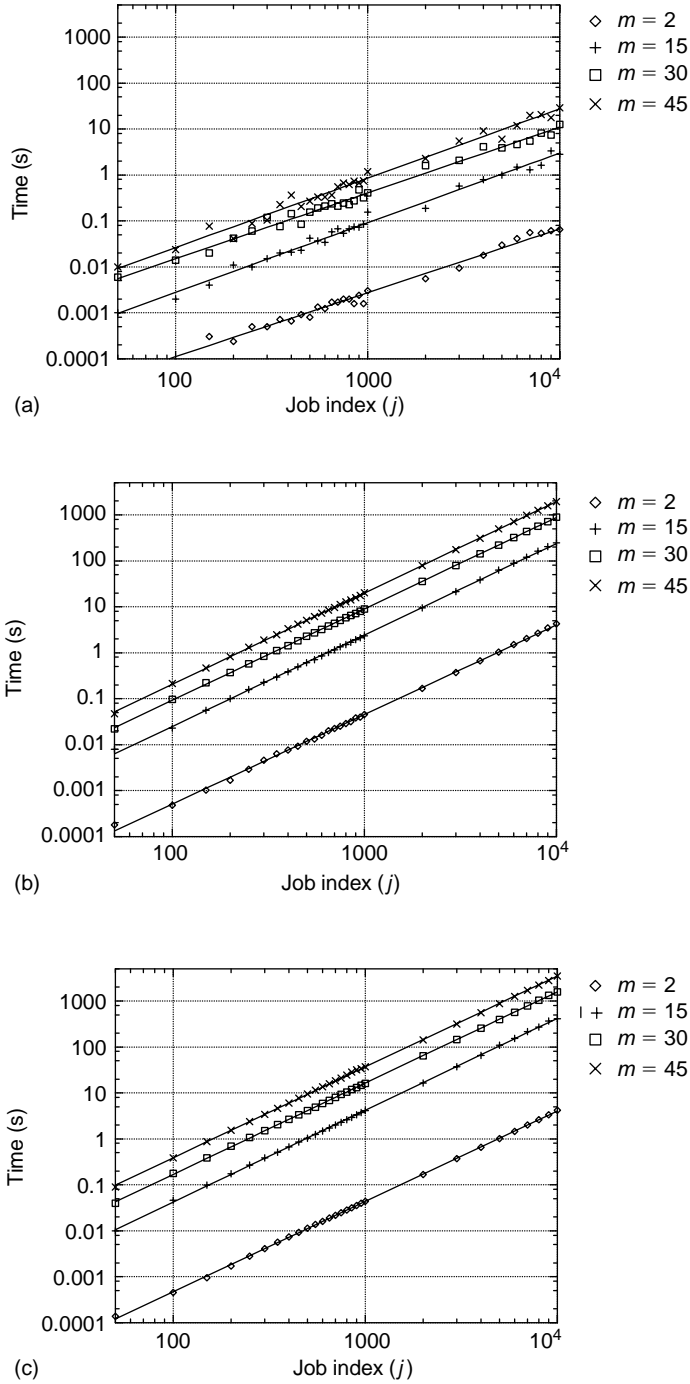


FIGURE 9.4 Backlog analysis time. (a) $\bar{T} = m$ ($U^{\max} \approx 1$), (b) $\bar{T} = m/2$ ($U^{\max} \approx 2$), and (c) $\bar{T} = m/10$ ($U^{\max} \approx 10$).

still in logarithmic scale. From this figure, we can see that the interference analysis time for a single job also increases in polynomial order $O(k^2 m^2)$ as the interference degree increases. Note, however, that the interference degree considered before the deadline is usually very small in practice. In contrast, Figure 9.5b shows the interference analysis times measured for each job J_j while fixing all the other system parameters.

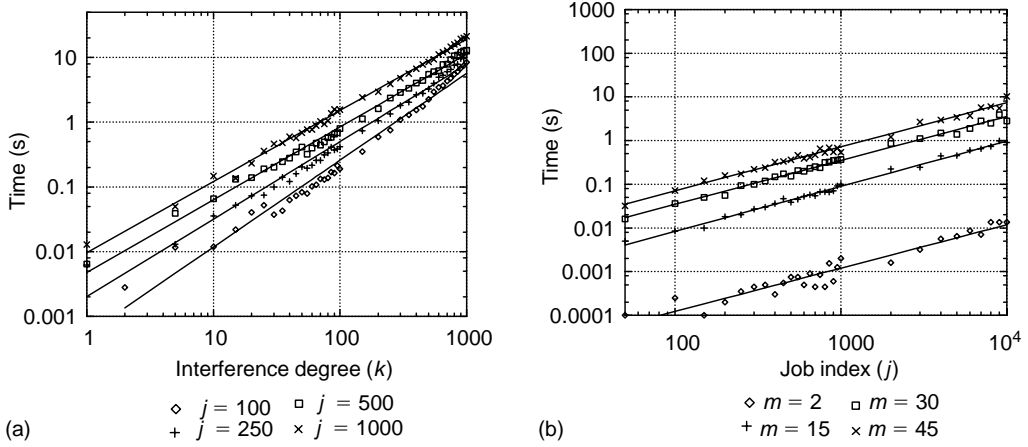


FIGURE 9.5 Interference analysis time. (a) effect of changes in the interference degree k ($m = 20$ and $\bar{T} = 1$) and (b) effect of changes in the length of the backlog distribution ($k = 10$ and $\bar{T} = 2$).

In this figure, we can indirectly see the effect of the length of the p_j -backlog distribution for the j th job to which the interference analysis is applied. As the p_j -backlog distribution length increases the interference analysis time also increases, but slowly.

9.8 Conclusions and Future Work

In this chapter, we have proposed a stochastic analysis framework to accurately compute the response time distributions of tasks for general priority-driven periodic real-time systems. We have shown that the proposed analysis framework can be uniformly applied to general priority-driven system including both fixed-priority systems such as RM and DM, and dynamic-priority systems such as EDF, by proving the backlog dependency relations between all the jobs in a hyperperiod. In our framework, the system is modeled as a Markov chain, and the stationary backlog distribution is computed by solving the Markov matrix, which is used as input to the formal structure encapsulating the backlog dependencies. This approach greatly reduces the complexity of the whole steady-state analysis. It has also been shown that the complexity of the exact method to compute the stationary backlog distribution and thus the response time distributions is $O(n^3 m^3)$, and that the approximation methods can significantly reduce the complexity without loss of accuracy, e.g., when the average system utilization \bar{U} is less than 0.8. For future work, we aim to develop a strategy to choose an appropriate value for the control parameters of the approximation methods, in particular, investigating the relationship between the system utilization and the convergence rate of the stationary backlog distribution.

References

1. L. Liu and J. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *Journal of ACM*, vol. 20, no. 1, pp. 46–61, 1973.
2. J. P. Lehoczky, L. Sha, and Y. Ding, The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, in *Proceedings of the 10th IEEE Real-Time Systems Symposium*, 1989.
3. J. P. Lehoczky, Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines, in *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pp. 201–209, 1990.

4. G. Bernat, A. Colin, and S. Petters, WCET Analysis of Probabilistic Hard Real-Time Systems, in *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002.
5. T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu, Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times, in *Proceedings of the Real-Time Technology and Applications Symposium*, pp. 164–173, 1995.
6. M. K. Gardner and J. W.-S. Liu, Analyzing Stochastic Fixed-Priority Real-Time Systems, in *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.
7. M. K. Gardner, Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems, Ph.D. dissertation, School of Computer Science, University of Illinois, 1999.
8. J. P. Lehoczky, Real-Time Queueing Theory, in *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 186–195, 1996.
9. J. P. Lehoczky, Real-Time Queueing Network Theory, in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 58–67, 1997.
10. L. Abeni and G. Buttazzo, Stochastic Analysis of a Reservation Based System, in *Proceedings of the 9th International Workshop on Parallel and Distributed Real-Time Systems*, 2001.
11. A. K. Atlas and A. Bestavros, Statistical Rate Monotonic Scheduling, in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 123–132, 1998.
12. J. Leung and J. M. Whitehead, On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks, *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.
13. S. Manolache, P. Eles, and Z. Peng, Memory and Time-Efficient Schedulability Analysis of Task Sets with Stochastic Execution Times, in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pp. 19–26, 2001.
14. A. Leulsegged and N. Nissanke, Probabilistic Analysis of Multi-processor Scheduling of Tasks with Uncertain Parameter, in *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003.
15. A. Terrasa and G. Bernat, Extracting Temporal Properties from Real-Time Systems by Automatic Tracing Analysis, in *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003.
16. J. W. S. Liu, *Real-Time Systems*, Prentice-Hall, Upper-Saddle River, NJ, USA, 2000.
17. J. L. Díaz, J. M. López, and D. F. García, Stochastic Analysis of the Steady-State Backlog in Periodic Real-Time Systems, Technical Report, Departamento de Informática, University of Oviedo, 2003, Also available at <http://www.atc.uniovi.es/research/SASS03.pdf>.
18. J. L. Díaz, D. F. García, K. Kim, C.-G. Lee, L. Lo Bello, J. M. López, S. L. Min, and O. Mirabella, Stochastic Analysis of Periodic Real-Time Systems, in *Proceedings of the 23rd Real-Time Systems Symposium*, pp. 289–300, 2002.
19. G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed., ser. Johns Hopkins Studies in the Mathematical Sciences, The Johns Hopkins University Press, Baltimore, MD, USA, 1996.
20. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed., Cambridge University Press, Cambridge, UK, 1992.
21. Intel Math Kernel Library: Reference Manual, 2001, <http://developer.intel.com/software/products/mkl>.

II

Programming Languages, Paradigms, and Analysis for Real-Time and Embedded Systems

10

Temporal Control in Real-Time Systems: Languages and Systems

Sebastian Fischmeister

University of Pennsylvania

Insup Lee

University of Pennsylvania

10.1	Introduction	10-1
10.2	The Model	10-3
10.3	The Example: A Stopwatch	10-4
10.4	Implicit Temporal Control	10-6
10.5	Programming with Temporal Control	10-7
	PEARL • Temporal Scopes • The ARTS Kernel • Real-Time Specification for Java • Esterel • Giotto	
10.6	Comparison and Conclusions	10-18

10.1 Introduction

Real-time application cannot rely only on predictable output values, but they must also have predictable timing tied to these outputs. Non-real-time applications, such as a spreadsheet program or a word processor, are insensitive to minor variances in timing but sensitive to performance. For such applications, it is acceptable that operations sometimes complete a little earlier or later as long as the overall performance is sufficiently fast. In real-time applications, such unbounded delays are unacceptable. Failure in the application's timing predictability is as unwanted by the stakeholders as failure in the application's output predictability. So, for example, when the application returns a correct result too late it could be as bad as if it would return a wrong result. Consider an electronic braking system built into a car where we replace the standard braking system with a completely electronic one. In this system, it is unacceptable for the driver if there is an unpredictable, sometimes even very long, delay between the driver hitting the brake pedal and the brakes applying force to the tires. This scenario makes it essential to think about temporal constraints in real-time applications.

Temporal constraints, placed on actions of the real-time application, specify the timing behavior of the application. Such constraints are often formulated as a range of possible time values or specified as upper bounds to the occurrence time of specific events. To ensure that these temporal constraints are met, the developer may need to analyze a chain of operations and specify the latest time (i.e., deadline) when each operation must complete. In the braking example, a temporal constraint on the braking action could be that the brakes have to apply force to the tires within 1 ms once the driver has hit the brake pedal. To guarantee that this temporal constraint is met, the developer will have to analyze and sum up the reaction time of the brake pedal level detection, the transmission time of the signal to the brake controllers, and

the reaction time of the disc brakes mounted on the tires. If worst case of this elapsed time is lower than the temporal constraint, the constraint is met.

Real-time computing and temporal constraints are not about performance but about timing precision. The range over which and the unit in which temporal constraints are specified is insignificant; it is the precision of the predictability that matters. Some real-time applications define temporal constraints with bounds in nanoseconds and some define constraints in seconds. For example, while the electronic braking system might require a temporal constraint of a millisecond, a patient ventilation machine, which pumps oxygen-enriched air into a patient's lungs during a surgery, has a much slower temporal constraint. The machine must ventilate the patient at a frequency of 12–14 times per minute with 5–6 ml/kg breathing volume. Here, the temporal constraint between two pump operations is roughly 5 s and it is relatively slow compared to the braking system.

Temporal constraints may or may not be met; however, a specific class of applications, called safety-critical real-time applications, requires that the temporal constraint must always be met. Such safety-critical real-time applications are systems whose failure or malfunction may lead to human losses. Since temporal behavior is critical, such applications have to be certified to always meet the temporal constraints before they can start operating. Two examples of safety-critical real-time applications have already been discussed: (1) if the braking system fails to meet the timing requirements, then a car crash and casualties may be the consequence and (2) if the ventilation machine fails to meet its timing requirements, then the patient might suffocate and die. Other examples are typically found in the domain of avionics, transportation, and medicine, such as air traffic control, drive-by-wire, fly-by-wire, and infusion pumps.

Temporal constraints exist in the forms of soft temporal constraints, hard temporal constraints, and firm temporal constraints. *Soft temporal constraints* are found in soft real-time systems. A soft real-time system is one in which the response time (i.e., a temporal constraint) is specified as an average value and the system does a best effort approach to meet these requirements. A soft temporal constraint specifies an average bound between two events, and a single violation of this constraint does not significantly matter, whereas many violations and high value deviations do. An example system with soft constraints is an airline reservation system. It is irrelevant if the reservation sometimes takes a little more time to book a reservation, as long as the booking operation's execution time is bounded reasonably. *Hard temporal constraints* are found in mission-critical real-time systems. A hard real-time system is a system where the response time is specified as an absolute value that is typically dictated by the environment. A hard temporal constraint specifies an absolute upper bound between two events. A single violation of such a constraint does matter as hard real-time applications are often safety-critical applications. An example of such a system is the electronic braking system mentioned above; if the braking system does not meet the constraint just once, this might have been the critical instant where meeting the constraint could have saved lives. *Firm temporal constraints* are found in firm real-time systems. Each constraint has a soft and a hard part consisting of an average bound (soft) and an absolute bound (hard), respectively. An example is the mentioned patient's ventilation system. Specifying the bound between two ventilation cycles, this timing constraint has a soft and a hard constraint. The soft constraint is 14 ventilations per minute, and the hard constraint is 12 ventilations per minute.

Temporal constraints can be absolute or relative. *Absolute temporal constraints* are measured with respect to a global clock, usually our wall clock. An example of such an absolute temporal constraint is that the electronic Christmas tree should light up between 5 p.m. and 2 a.m. each day from November 24th until December 27th. *Relative temporal constraints* are measures with respect to the occurrence times of specific events on the local clock. An example of such a relative temporal constraint is that the ventilation task should be activated 4 s after the last ventilation ended. Given an absolute or a relative temporal constraint, if the constraint does not hold, then a *timing violation* has happened. A timing violation occurs, for example, when the ventilation task is not rereleased within 4 s after the last ventilation ended, or if we specify that a computation should finish within 1 s and it does not.

A specific form of temporal control is time determinism. *Time determinism* defines that an external observer can predict the points in time, at which an application produces output (can also be called time/value determinism). This means that the observer can treat the system as a black box and predict correct values without knowing the application's internals. The points in time specifying the output times

can be given by a time range or by exactly one point in time. The smaller the range, the more precise the temporal control must be. For example, in the patient ventilation system, it is easier to predict the time range of the ventilation impulse to be $[0, 5)$ than to predict a single number, such as the impulse will occur exactly every $4.7 \text{ s} \pm 0 \text{ ns}$.

10.2 The Model

Temporal constraints always refer to events, where an event denotes the beginning or ending of a significant activity and is considered instantaneous. Temporal constraints can be used to specify the time bound between any two events. For example, a temporal constraint can specify the delay between the “driver-hits-the-pedal” event and the “brakes-apply-force-to-tires” event. For real-time tasks, common events to which temporal constraints are assigned are the task’s release or its completion event. A temporal constraint, bounding the time between two user-visible events, is often called an *end-to-end constraint* or an *end-to-end delay*. We now describe the life-cycle model of a task (cf., Figure 10.1), identify the most common events, and show how they are used to specify temporal constraints.

Figure 10.1 provides a state diagram of a task’s life cycle in our model. We follow the traditional approach and describe the model with tasks. However, note that the model is also applicable at the level of functions, single statements, and transactions. For example, temporal scopes provide such a model at the statement level (cf., Section 10.5.2). In our notation, the rounded box with a label in it denotes a state (e.g., the state *Ready*). Arrows between states define transitions. A transition can be labeled, it can have an enabling condition, and taking the transition can lead to updates of variables. The format is as follows: *label* [*condition*]/*updates*. For example, the transition between the state *Completed* and *Ready* in Figure 10.1 specifies that it is called *Release*. This transition has no enabling condition, and when it is taken, then the variable t is set to zero. The filled circle points to the initial state. As a shorthand notation, if a transition has an empty enabling condition (i.e., it is always enabled) and has an empty update clause (i.e., it does not update any variable) the empty construct $[]/$ is omitted. In Figure 10.1, the task starts its life cycle in the state *Completed*. When it must accomplish its action, the scheduler releases it, and the task enters the state *Ready*. Once the scheduler assigns computing resources to the task, the task enters the state *Running*. However, the scheduler can suspend the task and can change its state to *Suspended*. Then, the scheduler can resume the task and it changes back to state *Running*. Some systems also allow the developer to actively suspend and resume tasks via program instructions. Once the task has completed its activity, it stops using computing resources and changes to the state *Completed*. The task resides in this state until it is released again. Note that there may be other events for the operating system tasks; however, they are not of particular interest here.

This life-cycle model describes a number of events:

- *Release event*. It marks the time at which the task enters the system and the scheduler adds it to its ready queue.

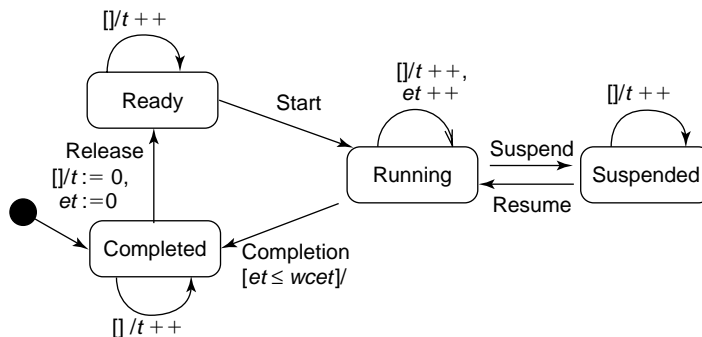


FIGURE 10.1 Task life cycle.

- *Start event.* It marks the time at which the task starts computing for the first time since its release event.
- *Suspend event.* It marks the time at which the scheduler removes the task's access to computing resources. The suspended task cannot continue computing.
- *Resume event.* It marks the time at which the scheduler gives the task access to computing resources again so that the task can resume its activity.
- *Completion event.* It marks the time at which the task finished its activity and indicates that the task no longer requires computing resources.

In addition to the events described in the model, the task and the environment can trigger application-specific events. Figure 10.1 does not show these events, because they differ for each application. For example, the task can sense an emergency shutdown button. If someone pushes this button, the event “button-pushed” will be created in the environment. The task can sense this event and respond to it by firing an application-specific event such as “perform-emergency-shutdown.”

To express timing constraints, the state diagram in Figure 10.1 includes the variables t and et . The variable t measures the elapsed time since the release event. The variable et measures the execution time that the task has consumed computing resources. The constant $wcet$ defines the task's worst-case execution time, that is, the maximum amount of time that the task needs to consume computing resources to complete its activity. Note that the notation used in Figure 10.1 is rather limited in its expressiveness. We could have used other formalisms, such as timed automata, real-time process algebra, and timed Petri net. Using the examples however, would be unnecessarily complicated, so we use the model shown in Figure 10.1 throughout this chapter and adapt it to the capabilities of the programming language under discussion.

Temporal constraints can be specified in different forms and notations, but essentially they define a bounded time between two events. A timing constraint can be tied to a one-time event in the lifetime of the system or to recurring events. An instance of a one-time event is the emergency shutdown described above. A temporal constraint can define a time bound between the event “button-pushed” and “perform-emergency-shutdown.” Since the system is off, after it has performed the shutdown, we consider this as a one-time event. Recurring events in our model are caused by repetitive computational activities getting rereleased once they are completed.

Real-time theory distinguishes three different recurrence types: periodic, sporadic, and aperiodic. A *periodic recurrence* produces release events at regular time intervals. It is possible to define a temporal constraint between two release events as time units. This value is also called the task's *period*. Periodic recurrences can be specified in different notations. In scheduling theory, they are usually specified by the period P , where P time units pass between two occurrences. For example, the patient ventilation pump has a period $P = 5$ s. Another way to specify a temporal constraint is to define a period P' and a frequency f . For the same behavior, one can specify that the ventilation event has a period $P' = 60$ s and a frequency $f = 12$. Yet another way is to specify just the frequency in hertz f' ; the pump has a frequency $f' = \frac{1}{5}$ Hz. Note that all these notations are compatible and one notation can be converted into the other. A *sporadic recurrence* produces release events with irregular but bounded time intervals. It is possible to specify a temporal constraint between two such events as an interval with a lower limit and an upper limit. For example, using the patient ventilation pump as a firm real-time system, the temporal constraint between two release events is bounded by $[4.28 \text{ s}, 5 \text{ s})$. This means that the elapsed time between two release events must be more than 4.28 s and less than 5 s. Finally, an *aperiodic recurrence* produces irregular release events and no meaningful temporal constraint can be placed on them.

10.3 The Example: A Stopwatch

We use the example of a software-controlled digital stopwatch to explain the different programming languages and their support for temporal control throughout this chapter. The stopwatch is appropriate, because it requires tight control of timing and needs virtually no program logic. The stopwatch has been

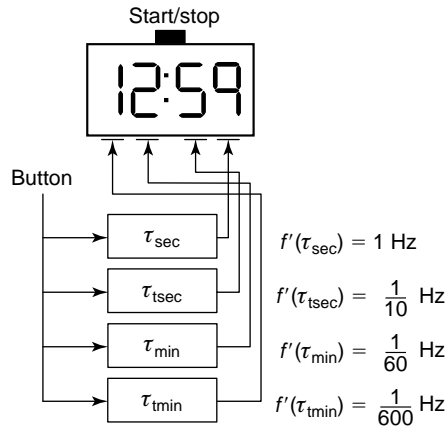


FIGURE 10.2 The software-controlled digital stopwatch.

used commonly in the literature as a guiding example to demonstrate different aspects of value and timing behavior in real-time applications (cf., Refs. 1–3).

The goal is to implement a software-controlled digital stopwatch with one start/stop button. The stopwatch is to behave as follows. When the user presses the start/stop button, the clock value is reset and it starts running. When the user presses the start/stop button again, the clock stops running. At any time, the stopwatch's display shows the clock's value. Additionally, the clock should exhibit predictable value update behavior; so, we require the common update sequence of starting from the one column and continuing to the hour column (e.g., 12:59 to 12:50 to 12:00 to 13:00).

Figure 10.2 shows an overview of the application. It consists of four tasks and each controls an individual digit of the stopwatch's display. Task τ_{sec} controls the seconds' ones column and runs at a frequency of 1 Hz. Task τ_{tsec} controls the seconds' tens column and runs at a frequency of $\frac{1}{10}$ Hz. Task τ_{min} controls the minutes' ones column and runs at a frequency of $\frac{1}{60}$ Hz. And finally, task τ_{tmin} controls the minutes' tens column and runs at a frequency of $\frac{1}{600}$ Hz. Each task receives the start/stop button's status as input. The functionality of the tasks τ_{sec} and τ_{min} is to output *invocations mod 10*. The value of variable *invocations* is the number of times each task has been released. Since the one column's value range of seconds and minutes is between 0 and 9, the function calculates the number of invocations modulo ten. The functionality of the tasks τ_{tsec} and τ_{tmin} is similar, these two tasks output *invocations mod 6*. Note that the intent of this design is to demonstrate the differences among the programming languages and systems reviewed in this chapter.

The difficulty in this clock's design is that it requires rigorous control of the tasks' timing and the value behavior. Figure 10.2 shows the stopwatch 779 s after pushing the start/stop button. It shows the value of 12 min and 59 s. So, the tasks' outputs are τ_{sec} outputs the value 9, τ_{tsec} outputs the value 5, τ_{min} outputs the value 2, and task τ_{tmin} outputs the value 1. We now highlight two potential problems, which have to be addressed when implementing this design:

- *Jitter accumulates and causes incorrect display.* If the programming language and the runtime system introduce jitter for each task invocation, then in some systems this jitter will accumulate over time and the display will eventually show incorrect values. Additionally, if each task manages its release time itself, then the stopwatch can behave incorrectly. For example, suppose that the task τ_{tsec} has updated its value from 5 to 0, but the task τ_{min} is late and has not been released yet. Here, the display value might change from 12:59 to 12:09.
- *Different causal ordering of events and tasks causes different behavior.* The four tasks run concurrently. Every 10 min, all four of them are released. If the computer has only one processor, then the tasks can only complete sequentially. If the runtime system does not provide means for controlling the ordering or the outputs' update behavior, then the display might disobey the specified update behavior.

For example, if the order of the completion events is τ_{sec} , τ_{min} , and τ_{tsec} , then the display values will change in this order: starting with 12:59 to 12:50 to 13:50 to 13:00. The typical order is to update from the rightmost column to the leftmost column: starting with 12:59 to 12:50 to 12:00 to 13:00.

Another problem is to implement precise semantics for the effect of pushing the button. For example, do we start measuring the time after the start/stop button has been pressed or at the time it is pressed. We will elaborate on this problem when we discuss the synchronous languages, which target exactly this problem.

10.4 Implicit Temporal Control

Small systems often lack explicit temporal control and thus temporal control is programmed using hand-coded delay blocks. We call such means *implicit temporal control*, because it is embedded in the program source and does not use high-level language constructs. A widespread method is the background/foreground systems. In a *background/foreground system*, the application consists of exactly one loop without an exit condition. Within this loop, the application calls subroutines in a sequential order, which implements the application's logic. The loop's execution time essentially determines the application's temporal behavior. The loop is commonly referred to as the background. If an interrupt occurs, an interrupt service routine (ISR) preempts (suspends) the loop and services the interrupt. The ISR is commonly referred to as foreground; hence the name background/foreground system.

Figure 10.3 shows the foreground/background system's life cycle without nested interrupts. The application typically spends time in the background part, executing the main loop. When an interrupt occurs, the system switches to the foreground and the ISR services the interrupt. Once the ISR completes, the system switches back to the background operation and resumes the main loop.

The main application domain of background/foreground systems is small embedded systems, such as washers, dryers, microwave ovens, and simple radios. Compared to multithreaded systems with explicit temporal control, background/foreground systems require less system overhead and less understanding of concurrency and temporal control. However, the low system overhead has a price. The application's output is nondeterministic with respect to the timing. The points in time at which the application produces an output changes depending on the application's execution path for each run in the loop and how many and what types of interrupts occur. The application's timing is also sensitive to modifications to the loop. For example, one additional inner loop in the main loop changes the timing behavior of everything that follows after this inner loop. Such a change can alter the whole system's behavior. Also, modifying the ISR changes the timing behavior depending on how often the ISR preempts the main loop.

In this system, the stopwatch example cannot be implemented with multiple tasks, because a background/foreground system does not provide multitasking. So, we have to resort to the means of the

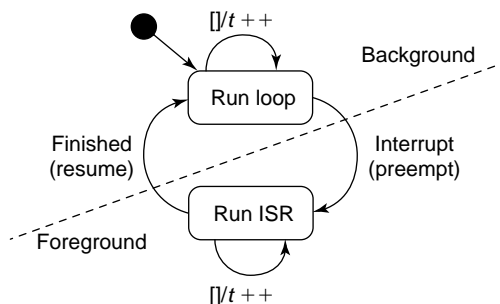


FIGURE 10.3 Foreground/background system's life cycle.

```

1 void main(void) {
    unsigned short val;
    unsigned int i;

    while ( 1 ) {
6      val = get_curr_sec();
        val++;
        update_display_seconds(val);

        if (val%60 == 0) {
11         // update tens
        }
        ...

        // may have nested loops, if too short
16      i=WHILE_INSTRUCTIONS_PER_SECOND;
        while( --i );
    }
}

```

LISTING 10.1 The task τ_{sec} with the superloop.

superloop to implement it. Listing 10.1 gives an idea on how the stopwatch can be implemented. The superloop starts at Line 5, and it updates one column of the watch's display after the other: First the seconds, then the tens of seconds, then minutes, and finally tens of minutes. At the end of outer loop, we place a tight loop to wait for the next second. Line 16 assigns to i the value that results in waiting for precisely 1 s in Line 17. The faster the processor, the higher will be this number. If the value range of i is too small, then nested loops can be used. After the tight loop at Line 17 terminates, the superloop will start all over again. The temporal behavior is controlled via conditional branches (cf., Line 10) that control how often specific function blocks are executed. For example, if the loop iterates once every second and the conditional branch specifies that it is executed only every 60th time, then the function blocks inside the branch will be executed once every minute.

This example implements the desired updated behavior; however, this implementation is prone to jitter and the displayed time is unequal to the measured time. The reason is that over time the displayed time diverts from the real time, because of the variance in the superloop's execution time. Even if we assume that the constant in Line 16 takes the execution time into consideration, every 10th invocation, when the tens of seconds column gets updated, the runtime differs from the other nine invocations. The same happens every 60th and 3600th invocation. Over time, jitter accumulates and results in an incorrectly measured time. Note that a stopwatch can be implemented with precise timing using a background/foreground system; however, it requires a more elaborate setup with interrupts and system timers. Real-time programming languages and systems provide means for this right in the language or the system.

10.5 Programming with Temporal Control

Different programming languages and systems allow specifying temporal constraints in different ways, as some systems support specification-based temporal constraints and others support program-based ones. *Specification-based* temporal constraints are defined by generating temporal behavior from annotated source code or some high-level specification. Specification-based approaches require additional support from the compiler, because the specification has to be translated into a code in addition to the functionality code. For example, assume that the ventilation task is responsible for generating the patient

ventilation impulse and it executes once every 5 s. In a specification-based approach, the developer annotates the task with something similar to “`freq 0.2`,” and the compiler translates this annotation and generates a code, which executes the ventilation task with the specified frequency. *Program-based* temporal constraints are programmed in the target programming language without annotations. This approach is quite similar to the implicit temporal constraints shown before, however, in program-based temporal constraints the programming language usually provides an interface for programming temporal constraints; so, no `while(--i);` is necessary. Some programming languages provide more sophisticated support than others. The least common denominator of these interfaces is a function to read the wall clock time and a function to delay execution for a specified amount of time. To program the ventilation task, the developer executes the task in an endless loop. At the loop’s end, the task enters a statement corresponding to `delay(5 s - ϵ)`, where ϵ compensates for the execution time of the ventilation-control function. The ventilation task will then run at least once every 5 s.

In the following sections, we present a number of programming languages and systems, which provide built-in means of temporal control either on the specification or the programming level.

10.5.1 PEARL

The programming language PEARL, an acronym for *Process and Experiment Automation Real-time Language*, is a high-level programming language with elaborate constructs for programming temporal constraints. The language itself has been standardized as DIN standard 66253 [4]. PEARL is tailored to implement time-sensitive systems, and it includes packages for input/output, multitasking, synchronization, interrupt handling, and signaling. For this chapter, we concentrate on the support for specifying temporal constraints on tasks. In this chapter, we use the revision PEARL-90 [5]. PEARL-90 is a reworked version of the original PEARL language, and in the remainder of the text, we always refer to PEARL-90 when we write PEARL.

10.5.1.1 The Model

PEARL provides temporal control for releasing and resuming tasks. Figure 10.4 shows the time-triggered parts of the task’s life cycle. PEARL also allows the developer to specify event-triggered constraints on the life cycle as well; however, they are unrelated to timing and thus they are not discussed in this chapter. Each task starts by entering the state *Completed*. PEARL allows specifying absolute and relative temporal constraints. The variable t_{abs} realizes an absolute timer that is reset whenever the system is started. The variable t_{rel} realizes a timer for relative temporal constraints. PEARL allows placing absolute and relative constraints on the task’s release. So, for the task to be released, the constraint guarding the transition from the state *Completed* to the state *Ready* must be satisfied. The variable t_{AT} represents the timing

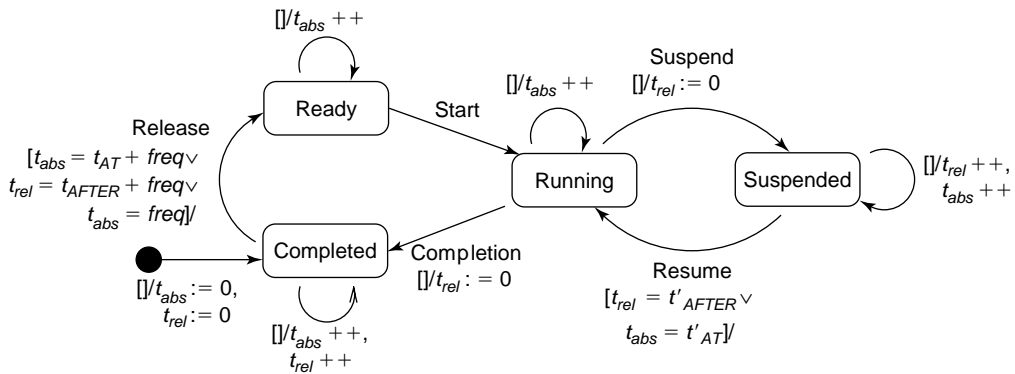


FIGURE 10.4 Task life cycle in PEARL without the event-triggered parts.

information specified with PEARL's AT statement, which is used to declare absolute temporal constraints such as `AT 12:00`. The variable t_{AFTER} represents the timing information specified with PEARL's AFTER statement, which is used in relative temporal constraints such as `AFTER 2 sec`. PEARL also supports specifying task periods shown as *freq* in Figure 10.4. The frequency determines at what time the task is released again. The task's recurrence can also be limited to a relative or absolute time span. So, for example, the developer can specify that a task runs every 5 s for the next 10 min or between 1 p.m. and 2 p.m. Once the task is released, it enters the state *Ready* and eventually the state *Running*. A task may be suspended and it enters the state *Suspended*. PEARL allows the developer to place absolute and relative temporal constraints on the transition back to the state *Running*. Note that t_{AFTER} on transition *Release* and t'_{AFTER} on transition *Resume* differ. t_{AFTER} is specified for the release event and t'_{AFTER} is specified for the resume event. The same is true for AT. For example, the developer can program that a task is to be continued at 1 p.m. (after the lunch break), or after 10 min measured from the task's suspension to realize a cool-down phase. Eventually, the task will complete and enter the state *Completed*.

10.5.1.2 The Stopwatch

Listing 10.2 shows part of the complete PEARL program that implements the stopwatch. For details on the syntax see Ref. 4. Specifically, the program shows the activation code for all four tasks and the complete source of one of them. The first four lines declare the temporal constraints on the individual tasks. The tasks controlling each clock column are activated depending on how often they must update their values. For example, the task updating the clock's 1-s column is activated once every second. The second part of the listing declares and implements task `clock_sec`. The task uses one variable called *ctr*. In the task's body, we first invoke a method to get the clock's current value of the 1-s column. We then increment the value by 1 and finally write it back to the clock.

PEARL allows the developer to assign priorities to the individual tasks. In Listing 10.2, task `clock_tsec` has the priority 2 (i.e., the second highest one). The other tasks have different priorities according to our intended order of invocation to update the clock display. So, when it is 12 o'clock noon, first the task `clock_sec` will update the clock value, followed by `clock_tsec`, `clock_min`, and `clock_tmin`. This way, we implement the update behavior as specified in Section 10.3.

10.5.2 Temporal Scopes

The notion of temporal scopes has been introduced with the Distributed Programming System (DPS) [6]. A temporal scope is a language construct that can be used to specify the timing constraints of code execution and interprocess communication. Such scopes allow the developer to specify timing constraints down to the statement level and register exception handlers to cope with timing violations. Other goals

```

1 WHEN start ALL 1 sec UNTIL stop ACTIVATE clock_sec;
  WHEN start ALL 10 sec UNTIL stop ACTIVATE clock_tsec;
  WHEN start ALL 60 sec UNTIL stop ACTIVATE clock_min;
  WHEN start ALL 600 sec UNTIL stop ACTIVATE clock_tmin;

6 clock_tsec: TASK PRIO 2;
  DCL ctr INTEGER;
  BEGIN
    GET ctr FROM DISPLAY_T_ONES;
    ctr := (ctr+1)%6;
11 PUT ctr TO DISPLAY_T_ONES;
  END

```

LISTING 10.2 The task τ_{tsec} as `clock_tsec` in PEARL.

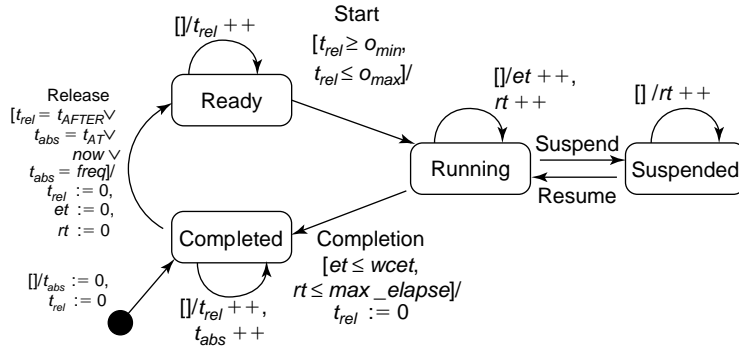


FIGURE 10.5 A temporal scope's life cycle.

of DPS are that the resulting system should be apt for distributed systems, temporal scopes should be embedded into an existing language, and the system should support runtime monitoring and exception handling. Temporal scopes are a specification-based approach. The developer annotates C source code with temporal scopes and specifies its temporal behavior.

10.5.2.1 The Scope's Model

Figure 10.5 shows the temporal scope's life cycle. The variables t_{abs} and t_{rel} are the same as that used before. The new constraints that are not present in the basic model (cf., Figure 10.1) are a minimum delay, a maximum delay, a maximum elapse time, and absolute and relative constraints on the release. Temporal scopes allow placing constraints on the task's release similar to PEARL. Additionally, they use the keyword *now*, which means an immediate release. The *minimum delay* (o_{min}) specifies a minimum amount of time that should pass before starting the execution of a temporal scope. So, the condition $t \geq o_{min}$ guards the transition from the state *Ready* to the state *Running* and inhibits a task from starting computing before sufficient time has passed. The *maximum delay* (o_{max}) specifies the maximum amount of time that could pass before starting the temporal scope's execution. As shown in the state diagram, the condition $t \leq o_{max}$ guards the transition from the state *Ready* to the state *Running*. This condition provides an upper bound for the temporal scope until it starts computing. The *maximum elapse time* specifies the maximum amount of time (max_elapse) that the temporal scope requires until completion once it starts computing. The state diagram in Figure 10.5 uses the variable rt to keep track of the amount of time that the task resides in the states *Running* and *Suspended*. The condition $rt \leq max_elapse$ limits the amount of time that the temporal scope can spend in the states *Running* and *Suspended*. In contrast to the execution time, the maximum elapse time considers the duration for which the task is suspended. By that, the maximum elapse time combines the execution time and all additional system overhead resulting from suspensions.

10.5.2.2 The Stopwatch

There are three types of temporal scopes: the sequential scope, the repetitive scope, and the consecutive scope. For the stopwatch example, we will only introduce the repetitive scope. The other types are elaborately described in Ref. 6. Listing 10.3 specifies the repetitive temporal scope's syntax.

Here the developer specifies the time interval $\langle start_time \rangle$ to $\langle end_time \rangle$ in which the scope will be repetitively executed at every interval specified in $\langle period \rangle$. For each repetition, he can also specify its worst-case execution time in $\langle e_time \rangle$ and the deadline in $\langle deadline \rangle$. For the periodic scope, the maximum elapsed time of a period is considered to be equal to its $\langle period \rangle$. This fully specifies the temporal behavior as denoted in Figure 10.5. Then, the developer can code the scope's logic (e.g., in C as implemented in DPS) and he/she can specify exception handlers to handle errors caused by timing-constraint violations.

```

from <start-time> to <end-time> every <period> execute <e-time> within <deadline>
do
3  <stmts>
    [<exceptions>]
end

```

LISTING 10.3 The basic temporal scope's syntax.

```

from 00:00 to 59:59 every 10 s execute 20 ms within 1s
do
    var ctr;
    ctr=get_cur_tsecs();
5  ctr=(ctr+1)%6;
    set_cur_tsecs(ctr);

    exception
        display_warning_light();
10 end

```

LISTING 10.4 The task τ_{tsec} using temporal scopes.

The stopwatch consists of four independent tasks. Listing 10.4 shows the pseudocode program for task τ_{tsec} using a repetitive temporal scopes. The scope is defined to run from 0 up to 1 h. During this time, it executes the <stmts> once every 10 s and within 1 s. One execution should take at most 20 ms. In the body of the scope, it reads the current value of the second seconds column and updates it accordingly. If a timing violation occurs, then the program will show a warning light on the stopwatch. Similar code can be written for the other three tasks. All four temporal scopes then run concurrently and update the stopwatch's display.

Temporal scopes concentrate on placing temporal constraints on statement blocks. The language leaves interscope control such as synchronization and prioritization to the programming language using temporal scopes. Therefore, the developer cannot specify an order in which concurrent temporal scopes are executed, and it is impossible to implement the update behavior as specified in Section 10.3 solely with temporal scopes. For example, at 12 o'clock it is up to the scheduler in what order it executes the tasks and it might choose an order that does not comply with the specification. Also, the task releases are outside the scope of temporal scopes, so the application logic for when the start or the stop button is pressed has to be specified outside the temporal scope.

10.5.3 The ARTS Kernel

The ARTS kernel [7] allows specifying temporal constraints similar to temporal scopes. The kernel uses the temporal constraints to implement its time-fence protocol which guarantees that the temporal constraints of the active tasks are met. Such a time fence is an annotation to the C++ code containing the functionality and can be associated with functions and threads. The kernel's time-fence protocol can then detect timing-constraint violations as it compares the specified time fence with the actual runtime behavior.

10.5.3.1 The Model

The ARTS kernel offers a different life-cycle model for threads and functions. In the following text, we explain both models by showing how it differs from the basic model in Figure 10.1. The variable t_{rel} implements a relative timer similarly to how we used it in the previous models. Figure 10.6 shows thread's life cycle. The developer can place temporal constraints on a thread by specifying a phase, an offset, a delay, a worst-case execution time, and a period. The phase is an initial offset that delays the transition from the state *Init* to the state *Released*. The offset guards the transition *Release*, and it allows to specify an offset between the thread's period and its rerelease. The delay guards the transition from the state *Ready* to the state *Running*, and it specifies a minimum delay until the thread starts consuming resources, similar to temporal scopes. The worst-case execution time specifies an upper bound on how much time can pass until the thread enters the state *Completed*. If the worst-case execution time is not met then a timing violation will occur. Eventually, the thread completes and waits in the state *Completed* until its period is due and it can get rereleased.

Figure 10.7 shows the life-cycle model for functions in threads. The developer can only place a temporal constraint about the maximum elapse time on functions. The maximum elapse time guards the transition from the state *Running* to the state *Completed* and the function can only enter the state *Completed*, if no more than max_elapse time units have passed since it entered the running state. Note the semantic difference in the maximum elapse time between temporal scopes and the ARTS kernel. In temporal scopes, the maximum elapse time starts when the scope enters the state *Running*, whereas in the ARTS kernel, clock starts ticking when the function enters the state *Ready*. The reason for this is that the ARTS kernel prohibits specifying minimum and maximum delays for functions.

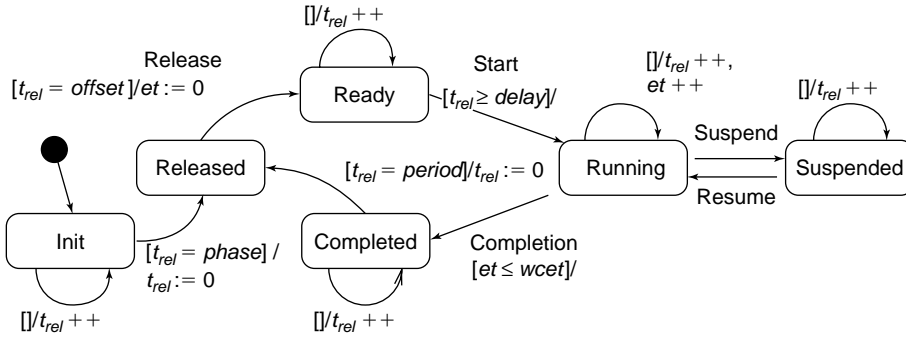


FIGURE 10.6 Thread life cycle in the ARTS kernel.

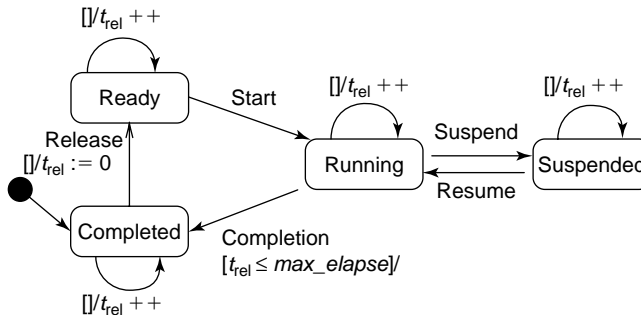


FIGURE 10.7 Function life cycle in the ARTS kernel.

10.5.3.2 The Stopwatch

The ARTS kernel provides specification-based support for temporal control. The developer annotates the C++ source code and writes values for the worst-case execution time, the period, the phase, and the delay next to threads and the maximum elapse time next to functions. The ARTS/C++ compiler introduces the token “//#,” which marks the beginning of the ARTS-related temporal specification. The constraint is specified as //# <priority>, <stack_size>, <wcet>, <period>, <phase>, <delay>.

Listing 10.5 implements the task τ_{tsec} of the stopwatch in the ARTS kernel. The task is implemented as a thread. The symbol “_” represents omitted annotation parameters, which are not relevant for the temporal control. As specified in the listing, it has a worst-case execution time of 10 ms, a period of 10 s, a phase of zero, and a delay of 0 s. In the complete stopwatch example, all four threads are implemented like this but with different periods.

To implement the specified update behavior on clock’s display, we use priorities explained in Section 10.5.1. So, the thread implementing the seconds’ one column has the highest priority and the thread implementing the minutes’ ten column has the lowest one.

10.5.4 Real-Time Specification for Java

Java became a popular programming language during the last decade. To support its use in real-time systems, a committee was formed to add real-time capabilities to Java. The result is the real-time specification for Java (RTSJ). It adds facilities for real-time systems as well as temporal control at the thread level. The goals of RTSJ are to fit real-time programming into the typical Java programming experience: it should be backward compatible, have no syntactic extensions, and fulfill the “write once, run everywhere” mantra. Consequently, RTSJ offers program-based temporal control, in which the developer must specify temporal behavior as program instructions. And, RTSJ offers means for temporal control only at the thread level.

10.5.4.1 The Model

RTSJ allows the developer to specify periodic, aperiodic, and sporadic tasks. In the following text, we discuss only RTSJ’s support for temporal control of periodic tasks (cf., Figure 10.8). The developer can specify an initial offset o_{init} , task’s period, its deadline relative to the period, and its worst-case execution time. The offset is a relative time and guards the transition from the state *Init* to the first release. After the task has been released once, the offset is irrelevant. Task’s period guards the transition from the state *Completed* to the state *Released*. The task will be rereleased once the time value equals its period. The developer can also specify a deadline dl relative to task’s period. This condition, $t \leq dl$, guards the transition from the state *Running* to the state *Completed* and requires the task to finish computation prior to that time. Finally, RTSJ also allows specifying the worst-case execution time. In RTSJ this value is called *cost*. It also guards the transition from the state *Running* to the state *Completed* and limits the time in which the task can reside in the state *Running*.

```
Thread Minutes :: RT_Thread( ) // # 2, _, 10 ms, 10 s, 0, 0 s
{
    // thread body
    int tens_seconds = get_cur_tens_seconds();
5   tens_seconds = (tens_seconds + 1) % 6;
    set_cur_seconds(tens_seconds);

    ThreadExit(); //reincarnate this thread
}
```

LISTING 10.5 Task τ_{tsec} using the ARTS kernel.

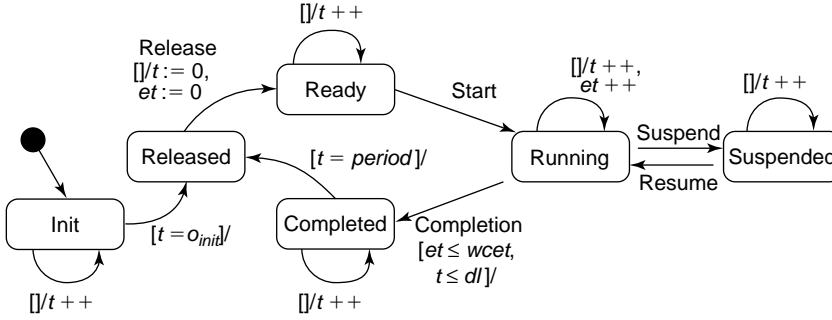


FIGURE 10.8 Task life cycle in RTSJ.

```

1 public class TSec extends RealTimeThread {
    ...

    public void run() {
        while ( true ) {
6         int val = getCurrentTSecValue();
            val = (val+1)%6;
            setCurrentTSecValue(val);
            waitForNextPeriod();
        }
11 }

    TMin createInstance() {
        ...
        PeriodicParameters pp = new PeriodicParameters(offset,
16         new RelativeTime(10.0*SECONDS), // the period
            new RelativeTime(5.0), // the cost
            new RelativeTime(10.0*SECONDS), // the deadline
            null, null);

21     return new TSec(priority, pp);
    }
}

```

LISTING 10.6 Task τ_{tsec} in RTSJ.

10.5.4.2 The Stopwatch

RTSJ is an object-oriented language and provides a program-based approach to specifying temporal constraints. Its programming interface for temporal control is encapsulated in an object named `ReleaseParameters`. Temporal-control parameters for periodic tasks are grouped in the subclass `PeriodicParameters`. When instantiating a new thread, the developer must pass release parameters to the thread's constructor. In the following, we implement stopwatch's task τ_{tmin} .

Listing 10.6 implements task τ_{tsec} in a class called `TSec`. Lines 5–9 specify task's functionality where the counter gets incremented by one and as a minute has at most 60 s, its output value is calculated modulo six. The method called `waitForNextPeriod()` in Line 9 declares that this task has completed and it returns control to the scheduler. Since temporal control in RTSJ is program-based, we also must add this code in the source code. Lines 15–19 specify the task's temporal behavior using the class

`PeriodicParameters`. The variable `offset` specifies the model's offset o_{init} . The next parameters specify task's period, its worst-case execution time, and its deadline. In our example, we set the deadline equal to task's period. The remaining two parameters are for error handling and are irrelevant here. Finally, we instantiate a new thread and return it as the result of the method `createInstance()`. This thread can then be put into the state *Init* by calling thread's `start()` method.

10.5.5 Esterel

The Esterel language [2] is an example for a synchronous language [1]. A synchronous language bases its model of time on the synchrony hypothesis, which assumes that the underlying machine is infinitely fast and program outputs become available at the same time instant as inputs are supplied. This hypothesis essentially reduces actions to events, and it allows Esterel to define precise temporal constraints. One notable difference to the other systems present before is that one of Esterel's goals is to support the verification of temporal properties. Its language semantics are mathematically formalized and, leveraging this, the developer can precisely predict the temporal behavior of applications written in Esterel.

10.5.5.1 The Model

Esterel's life-cycle model is trivial, because every reaction is reduced to an instant (Figure 10.9). As every task completes instantaneously, task's life cycle includes only the states *Completed* and *Running*. When a task switches from the state *Completed* to the state *Running*, it instantaneously (i.e., the measured time t equals zero) completes and switches back to the state *Completed*.

The synchrony hypothesis shifts the focus of controlling temporal behavior from inside tasks to in-between tasks. Another consequence of the synchrony hypothesis for temporal control is that in synchronous languages it is unnecessary to control the temporal behavior inside the task. Computation of tasks is reduced to an event and multiple tasks cannot interfere with each other, because all of them complete instantaneously; so, temporal control inside the task becomes unnecessary. The synchrony hypothesis allows the developer to concentrate on the temporal behavior in between tasks; he/she concentrates on what point in time the release events of individual tasks will occur. The language provides several constructs to specify temporal constraints between events. The smallest time unit is one instant, and Esterel language uses the signal `tick` to separate two instants. Since the computation is reduced to one instant, the developer can specify precisely whether, for example, a follow-up task starts (and completes) in the same instant or in the next instant. This type of temporal control is more precise in terms of semantics than the other approaches.

10.5.5.2 The Stopwatch

The Esterel language provides constructs to control the emission of events (called *signals*) and checks for their presence. For the stopwatch example, we show two possible implementations of task τ_{sec} and discuss the temporal behavior of each. We also explain the used language constructs as we explain the example. The example uses only a subset of the language constructs available. For a full list see Ref. 2.

Listing 10.7 implements the task τ_{sec} . The listing shows only the functionality part of the task and lacks the module and other declarations encapsulating it. We assume that the signal *S* becomes present every 10 s, the valued signal *TIME* is connected to the display, and the signals *STOP* and *START* occur when

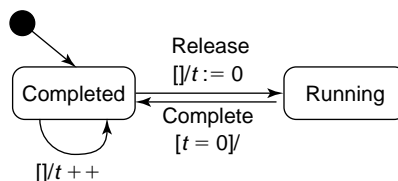


FIGURE 10.9 Esterel's life-cycle model.

stopwatch's start/stop buttons are pressed. The language construct `await` does nothing and terminates when the specified signal is present. The keyword `immediate` further refines the construct and specifies that it also checks the current instant. So, Line 1 waits until the start button *is* pressed. Note the emphasis on the expression “is pressed,” because the program continues within the same time instant. So, the `tick` is not over yet. The `abort . . . when` presents a preemption mechanism. Without the preemption condition, the statement runs to completion. If the condition ever holds, the inner statement block terminates immediately within the same time instant. This inner statement block, from Line 4 to Line 7, increments second's ten column every 10 s. Similar to other programs already shown, we have to compute the display value modulo six to show values between and including 0 and 5 for the 10 digit. At the instant, in which the stop button *is* pressed, the program stops incrementing the counter and just continually displays the last value.

As stated before, Esterel allows precise temporal control down to the level of individual time instants. In the stopwatch example this means: Does the stopwatch measure the time between pressing the start and the stop button, or does it measure the time from the moment the start button is pressed until the stop button has been pressed? It is similar to the ways that one can specify ranges with the excluding brackets `(,)` and the including brackets `[,]`. Listing 10.8 also implements the stopwatch, however, this time with different temporal behavior. The construct `weak abort` allows the body to run before it terminates. In the second implementation in Listing 10.8, the program increments the counter at the time instant the stop button *has been* pressed, and then continually displays the value; so, it adds one more time instant compared to the previous listing.

Esterel does not support specifying priorities explicitly in the programming language, but the execution ordering among tasks can be specified with the parallel operator. Using the operator `||`, the developer can specify concurrent behavior. In a construct such as `[A || B || C]`, *A* has priority over *B* and *C*, and *B* has

```

await immediate START;
2
abort
    every immediate S do
        count := count + 1;
        emit TIME(count mod 6);
7    end
when STOP

sustain TIME(count);

```

LISTING 10.7 Task τ_{tsec} in Esterel.

```

await immediate START;

weak abort
    every immediate S do
5        count := count + 1;
        emit TIME(count mod 60);
    end
when STOP
    pause
10 sustain TIME(count);

```

LISTING 10.8 Task τ_{tsec} version two in Esterel.

priority over *C*. This permits specifying priorities on a local level, and this can be used to implement the desired update behavior as in the other systems. So, the required update behavior of clock's display can be implemented.

10.5.6 Giotto

Giotto [8] provides a programming abstraction for specifying temporal constraints. It introduces a new layer, which controls the timing and guarantees value/time determinism between runs as introduced in Section 10.1. This layer is implemented via a virtual machine called an embedded machine. Giotto initially aimed strictly at single-node, multimode applications with periodic tasks, whose deadline is equal to the period. Its programming abstraction allows coding applications and guaranteeing their temporal constraints without inspecting the source code. Extensions [9,10] to Giotto introduce support for distributed systems, background tasks, and division of tasks into subtasks.

10.5.6.1 The Model

Giotto realizes a time-triggered approach for computation, which releases tasks in a time-triggered periodic manner. Each task executes the following stages: (1) read inputs (e.g., sensor data), (2) call task-specific functionality, and (3) write outputs (e.g., update actuators). The time between (1) and (3) is called the logical execution time (LET) of a task. The task is released at the beginning and terminated (i.e., completed) at the end of the LET. The release and the completion event are time-triggered. Within these two events, the task-specific functionality is executed according to a scheduling scheme. The start of the LET specifies the point in time when the input values are read. The end of the LET specifies the point in time when the output values are written (even if the task has completed before that time and the output is not available until the end of the LET). A task needs to terminate before the end of its LET, so the worst-case execution time must be smaller than the LET.

Figure 10.10 shows task's life cycle in Giotto. The model includes a new state *Waiting*, which is not present in the default model (cf., Figure 10.1). The only temporal constraint that the developer can specify is the LET. The condition $t == \text{LET}$ guards the transition from the state *Running* to the state *Completed*. The state resides in the state *Waiting* until the LET has expired and it switches to the state *Completed*. Note that the task cannot reside in the state *Completed*. It is instantly released again, since in Giotto all tasks by construction have a deadline equal to their period.

10.5.6.2 The Stopwatch

Giotto uses a specification language to describe temporal constraints of tasks. In this language, the developer specifies only the timing, but not the functionality. Other programming languages, such as C, can be used to program the functionality. In the following, we specify the timing of stopwatch's tasks in Giotto's specification language.

Listing 10.9 declares all four tasks for the four columns on the stopwatch. Each task consists of a driver that is responsible for reading and writing its values, and an actuator for updating clock's display. The

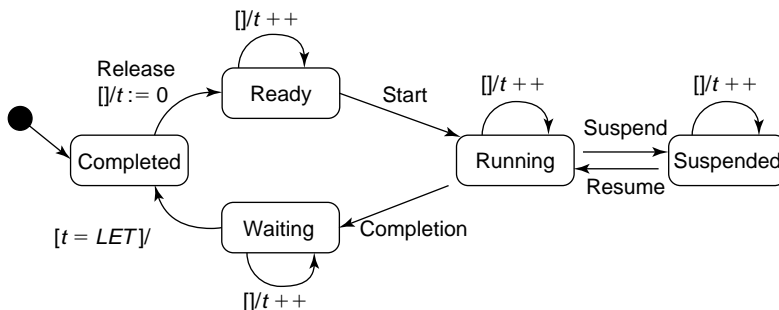


FIGURE 10.10 Task life cycle in Giotto.

```

    start Started {

mode Started() period 3600 {
    actfreq 3600 do act_sec(a_sec_driver);
5   taskfreq 3600 do comp_sec(sec_driver);

    actfreq 60 do act_tsec(a_tsec_driver);
    taskfreq 60 do comp_tsec(tsec_driver);

10   actfreq 10 do act_min(a_min_driver);
    taskfreq 10 do comp_min(min_driver);

    actfreq 1 do act_tmin(a_tmin_driver);
    taskfreq 1 do comp_tmin(tmin_driver);
15 }
}

```

LISTING 10.9 The stopwatch in Giotto.

listing also declares only one mode; the full example would need three modes: *Stopped*, *Started*, and *Reset*. Guards would then specify mode switches, so for example, the system starts in the mode *Reset* and when the user presses the start button, it switches into the mode *Started*. Listing 10.9 also excludes the declaration of drivers, tasks, and sensors.

Listing 10.9 declares the start mode to be *Started* in which the stopwatch is running. This mode *Started* has a period of 3600 s. Within this period, all tasks execute with a specific frequency. For example, the task `comp_tsec` executes with the frequency 60, so it executes 60 times within 3600 s which amounts to once every 10 s. The actuators also execute with a specific frequency when updating clock's display. For example, the actuator `act_min` has a frequency that results in actuator's execution once every minute. The functionality code for both the task and the actuator may be implemented in an arbitrary programming language.

Giotto's value update behavior differs from the other systems. The actuators are executed at a specific frequency and are independent of the task. Given a program as specified in Listing 10.9, a compiler generates a code that ensures the timely execution of these actuators. For example, at 12 o'clock, all four actuators will be executed concurrently. Consequently, the clock's display will change independently from task's runtime behavior, because it is irrelevant when, for instance, task `task_min` produces its new value as long as it produces this new value before its actuator is activated. However, Giotto lacks mechanisms for specifying priorities for actuator updates, so it is up to the code generator to put the actuator execution into a specific order. Consequently, the updates on clock's display will always be the same in all runs; however, the developer cannot specify this behavior in Giotto.

10.6 Comparison and Conclusions

Temporal control is an essential part of real-time systems. It means putting constraints on the program with respect to timing. Here we summarize the examined systems and compare them. Note that this comparison looks only at the direct support in the language and system, because virtually any temporal constraint can be programmed implicitly.

The Esterel and other synchronous languages [1] provide the most elaborate control. The reduction of computation to time instances allows the developer to specify precise temporal constraints such as:

TABLE 10.1 Comparison Sheet

Name	Granularity	Task Model	Type	Constraints	Error Handling
PEARL-90	Task	per, sus	Spec.	Abs.&rel.	No
Temp. scopes	Statement	off, per, dl ^a	Spec. ^b	Abs.&rel.	Exceptions
ARTS kernel	Task, fun. ^c	ph, off, per ^d	Spec. ^b	Rel.	No
RTSJ	Task	off, per, dl	Prgm.	Abs.&rel.	Exceptions
Esterel	Statement	Reactive	Prgm.	—	No
Giotto	Task	per	Spec.	Rel.	No

^a Also timing control for one-time execution (e.g., statement blocks) with offset.

^b Although it is specification-based, it intertwines code and timing specification.

^c ARTS provides different temporal control elements for tasks and functions.

^d Also offers deadlines for function calls.

Does the stopwatch stop at the moment you press the stop button or right after that moment has passed? The PEARL language and temporal scopes provide the most elaborate support for specifying temporal constraints for tasks and statements. PEARL concentrates on temporal constraints for task releases and when they are resumed; however, it leaves out control for deadlines and task completion. Temporal scopes provide means for controlling task releases as well; they concentrate on constraints during task's runtime such as minimum/maximum delay and maximum elapse time. Temporal scopes also provide the most fine-grained support for temporal constraints, since they permit placing constraints on individual statements. Most other systems, except ARTS, only operate at the level of tasks. ARTS permits placing constraints on tasks and individual functions. Esterel also permits coding constraints on individual statements, however, it is more an implicit control compared to temporal scopes. Also, all systems with tasks can theoretically wrap individual statements into a single task and thus also support constraints on individual statements.

For each of these systems, we presented a state machine showing the life cycle of tasks, functions, and statement blocks. Table 10.1 provides an overview of the state machine's properties of the systems and languages that we discussed in this chapter. The first column lists the name. The second column lists the code granularity, the system provides the developer to place temporal constraints. The third column lists the constraints that are supported in the system and language. The legends are as follows: *per* means supporting periodic rerelease, *dl* means supporting deadlines different from the task period, *off* means supporting task offsets from its release, *sus* means supporting suspension constraints, and *ph* means phasing.

The fourth column lists the type of support for temporal constraints the system provides. Two systems—RTSJ and Esterel—require the developer to implement the temporal constraints using the programming language. RTSJ provides an object-oriented programming interface with classes like `ReleaseParameters` and `RelativeTime`. In Esterel, all these concepts have to be implemented on the basis of *ticks*, which separates time instances from each other. The remaining systems offer specification-based support. PEARL and temporal scopes use a very readable statement to declare temporal constraints for tasks and program blocks. In the ARTS kernel, this declaration is more cryptic, because it is merely a sequence of comma-separated numbers and each position denotes a particular parameter in the temporal constraint. For example, the fourth parameter in the list specifies the task period. Giotto uses a different approach than the other systems with specification-based support. In the previously mentioned systems, the temporal constraints and the program logic are intertwined and stored together in one file. In PEARL, a clear separation can be made, but a temporal constraint can also be placed somewhere in the code. In Giotto, temporal constraints are specified independently from the program sources and are completely separated from the program logic.

The fifth column lists what kinds of temporal constraints the system supports. Only PEARL and temporal scopes support absolute temporal constraints throughout the life cycle. PEARL provides the `AT` instruction and temporal scopes provide repetitive temporal scopes that can also use absolute time.

RTSJ also supports absolute constraints, however, throughout the whole life cycle. All other systems only support relative temporal constraints, although absolute constraints can be implemented implicitly.

Finally, the last column lists system's support for error handling. Temporal scopes and RTSJ support timing error handling. Both systems use the concept of exception handling, and in case the runtime system detects a timing violation, it raises an exception and executes the programmed exception handler. ARTS also supports error handling; however, it is limited to timing violations in functions, not in tasks.

The language constructs and systems overviewed in this chapter are a representative subset of the many different systems and languages available. Most real-time kernels provide a programmable interface for temporal constraints and they use similar concepts as presented in this work. A large number of systems support the basic temporal constraints, such as period and deadline. For example, SHaRk [11], Asterix kernel [12], the HARTIX kernel [13], nano-RK [14], timed multitasking [15], and virtually all commercial real-time systems. Some of them have interesting extensions of the presented concepts. For example, the Spring kernel [16] allows specifying a function to compute the worst-case execution time depending on the current task inputs. It also introduces incremental tasks that produce an immediate result and use the remaining time to refine that answer, similar to imprecise computation [17]. In such systems, temporal constraints can be specified for both parts: the mandatory part for the immediate answer and the part that refines the result. Another system is TMO [18], an acronym for *time-triggered, message-triggered objects*, that also supports temporal constraints on tasks and functions similar to the presented systems; however, its main focus is on the object-oriented programming paradigm.

As a concluding remark, we note that as timeliness of computation is a requirement in real-time systems, temporal constraints are a fundamental concept for such systems and research on how to specify and where to place these constraints has been extensive. The investigated sample of systems and languages shows their different approaches and their different levels of support for placing temporal constraints; however, none has it all. Depending on the needs of the actual application, one or the other system may be better suited. Future research on this topic should consider the currently established work and consolidate the different approaches into one unified framework for placing temporal constraints using a specification-based approach with comprehensive error handling.

References

1. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1997.
2. G. Berry. *The Esterel v5 Language Primer*. Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, version 5.21 release 2.0 edition, April 1999.
3. S. Fischmeister and K. Winkler. Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005.
4. DIN 66253, Teil 1: Informationsverarbeitung, Programmiersprache PEARL, Basic PEARL Entwurf. Beuth Verlag Normenausschuss Informationsverarbeitung im DIN Deutsches Institut für Normung e.V, Berlin, Köln, June 1978.
5. GI-Fachgruppe Echtzeitsysteme und PEARL. *PEARL 90—Language Report*, 2.2 edition, 1998.
6. I. Lee and V. Gehlot. Language constructs for distributed real-time programming. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, 1985.
7. H. Tokuda and C. W. Mercer. Arts: A distributed real-time kernel. *SIGOPS Oper. Syst. Rev.*, 23(3): 29–53, 1989.
8. T. A. Henzinger, C. M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. In T. A. Henzinger and C. M. Kirsch, editors, *Proc. of the 1st International Workshop on Embedded Software (EMSOFT)*, number 2211 in LNCS. Springer, Berlin, October 2001.
9. G. Menkhaus, S. Fischmeister, M. Holzmann, and C. Farcas. Towards efficient use of shared communication media in the timed model. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

10. S. Fischmeister and G. Menkhaus. Task sequencing for optimizing the computation cycle in a timed computation model. In *Proc. of the 23rd International Digital Avionics Systems Conference (DASC'04)*. IEEE Press, 2004.
11. P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proc. of the 13th IEEE Euromicro Conference on Real-Time Systems (ECRTS'01)*, June 2001.
12. H. Thane, A. Pettersson, and D. Sundmark. The Asterix realtime kernel. In *Proc. of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, June 2001.
13. C. Angelov and J. Berthing. A jitter-free kernel for hard real-time systems. In *Proc. of the 1st International Conference on Embedded Software and System (ICESS'04)*, Hangzhou, China, December 2004.
14. A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An energy-aware resource-centric operating system for sensor networks. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS'05)*, December 2005.
15. J. Liu and E. Lee. Timed multitasking for real-time embedded software. *IEEE Control Syst.*, 22(6): 65–75, 2003.
16. J. A. Stankovic and K. Ramaratham. The Spring kernel: A new paradigm for hard real-time operating systems. *IEEE Software*, 8(3): 62–72, 1991.
17. S. Natarajan. *Imprecise and Approximate Computation*. Kluwer Norwell, MA, 1995.
18. K. H. Kim. Object structures for real-time systems and simulators. *IEEE Computer*, 30(8): 62–70, 1997.

11

The Evolution of Real-Time Programming

Christoph M. Kirsch
University of Salzburg

Raja Sengupta
University of California

11.1	Introduction	11-1
11.2	The Computing Abstractions of Control Engineering	11-3
11.3	Physical-Execution-Time Programming	11-6
11.4	Bounded-Execution-Time Programming	11-7
11.5	Zero-Execution-Time Programming	11-10
11.6	Logical-Execution-Time Programming	11-12
11.7	Networked Real-Time Systems	11-14
	DNRTS • NNRTS	

11.1 Introduction

Real-time programming has always been one of the most challenging programming disciplines. Real-time programming requires comprehensive command of sequential programming, concurrency, and, of course, time. Real-time programming has many application domains. In this chapter, however, we structure our discussion around digital control systems for the following two reasons. First, digital control systems represent a large portion of real-time software in a diverse set of industries. Examples include automotive power train control systems, aircraft flight control systems, electrical drive systems in paper mills, or process control systems in power plants and refineries. Many of these systems distribute their components over networks with significant communication delays. Therefore, we also discuss networked real-time programming (Section 11.7). The second reason is pedagogical. Digital control is defined by a set of abstractions that are real-time programmable and mathematically tractable in the context of the dynamics of physiochemical processes. The abstractions clearly define what application engineers (control engineers) expect of real-time programming. These abstractions therefore constitute a precise definition of the real-time programming problem (Section 11.2).

Control engineers design in difference equations and often use modeling tools such as Simulink* to simulate and validate their control models. The part of Simulink most commonly used to specify digital control, i.e., discrete time with the discrete fixed-step solver and mode:auto, is well explained by the so-called synchronous semantics [7,38]. The key abstraction, we call the *synchronous* abstraction, is that any computation and component interaction happens instantaneously in zero time or with a delay that is exactly the same at each invocation. For example, if a controller computes every 20 ms, either the controller is assumed to read its inputs, compute its control, and write the corresponding outputs instantaneously

*www.mathworks.com.

at the beginning of each period in zero time, or read its inputs instantaneously at the beginning of the period, compute during the period, and write the corresponding output exactly 20 ms (one period) later. In control parlance, these are the *causal* and *strictly causal* cases, respectively. Systems of difference equations can directly be expressed and structured in this model. The inclusion of strictly causal components (components with delays of a period or more) is typically required to break feedback loops and account for the unavoidable computation or communication delays that will be present in the real control system.

Real-time programmers design in interrupt handlers, device drivers, and schedulers, which are concepts on levels of abstraction that are obviously unrelated to difference equations. The result is a conceptual and technical disconnect between control model and real-time code that bedevils many control building projects. The control engineer specifies a Simulink design that the real-time programmer finds unimplementable. The real-time programmer then fills in gaps to produce an implementation that a control engineer finds uncontrollable. The process of interaction as the two iterate to produce a correct design is prolonged by the different backgrounds of control engineers and real-time programmers. This gap explains many of today's problems in real-time software design for digital control such as high validation and maintainance overhead as well as limited potential for reusability and scalability.

However, during the last two decades, real-time programming has come closer and closer to the computing abstractions of digital control. This convergence is visible in the form of an increasingly automated real-time programming tool chain able to generate real-time code from high-level specifications produced by control engineer for even large and complex control systems. We tell this story by tracing how real-time programming methodologies have evolved from early, so-called physical-execution-time (PET) and bounded-execution-time (BET) programming [5] to high-level programming models that incorporate abstract notions of time such as synchronous reactive or zero-execution-time (ZET) programming [13] and logical-execution-time (LET) programming [17].

PET programming had been developed to program control systems onto processor architectures with simple instructions that have constant execution times such as many microcontrollers and signal processors. The control engineer had to specify a sequence of instructions, sometimes in an assembly language. One could not declare concurrent, multicomponent designs in the dataflow. While this forced the control engineer to work in a low-level language, it resulted in temporally accurate programs and a precise accounting of input–output delay.

The emergence of operating systems and real-time scheduling led to PET programming being replaced by BET programming. BET programming aimed at handling concurrency and real time. A control engineer could express design in terms of multiple concurrently executing components, i.e., *tasks* in BET terminology. Each task could have a deadline, the period of the controller being the typical example of a deadline. This captured another essential aspect of the computing abstraction desired by the control engineer. Real-time programmers built real-time operating systems that then brought the advances of real-time scheduling theory to bear on the correct temporal execution of these concurrent programs. Each component was expected to execute in the worst case before its deadline or at the correct rate. Earliest-deadline-first (EDF) scheduling and rate-monotonic (RM) scheduling [27] emerged as the most successful scheduling disciplines in the context of the BET model. The BET model is still the most widely used real-time programming model. It is supported by a large variety of real-time operating systems and development tools.

The emergence of ZET programming has been motivated by a critical weakness of the BET model with regard to what we call *I/O compositionality*. While BET schedulability analysis techniques check that a collection of tasks with deadlines, when composed, will all execute before their respective deadlines, the same is not true of functional or temporal I/O behavior. The addition of new tasks in parallel to a set of old tasks can change the behavior of the old tasks even if the new tasks have no input–output interactions with the old tasks, and the entire task collection continues to meet all deadlines. This is because the order and times when the old tasks interact are not necessarily maintained when a system is augmented. Thus, the BET model lacks I/O compositionality.

ZET programming is based on the so-called synchronous semantics in the sense that the semantics of ZET programs are in terms of a zero computing time abstraction. Well-known examples of ZET programming languages are Lustre [14] and Esterel [3]. ZET programs are I/O-compositional in the

sense that if new tasks are added to old tasks as previously discussed, the I/O behavior of the old tasks will remain unchanged. This is guaranteed by ZET compilers, which accept concurrent, multicomponent control designs but produce a sequential program to be executed by the operating system. Thus, ZET compilers take no advantage of the scheduling facilities of real-time operating systems. Thus, while ZET programming brought I/O compositionality to programming real-time control, it lost the BET connection with real-time scheduling.

The most recent of the real-time computing abstractions is the LET model. The notion of LET programming was introduced with Giotto [17]. A LET program is assumed to take a given, strictly positive amount of logical time called the LET of the program, which is measured from the instant when input is read to the instant when output is written. The time the program actually computes between reading input and writing output may be less than the LET. The LET compilation process reproduces the I/O compositionality of the ZET model in its first stage of compilation. This produces code composed of multiple tasks with their I/O times specified to ensure a concurrent execution that is restricted only to the extent necessary to be consistent with the I/O dependencies between the tasks [16]. This multitask code with its I/O specifications can be handled by a real-time operating system thereby leveraging the strengths of real-time scheduling during execution.

Both the ZET and the LET models close the semantical gap between digital control design and implementation, and have been shown to integrate well with simulation environments such as Simulink [7,21]. The LET model is closer in the sense that it tries to handle input and output exactly at times modeled in the control system, e.g., at the beginning and the end of the controller period. This corresponds exactly to the assumptions made by digital control as discussed in Section 11.2. ZET programs may produce output before the end of the period, but in the case of strictly causal designs, they can conform to the digital control abstraction by buffering the output until the end of the period. The other difference between the two is the use of real-time scheduling and operating systems [21]. Only the buffered, i.e., LET portions of ZET programs, have recently been shown to utilize real-time scheduling services such as EDF [37]. Recent research discussed in Section 11.7 shows the ZET and LET models also extend to some control systems distributed over networks.

In our discussion, we show how the evolution of real-time programming enables modern control software engineering. In the following section, we begin by describing the computing abstractions of control engineering. Then, we continue by covering the previously mentioned four real-time programming models in detail. The final section provides an overview of the most recent work on control software engineering for distributed architectures using deterministic and nondeterministic communication channels.

11.2 The Computing Abstractions of Control Engineering

In this section, we explain the dominant abstraction in control engineering for the specification of digital control. Figure 11.1 shows the basic kind of component in a control system. Control engineers design digital

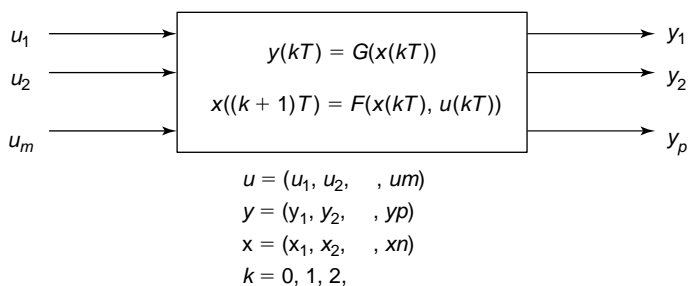


FIGURE 11.1 Basic type of component in a control system.

control in difference equations. The component in the figure has two equations, the first being called the *output* equation and the second the *state* equation. The input signals u are typically measurements from sensors, commands from operators, or commands from other controllers. The output signals y are typically commands to actuators, commands to other controllers, or information displayed to operators. x , called the state in control parlance, is the memory of the component. T is the period of the component. At each expiration of the period T , the component

1. Reads its inputs u
2. Computes its outputs y , and
3. Updates its state x

Thus the control engineer's assumption about real-time behavior is as illustrated in Figure 11.2. On expiration of each period T , a read-and-write operation with the environment is expected to occur. It is not expected to occur earlier or later. Thus, it is expected that the embedded computing environment will produce an execution that conforms to the idealization as closely as possible. The dotted arrows indicate that the output values expected at the end of the period depend on the input values read at its beginning. The component has a so-called one-step delay. The intervening time T makes it possible to implement embedded computing processes that conform to the assumptions of the control engineer. Often, the different input and output signals constituting u and y will have different periods or frequencies. If so, the period or frequency of the component will be the highest common factor or the lowest common multiple of the period or frequency, respectively. For example, if the component in Figure 11.1 were to have 2 and 3 Hz input and output signals, the component would compute at 6 Hz, i.e., T would be $1/6$ s.

Control engineers design large control systems compositionally. The rules of composition are also reasonably well established. Figure 11.3 illustrates a two-component control system. The real-time behavior of each component will be assumed to be similar to Figure 11.2. Note that the first component has no one-step delay, i.e., the output at time T depends on the input at time T . However, this is not a problem if the signal z is internal to the program and only signals u and y interact with the environment. One can see by working through the equations that the dependence between u and y is still delayed by one period. Thus, if the embedded computing environment is able to compute the equations in both components within period T , it will be able to conform to the control engineers timing assumptions at the interface with the environment, which is all that matters for the correct control of the environment.

Figure 11.4 illustrates a three-component control system to clarify another aspect of the rules of composition. To see this in a simple fashion, we have dropped the state equations and assumed the periods of the three components in Figure 11.4 are the same.

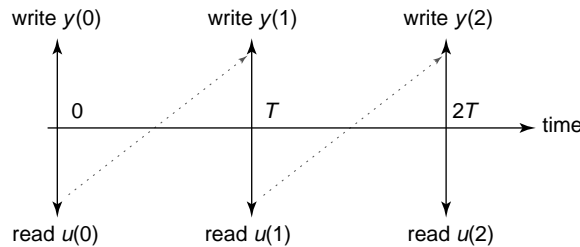


FIGURE 11.2 Basic timing behavior of a controller.

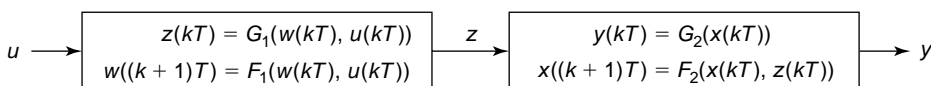


FIGURE 11.3 A two-component control system.

Figure 11.5 illustrates a behavior of the three-component control system. At each expiration of the period, the system would be expected to execute the following actions in the following order:

1. Read u
2. Execute block A to output y and z using the new value of u
3. Execute block B to output w using the new value of z , and
4. Execute block C to output v based on the new values of w and y

Thus the data dependencies between components as indicated by the arrows between components constrain the order in which the components are to be executed. In general, the components in the system have to be linearized in an order consistent with the arrows interconnecting the components [38]. Figure 11.5 shows the values of the variables that should be obtained at four consecutive sample times if u were to be as shown in the figure and execution were to conform to the above semantics.

Finally, control engineers connect components in feedback as illustrated by Figure 11.6. In such a case, the signal y is expected to solve the equation

$$y(kT) = H(x(kT), u(kT), E(w(kT), y(kT)))$$

Without restrictions on H or E , it is difficult to provide real-time guarantees on the time required to find a solution. The restriction most commonly assumed is to require at least one of the components in the

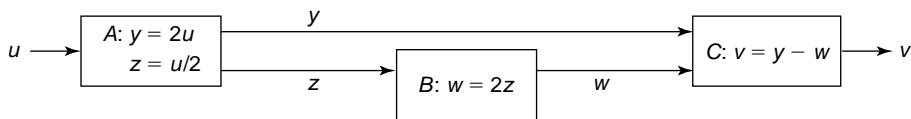


FIGURE 11.4 A three-component control system.

Clock	0	1	2	3	4
u	2	4	2	4	2
$y (=2u)$	4	8	4	8	4
$z (=u/2)$	1	2	1	2	1
$w (=2z)$	2	4	2	4	2
$v (=y - w)$	2	4	2	4	2

FIGURE 11.5 Behavior of the three-component control system.

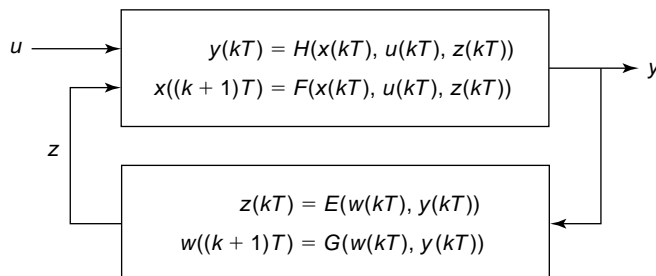


FIGURE 11.6 A feedback control system.

feedback loop to have a one-step delay. For example, if we assume $z(kT) = E(w(kT))$, the equation above becomes

$$y(kT) = H(x(kT), u(kT), E(w(kT)))$$

The problem now becomes similar to that of computing any other block. In general, a multicomponent system with feedback loops having a one-step delay can be handled as before by linearizing the components in any order consistent with the input–output data dependencies.

11.3 Physical-Execution-Time Programming

Real-time systems interact with the physical world. Despite functional correctness, temporal accuracy and precision of I/O are therefore particularly relevant properties. Early real-time programmers have addressed the challenge of programming such systems by taking the execution times of machine code into account. As an example, consider Program 11.1, which shows an implementation of a task t , and Figure 11.7, which depicts the execution of t . Throughout this chapter, we assume that a *task* is a code that reads some input u , then computes from the input (and possibly some private state x) some output y (and new state), and finally writes the output. A task does not communicate during computation except through its input and output. In Figure 11.7, task t is reading input for one time unit from time instant 0 to 1, and then starts computing for eight time units. At time instant 9, t completes computing and is writing its output for one time unit from time instant 9 to 10. Thus the PET of task t , i.e., the amount of real time from input to output, which is the only physically relevant time, is 10 time units. Here, the intention of the programmer is to have t write its output exactly after eight time units have elapsed since t read its input to achieve a physical execution time of 10 time units. In other words, the implementation of t is correct only if t computes its output in exactly eight time units. We call this method *PET Programming*.

PET programming only works on processor architectures where execution times of instructions are constant and programs can have exclusive access to the processor. Widely used architectures enjoy these properties, e.g., many microcontrollers and signal processors. PET programming results in cycle-accurate real-time behavior and enables high I/O throughput. However, a critical drawback of the PET model is the lack of compositionality. Integrating multiple PET programs on the same processor or distributing PET programs on multiple processors with nontrivial communication latencies is difficult. In the evolution of real-time programming, the following BET model (or scheduled model [24]) can be seen as an attempt to address the lack of compositionality and support for other, less predictable processor and system architectures.

```

read  $u$  ;
compute  $x := F(x, u)$  ;
compute  $y := G(x)$  ;
write  $y$  ;

```

PROGRAM 11.1 A task t .

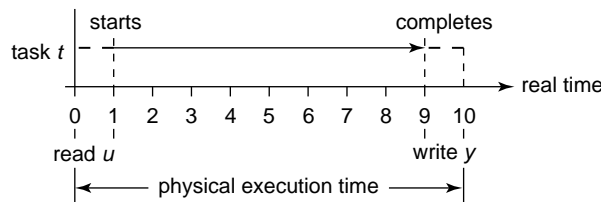


FIGURE 11.7 The execution of Program 11.1.

11.4 Bounded-Execution-Time Programming

Real-time systems interact with the physical world. Handling multiple concurrent tasks in real time is therefore an important requirement on such systems. This challenge has traditionally been addressed by imposing real-time bounds on the execution times of tasks. As an example, consider Program 11.2, which shows an implementation of a periodic task t , and Figure 11.8, which depicts the execution of t in the presence of other tasks. At time instant 0, t is released but does not read its input before time instant 2 because other tasks are executing. Task t is reading input for one time unit and then starts computing until time instant 4 at which t is preempted by some other task. Only two time units later, the execution of t is resumed. At time instant 7, t completes computing and immediately starts writing its output for one time unit until time instant 8. At this point, t loops around in its while loop and waits for time instant 10. In the example, the period of t is 10 time units. At time instant 10, t is released again but executes differently now because of different interference from other tasks. This time t starts reading input already after one time unit has elapsed since t was released. Moreover, t gets preempted three times instead of just one time, and t needs one time unit more to compute. As a result, compared to t 's previous invocation, t starts writing output one time unit later with respect to the time t was released. Nevertheless, as long as t completes reading, computing, and writing before it is released again, i.e., within its BET, the while loop of t will execute correctly. In the example, the BET of t is equal to t 's period, i.e., 10 time units. In other words, the implementation of t is correct only if t reads input, computes state and output, and writes output in less than its BET. We call this method *BET Programming*.

BET programming works on processor architectures for which upper bounds on execution times of individual tasks can be determined, and runtime support is available that handles concurrent tasks according to scheduling schemes for which upper bounds on execution times of concurrent tasks can be guaranteed. An upper bound on the execution time of an individual task is commonly referred to as the *worst-case execution time* (WCET) of the task. WCET analysis is a difficult problem, depending on the processor

```

initialize  $x$  ;
int  $n := 0$  ;
while (true) {
    wait for  $n$ -th clock tick ;
    read  $u$  ;
    compute  $x := F(x, u)$  ;
    compute  $y := G(x)$  ;
    write  $y$  ;
     $n := n + \text{period}$  ;
}

```

PROGRAM 11.2 A periodic task t .

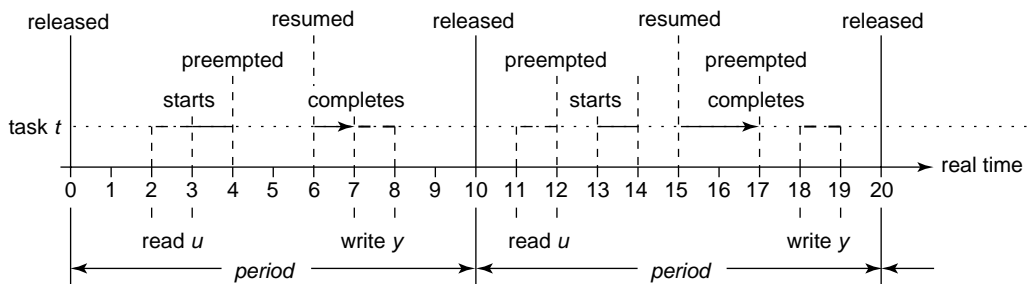


FIGURE 11.8 The execution of Program 11.2 in the presence of other tasks.

architecture and the task implementation [15]. For example, memory caches and processor pipelines often improve average performance significantly but are inherently context-sensitive concepts and therefore complicate WCET analyses, in particular, if tight WCET bounds are needed. If loops are present in the task implementation, WCET analysis usually requires programmers to provide upper bounds on the number of loop iterations because of the undecidability of the halting problem.

Many real-time operating systems today provide the necessary runtime support for BET programming. The problem of scheduling concurrent real-time tasks has been studied extensively [6]. There is a large variety of real-time scheduling algorithms and schedulability tests, many of which have been implemented in state-of-the-art real-time operating systems and tools. The most widely used scheduling algorithms in BET systems are probably RM and EDF scheduling [27]. The main advantage of BET programming is compositionality with respect to BETs, a property that effectively rests on real-time scheduling theory. A system of BET tasks can be extended by a new task provided the extended system is still schedulable. In this case, all tasks in the extended system will still execute within their BETs. While BET programming is compositional in the sense that the bounds on execution times of individual tasks do not change, testing schedulability usually is not, i.e., testing the extended system may involve reconsidering the entire original system. The so-called *compositional scheduling* aims at reducing the need to reconsider already scheduled tasks. Compositional scheduling techniques have only recently received more attention [11,20,32]. The ZET model also benefits from the results in scheduling hybrid sets of real-time and non-real-time tasks [6], which have also made it into real-time operating systems.

The BET model is probably the most widely supported real-time programming model. Many real-time programmers are familiar with this model. Since enabling BET programming is essentially a matter of adequate operating system support, existing programming languages can easily be utilized in developing BET programs. Moreover, there are many real-time programming languages [5] that are based on the BET model such as Ada and PEARL. Recently, a real-time Java specification (RTSJ), which is also based on the BET model, has been proposed and implemented. Real-time scheduling theory even provides results on how to schedule processes and threads in real time that share data through potentially blocking mechanisms such as semaphores and mutexes. In other words, the BET task model may even be and has been extended to, e.g., standard process and thread models. As an example, consider Program 11.3, which shows a periodic task h that shares its state with other, possibly periodic tasks using a mutex s , and Figure 11.9, which depicts an execution of h sharing its state with a task l . At time instant 0, tasks h and l are released where h has a period of 10 time units and l has a period of, say, 30 time units. The example shows an EDF schedule of both tasks, i.e., h is scheduled to execute first. We assume in this example that deadlines are equal to periods. The first attempt of h to lock s at time instant 2 succeeds immediately because no other task holds the lock on s . At time instant 6, h has completed writing its output and l is scheduled to execute next. At time instant 9, l 's attempt to lock s succeeds, again immediately because no

```

initialize  $x$  ;
int  $n := 0$  ;
while ( true ) {
    wait for  $n$ -th clock tick ;
    read  $u$  ;
    compute  $x := F(x, u)$  ;
    lock  $s$  ;
    share  $x$  ;
    unlock  $s$  ;
    compute  $y := G(x)$  ;
    write  $y$  ;
     $n := n + \text{period}$  ;
}

```

PROGRAM 11.3 A periodic task h sharing its state using a mutex s .

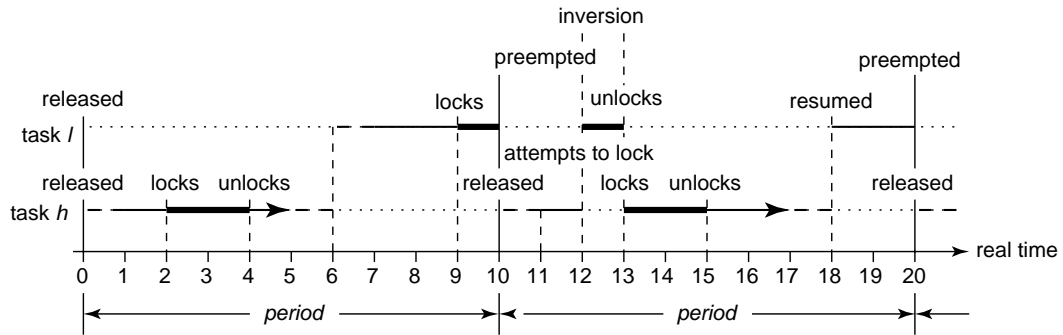


FIGURE 11.9 The execution of Program 11.3 sharing its state with a task *l*.

other task holds the lock on *s*. At time instant 10, however, *h* is released again and scheduled to execute next because *h* has an earlier deadline than *l*. As a consequence, *l* is preempted while holding the lock on *s*. At time instant 12, *h* attempts to lock *s* but is blocked because *l* still holds the lock on *s*. Since *l* is the only unblocked task, *l* is scheduled to execute next. One time unit later, *l* unlocks *s*. As a result, *h* is unblocked and immediately receives the lock on *s*. Then, *h* is scheduled to execute next because *h* has an earlier deadline than *l*. Finally, *h* unlocks *s* at time instant 15 and completes writing its output at time instant 18. Here, the interesting system anomaly is from time instant 12 to 13, which is traditionally called *priority inversion* because a task with lower priority (task *l*, which has a later deadline) is scheduled to execute in the presence of a task with higher priority (task *h*, which has an earlier deadline). Priority inversion is problematic because it gives tasks, which may even never attempt to lock *s*, the chance to prevent both task *h* and *l* from executing. For example, if a task *m* is released during priority inversion and *m* has a priority that is less than *h*'s priority but greater than *l*'s priority, then *m* is scheduled to execute instead of *l*. Therefore, *m* may not only prevent *l* from ever unlocking *s* again but also, as a result, prevent *h* from ever executing again although *m* has a lower priority than *h*. Priority inversion is often explained using all three tasks *l*, *m*, and *h* at once. However, priority inversion itself merely requires *h* and *l*, whereas *m* is only necessary to explain the danger of priority inversion. In other words, only in the presence of tasks such as *m*, priority inversion may be harmful and needs to be avoided using techniques such as *priority inheritance* or *priority ceiling* [31]. For example, with priority inheritance, *l* would inherit the priority of *h* from time instant 12 to 13 and thus prevent *m* from being scheduled to execute during that time.

Instead of blocking mechanisms such as semaphores and mutexes, BET tasks may also utilize nonblocking communication techniques such as the *nonblocking write (NBW) protocol* [26] or extensions thereof [23] to avoid priority inversion. For example, with NBW techniques, write attempts always succeed while read attempts may have to be repeated because write attempts were in progress. In any case, there remains the burden of showing that all task executions stay within their bound execution times. Blocking techniques require that the time to communicate is bounded using scheduler extensions such as, e.g., priority inheritance. Nonblocking techniques require that the number of retries to communicate is bounded, which is possible if the maximum amount of time needed to communicate is considerably shorter than the minimum amount of time between any two communication attempts.

The main drawback of the BET model is the lack of compositionality with respect to I/O behavior, which is a semantically stronger concept than compositionality with respect to BETs. A system of BET tasks may change its I/O behavior, i.e., output values and times, as soon as tasks are added or removed, the scheduling scheme changes, or the processor speed or utilization are modified, even though the modified system might still be schedulable. Real-time programmers therefore frequently use the infamous expression “priority-tweaking” to refer to the BET programming style because it is often necessary to modify scheduling decisions manually to obtain the required I/O behavior. Once a BET system has been released, it is hard to change it again because of the system-wide scheduling effects. As a consequence, BET

systems are often expensive to build, maintain, and reuse, in particular, on a large scale. The following ZET model (or synchronous model [24]) can be seen as an evolutionary attempt to address the lack of semantically stronger notions of compositionality in the BET model.

11.5 Zero-Execution-Time Programming

Real-time systems interact with the physical world. Designing and implementing such systems in a way that is compositional with respect to I/O behavior is therefore an important challenge. Compositionality in this sense requires that a real-time program always produces the same sequence of output values and times for the same sequence of input values and times, even when running in the possibly changing presence of other programs, or on different processors or operating systems. We say that a program, i.e., its I/O behavior, is *input-determined* if, for all sequences I of input values and times, the program produces, in all runs, unique sequences $f(I)$ of output values and times. The so-called *synchronous reactive programming* [13] aims at designing and implementing input-determined real-time programs. Note that we deliberately avoid the term “deterministic” here because it has been used with very different meanings in different communities. BET communities such as the real-time scheduling and real-time operating systems communities call a real-time program deterministic if the program always runs with BETs. Language-oriented communities such as the synchronous reactive programming community call a real-time program deterministic if the I/O behavior of the program is fully specified by the program. In other words, the term “input-determined” is related to the term “deterministic” in the sense of the language-oriented communities.

The key abstraction of the synchronous reactive programming model is to assume that tasks have logically ZET. As an example, consider Program 11.4, which shows an implementation of a synchronous reactive task t , and Figure 11.10, which depicts the semantics of t based on some logical clock. At the occurrence of the first event at time instant 0, t reads its input, computes state and output, and writes its output logically in zero time. The same happens at the occurrence of the next event that triggers t at time instant 10, and so on. In other words, each iteration of t ’s while loop is completed instantaneously before any other event can occur. In terms of the logical clock, task t is therefore an input-determined program. In terms of real time, Figure 11.11 depicts an execution of t that approximates the logical semantics in the sense that t ’s output values are still input determined, while the time instants at which t writes its output are only bounded by the duration between the current and next event, i.e., by 10 time units in the example. Therefore, we also say that the synchronous reactive programming model is compositional with respect to bounded I/O behavior. Note that for t ’s output values to be input determined, it is necessary that t reads its input as soon as the event occurs that triggered t while the actual computation of t may be preempted at any time. For example, t is preempted at time instant 1 right after t read its input, or at time instant 16 right after t completed computing before writing its output. As a consequence, output may be written with latency and jitter. In the example, the output jitter is one time unit. In other words, the implementation of t is correct if t reads input at the occurrences of the triggering events but then computes state and output, and writes output only some time before the next event occurs. Despite synchronous reactive programming, we also call this method figuratively *ZET Programming*.

```

initialize  $x$  ;
while (true) {
    read  $u$  at next occurrence of event ;
    compute  $x := F(x, u)$  ;
    compute  $y := G(x)$  ;
    write  $y$  ;
}

```

PROGRAM 11.4 A synchronous reactive task t .

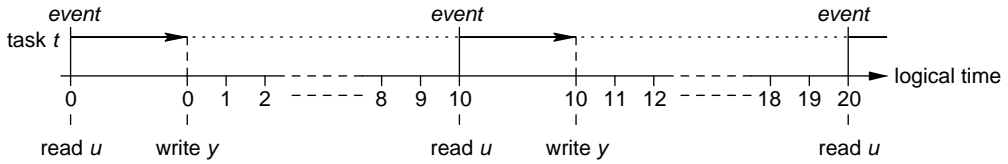


FIGURE 11.10 The logical semantics of Program 11.4.

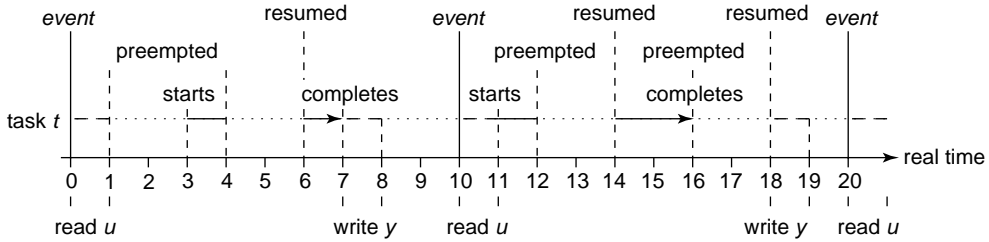


FIGURE 11.11 An execution of Program 11.4 in real time.

Similar to BET programming, ZET programming works on processor architectures for which WCETs of individual tasks can be determined. The runtime support required for ZET programming is often simpler than for BET programming because ZET tasks are usually compiled into a single sequential program that implements some form of finite state machine. As a consequence, a ZET runtime system may only provide mechanisms for event handling and buffering of input values. Shifting complexity from ZET compilers to runtime systems by utilizing dynamic scheduling schemes such as EDF at runtime has only recently received more attention. The challenge is to minimize the amount of memory (buffers) required to preserve the logical semantics of ZET tasks when using dynamic schedulers. Similar to BET programming, the advantage is potentially more effective processor utilization but possibly at the expense of temporal accuracy and precision of I/O. There is a large variety of ZET programming languages. Esterel [4] and Lustre [14] are probably the most widely used languages. Programming in Esterel is similar to imperative programming, while programming in Lustre is more declarative and oriented toward data rather than control flow.

ZET programming has several advantages. The ZET semantics is close to the semantics of modeling tools such as Simulink, which are used to develop and simulate controller designs. ZET programming therefore enables model-based design of embedded control systems in the sense that control models may directly be translated into executable ZET programs [29]. Moreover, ZET programs may be verified for functional correctness because most ZET programming languages have formal semantics. ZET-based implementations of embedded control systems are thus suitable for mission- and safety-critical applications and have already been deployed successfully by railway and avionics companies.

The ZET model also has several disadvantages. First, ZET programming distributed systems with non-negligible communication latencies are complex but have been shown to be possible on platforms with [7] and without built-in clock synchronization [2]. In the presence of unreliable communication channels, the problem is even more difficult, see Section 11.7 for more details. Second, in contrast to the BET model, most ZET programming languages are self-contained, i.e., offer combined timing and functional primitives, and thus have often not been integrated with standard programming languages. This may not be an inherent problem of the ZET model but still an economical obstacle. Third, ZET programs are typically compiled into static executables with no support for dynamic changes at runtime. In other words, while the ZET model is compositional with respect to bounded I/O behavior, the compositionality may only be utilized at design time. The following LET model (or timed model [24]) is a recent attempt to address these issues.

```

initialize  $x$  ;
int  $n := 0$  ;
  while ( true ) {
    int  $k := n + \text{offset}$  ;
    read  $u$  at  $k$ -th clock tick ;
    compute  $x := F(x, u)$  ;
    compute  $y := G(x)$  ;
     $k := k + \text{let}$  ;
    write  $y$  at  $k$ -th clock tick ;
     $n := n + \text{period}$  ;
  }

```

PROGRAM 11.5 A LET task t .

11.6 Logical-Execution-Time Programming

Real-time systems interact with the physical world. The I/O behavior of such systems and, in particular, the timing of I/O is therefore an important aspect, which has recently received more attention in the latest evolutionary attempt at defining real-time programming models. We call this method *LET programming*. The key abstraction of the LET model can be phrased in a computation-oriented and an I/O-oriented way. In the LET model, a task computes logically, i.e., from reading input to writing output, for some given, positive amount of time called its *logical execution time*. Equivalently, the input and output of a task in the LET model is read and written, respectively, at some given instants of time, independent of when and for how long the task actually computes. As an example, consider Program 11.5, which shows an implementation of a LET task t , and Figure 11.12, which depicts the semantics of t based on some logical clock. At time instant 0, task t is released but only reads its input with an offset of one time unit at time instant 1. Then, t is computing for eight time units. At time instant 9, t writes its output. Thus, the logical execution time of t is eight time units. At time instant 10, t is released again and repeats the same behavior. Similar to the ZET model, a LET task such as t is therefore an input-determined program in terms of the logical clock. In terms of real time, Figure 11.13 depicts an execution of t that approximates the logical semantics in the sense that t 's output values are still input determined, while the time instants at which t writes its output are only bounded by the amount of time needed to perform I/O of other tasks at the same time instants. For example, if there is another task u that needs to write its output at time instant 9, then either u 's or t 's output gets delayed by the amount of time it takes to write the output of the other task. Therefore, similar to the ZET model, we say that the LET model is compositional with respect to bounded I/O behavior but, unlike the ZET model, the output times are bounded by the overall I/O load rather than the duration between events. In the LET model, if I/O load is low, which is true for sensor and actuator I/O in many control applications, then temporal accuracy and precision of I/O may be close to hardware performance. Note that the input times can in fact not be arbitrarily accurate and precise as well but are also just bounded by the overall I/O load. The ZET model is affected by the same phenomenon. We omitted this observation in the section on the ZET model to simplify the discussion. Figure 11.13 shows that the actual times when task t computes do not affect the times when t reads input and writes output. During t 's first invocation, t starts computing with a delay of one time unit at time instant 3, is preempted once at time instant 4, and requires two time units to compute. During the second invocation, t starts computing right after reading its input, is preempted once at time instant 13, and requires three time units to compute. Note that even though t completes computing early, t waits until time instants 8 and 18 to start writing output to approximate its logical execution time as close as possible. In other words, the implementation of t is correct if t starts computing some time after t completes reading its input, and t completes computing some time before t starts writing its output. In this case, we say that t 's implementation is *time-safe* [18].

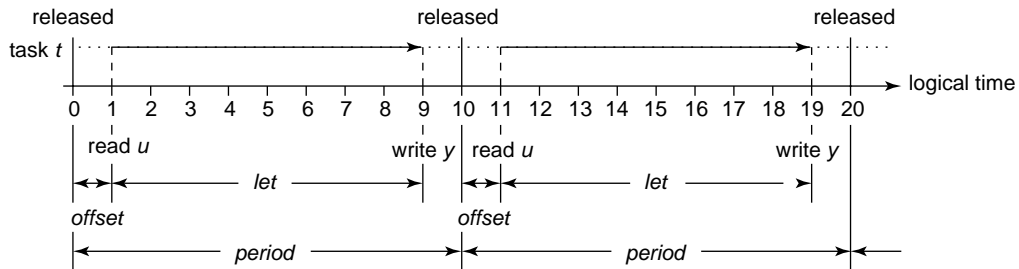


FIGURE 11.12 The logical semantics of Program 11.5.

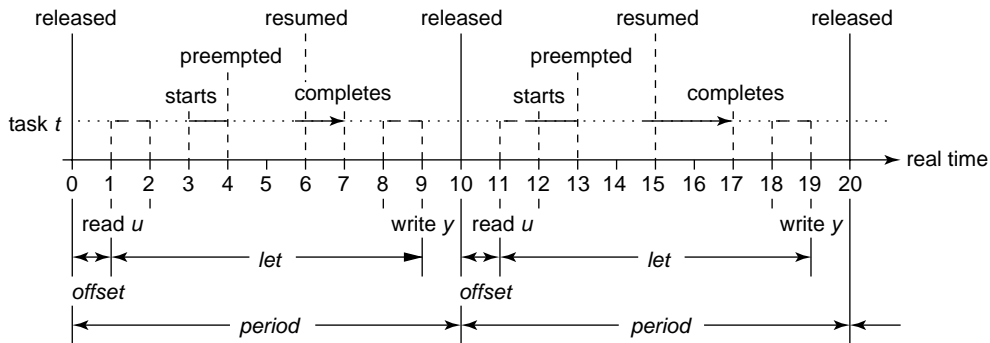


FIGURE 11.13 An execution of Program 11.5 in real time.

LET programming works on processor architectures for which WCETs of individual tasks can be determined. The runtime support required for LET programming is a combination of the runtime support necessary for BET and ZET programming. LET tasks need to be scheduled to execute, like BET tasks, according to some real-time scheduling strategy such as, e.g., RM or EDF scheduling. Accurate and precise timing of task I/O requires mechanisms for event handling similar to the mechanisms needed for ZET programming. There are four LET programming languages, called *Giotto* [17], *xGiotto* [12], *HTL* [11], and *TDL* [36]. The notion of LET programming was introduced with *Giotto*, which supports the possibly distributed implementation of periodic tasks whose logical execution times must be equal to their periods. *Giotto* also supports switching modes, i.e., switching from one set of tasks to another. *xGiotto* extends *Giotto* with support for event-triggered tasks. *TDL* extends a subclass of *Giotto* with a module concept. *HTL* is the most recent language, which supports LET programming of tasks with logical execution times less than their periods. All four languages are timing languages, which still require another programming language for the implementation of the tasks. For example, *Giotto* programs only determine when tasks are released, read input, and write output but not what and how the tasks compute. The existing LET languages are all compiled into so-called *E code*, which is executed by a virtual machine called the *Embedded Machine* [16]. *E code* is low-level timing code that determines when tasks are released, read input, and write output. The *Embedded Machine* still requires, e.g., an EDF scheduler to execute the tasks. Note that the execution of LET tasks may also be scheduled at compile time using so-called *schedule-carrying code* (SCC) [19], which is *E code* extended by instructions that explicitly dispatch tasks to execute. With SCC, no runtime scheduler is needed to execute LET programs.

LET programming has several advantages that are rooted in the compositionality of the LET model. Real-time programs written in the LET model are predictable, portable, and composable in terms of their I/O behavior. For example, Program 11.5 will show the same I/O behavior modulo I/O load on any platforms and in any context as long as the program executes in a time-safe fashion. Thus, LET programming can be seen as a generalization of the early PET programming style to handle concurrent

tasks while maintaining temporal accuracy and precision of the individual tasks' I/O. Note that LET programs may also implement conditional timing behavior while still being input determined such as, e.g., changing the frequencies of tasks or even replacing tasks by others at runtime [11]. An important advantage of Giotto is that time safety of Giotto programs can be checked in polynomial time with respect to the size of the programs although there might be exponentially many conditional timing behaviors [18]. Checking time safety of programs written in more general LET languages such as HTL may only be approximated with less than an exponential effort [11]. The LET semantics is abstract in the sense that platform details such as actual execution times and scheduling strategies are hidden from the programmer. Similar to ZET programs, modeling tools such as Simulink can be used to develop and simulate LET programs. LET programming therefore also enables model-based design in the sense that models may directly be translated into semantically equivalent executable programs [21]. Finally, LET programs written in Giotto [20], HTL [11], and TDL [9] have been shown to run on distributed systems with support for clock synchronization.

LET programming also has several disadvantages. A LET task may not take advantage of available processor time to compute and write output faster than its logical execution time. Unused processor time may only be used by other tasks. With the LET philosophy, getting a faster processor or better scheduler, or optimizing the implementations of tasks does not result in faster I/O behavior but in more processor availability to do other work. While this can be seen as a strength for certain applications including control, it can also be seen as a weakness for other applications. Nevertheless, the essence of LET programming is to program explicitly how long computation takes or, equivalently, when I/O is done. So far, with the exception of xGiotto, only real-time clock ticks have been used to determine the LET of tasks although other types of events may be used. LET programming is the most recent style of real-time programming and still immature compared to the more established concepts. LET programming languages and development tools are mostly prototypical, and have not been used in industrial applications.

11.7 Networked Real-Time Systems

This section provides an overview of the most recent work on control software engineering for distributed architectures using deterministic and nondeterministic communication channels. Recall the two- and three-block control systems illustrated in Figures 11.3 and 11.4, respectively. It is usually assumed the intercomponent dataflows represented by the arrows between the blocks are instantaneous. Here, we discuss the real-time programming of systems in which these dataflows take time. Within the control community, this class of systems has come to be called *networked control systems* (NCS).

Well-established NCS examples include systems of many controllers connected over a field bus in a process plant, or controllers communicating over a CAN* bus in a car. The networking revolution is also driving an efflorescence of new control systems for new domains like tele-surgery, smart cars, unmanned air vehicles, or autonomous underwater vehicles, distributed over the Internet or wireless LANs. This surge in NCS has led to new techniques for networked real-time programming.

We subdivide networked real-time systems into two categories distinguished by the determinism of the network interconnecting components. We call the categories deterministically networked real-time systems (DNRTS) and nondeterministically networked real-time systems (NNRTS). The DNRTS category abstracts systems in which intercomponent communications over the network are highly reliable. The loss of messages is rare enough and jitter in the intermessage time interval is small enough to be treated as exceptions or faults that are specially handled. Control systems interconnected over time-triggered architecture (TTA) [25], or lightly-loaded CAN* or Ethernets are usually in this category. The NNRTS category abstracts systems where loss or jitter in intercontroller communications is so frequent that it must be viewed as part of normal operation. In this category are systems in which intercontroller

*CAN standard ISO 11898-1.

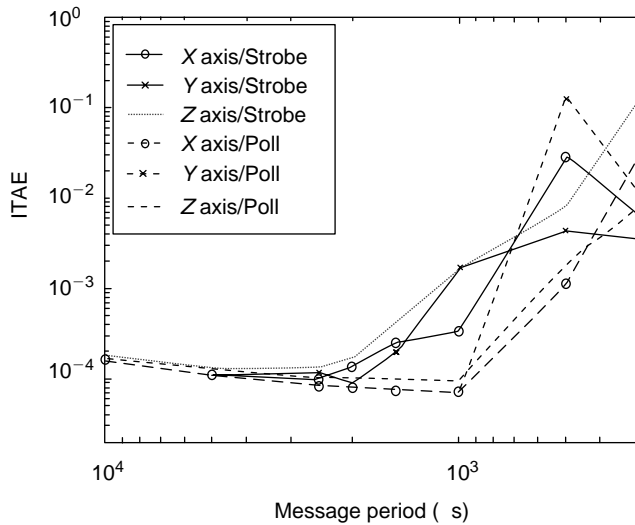


FIGURE 11.14 Control performance over Ethernet. (From L. Feng-Li, J. Moyne, and D. Tilburg. *IEEE Transactions on Control Systems Technology*, 10(2): 297–307, 2002.)

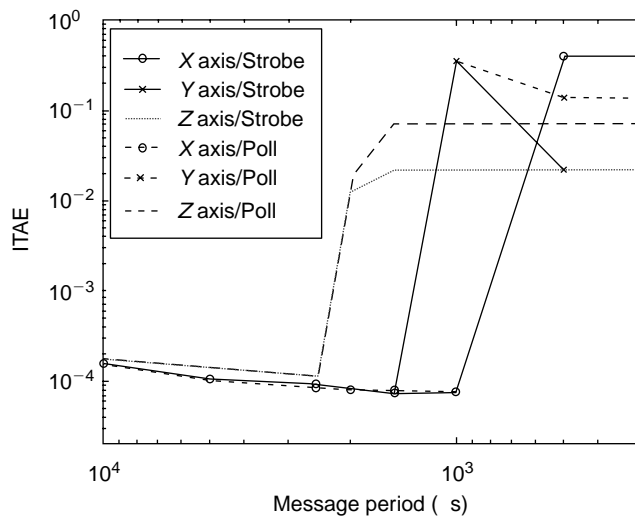


FIGURE 11.15 Control performance over CAN. (From L. Feng-Li, J. Moyne, and D. Tilburg. *IEEE Transactions on Control Systems Technology*, 10(2): 297–307, 2002.)

communications flow over TCP connections on the Internet, as UDP messages over wireless LANs like Wi-Fi* or Bluetooth†, and well-utilized Ethernet or CAN.

Figures 11.14, 11.15, and 11.16 illustrate the impact of network losses on real-time control over Ethernet, CAN, and a token passing network, respectively. The figures are reproduced from [10]. The vertical axis of the figures shows the performance of a three-axis machine tool under real-time control. Performance gets worse up the axis. There are three controllers that aim to position the machine tool as desired on the

*<http://standards.ieee.org/getieee802/802.11.html>.

†<http://www.bluetooth.com>.

X, Y, and Z axes. The controllers send their commands to and receive their measurements from the machine tool over a shared network. In all other respects, they are decoupled from each other. The horizontal axis shows the sampling rate. The sampling rate increases to the right. Figure 11.14 shows performance when the machine is controlled over an Ethernet. Initially, the Ethernet is underutilized. As the sampling rate is increased, performance improves. When there is too much traffic on the Ethernet, messages are lost, and performance deteriorates. Figure 11.15 shows performance when the controllers communicate over a CAN network. Like Ethernet, CAN is a contention access network. Any node with a message to send may contend for the bus if it detects the bus is idle. Therefore, one observes a performance pattern similar to Figure 11.14. However, unlike Ethernet, each CAN message can have a priority level. When multiple messages contend for the bus at the same time, the higher priority wins. Here, the Z-axis controller has the highest priority and the X-axis controller has the lowest priority. The figure shows the performance of the low-priority control system deteriorates first, followed by the medium- and high-priority control systems. Thus, Ethernet and CAN at low utilization can be viewed as DNRTS. At high utilization we view them as NNRTS.

Figure 11.16 shows performance when the controllers share the channel using a token-passing protocol. The network has a token that passes in a fixed order through the controllers with each vehicle receiving the token at fixed periods. On receiving the token it sends its data while the others are silent. This ensures that each controller sends and receives all its data as long as the total channel data rate is not exceeded. The performance will remain good until the channel data rate is exceeded at which point it will cease to function. Token-passing protocols represent a way to realize DNRTS for real-time control over wired networks. They are an alternative to the time-division multiple-access (TDMA) approach followed by TTA. They also improve the determinism of delay over wireless networks though not as greatly.

An example of jitter in channel access delays associated with a wireless token passing protocol is presented in [8]. The protocol is configured to give controllers access at 50 Hz, i.e., every 20 ms. The control system is activated three times as represented by the three bursts near 20 ms. Most access delays are clustered around 20 ms. However, as one can see, there is considerable jitter. The data is in UDP broadcast packets. The broadcasts are either received almost immediately or not at all as illustrated by the Erasure/Loss line in Figure 11.19.

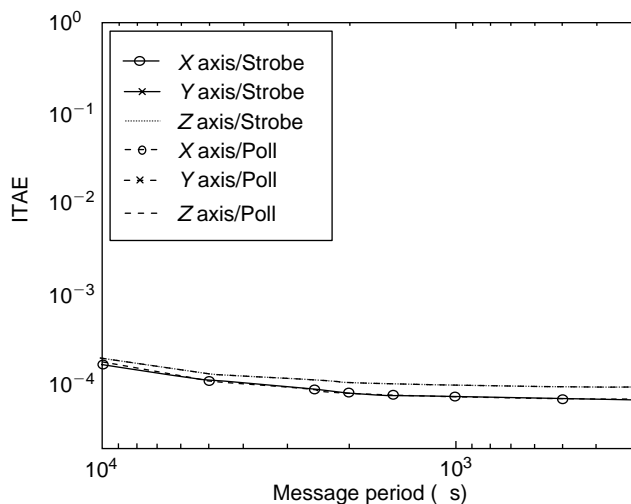


FIGURE 11.16 Control performance over Wired Token Network. (From L. Feng-Li, J. Moyne, and D. Tilburg. *IEEE Transactions on Control Systems Technology*, 10(2): 297–307, 2002.)

More recently, the TTA architecture has been fused with CAN to create TTCAN* for wired communication. This is more suitable as a networking environment for DNRTS. Other networking environments such as the token bus (IEEE 802.4) or polling protocols can deliver deterministic access delays to network nodes. These work well when all components at a node send messages with the same period. However, when network nodes have control components computing at multiple rates, TTA provides the best support.

11.7.1 DNRTS

Research over the last decade indicates that engineers should be able to use high-level tools like Simulink or Lustre to program DNRTS. We explain this abstractly in terms of the computing semantics of digital control in Section 11.2. For each variable in a program and a sequence of values for each of its input variables, the semantics gives us the sequence of values taken by the variable and the timing of each value (see, for example, Figure 11.5). We will refer to the set of value sequences as the logical aspect of the semantics and the timing as the real-time aspect of the semantics. Preserving the logical aspect in a distributed environment is now well understood [1,38]. It turns out to be possible under pretty weak assumptions on the network. Preserving the real-time aspect is not as well understood and appears challenging. We will discuss this in the context of the TTA and comment briefly on other networking environments. Among current networking environments, this one may be the best suited for DNRTS.

Suppose we have a three-block control system as shown in Figure 11.4. Assume each block is on a computing node and the input–output connections between the blocks, denoted by the arrows, need to be network channels. This sort of control schematic is easily specified in Simulink without the network channels. In Simulink, each block will also have a period at which it reads and writes its outputs. The literature gives insight into the following questions:

- What sort of channels can enable us to preserve the logical aspect of the semantics?
- What sort of channels can enable us to preserve the real-time aspect of the semantics?

It turns out that the logical aspect can be supported by many channels. For example, any reliable first in first out (FIFO) channel works. In other words, a TCP channel can do the job.

Figure 11.17 illustrates the logical aspect of the semantics in Figure 11.5. We care only about the sequence of values for each variable but not about the vector of values represented by the columns in Figure 11.5. An execution that preserves the logical aspect of the semantics will be one that preserves the sequence of values for each variable. It may not preserve any ordering of values across the variables, i.e., the vector.

If the components are connected by reliable FIFO channels, there is a conceptually simple compilation scheme able to preserve the logical semantics. This is illustrated by Figure 11.18 in the context of the three-block example. Basically, each component must be compiled to block until it receives the right number of inputs on each channel. Since all blocks in Figure 11.18 are assumed to have the same rate, this means block *C* should block until it receives the first inputs on both the *y* and *w* ports, compute the first value of *v* as soon as it does so, and then block until it receives another pair of inputs. If the second value on port *w* is delayed relative to the second value on port *C*, then *C* must block for *w*.

```

u → 2, 4, ., ., ., 2, 4, ., 2, ...
y → 4, 8, 4, ., ., 8, 4, ., ., ...
z → 1, ., 2, 1, 2, 1, ., ., ., ...
w → 2, 4, ., 2, ., 4, ., ., ., ...

```

FIGURE 11.17 Logical behavior of the three-block control system.

*TTCAN standard ISO 11898-4.

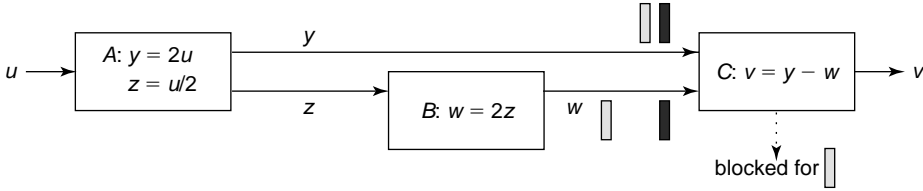


FIGURE 11.18 A compilation scheme to preserve the logical semantics over FIFO channels.

Locally, each program is required to behave exactly like the usual synchronous program illustrated by Program 11.4. Components on different network nodes can be driven entirely by the arrival of messages on the input–output channels. Thus there is no global clock or scheduler, rendering the architecture globally asynchronous and locally synchronous, abbreviated GALS by Benveniste et al. [1]. If the synchronous subsystem on each network node is compiled to accept its inputs in any order, this kind of compilation is also modular. Any change to the program on a network node will require recompilation of that node alone. It may also be noted that the same scheme has been shown to work for programs with feedback loops provided each loop has a delay or is strictly causal. In other words, each loop should include at least one component whose output at time k depends only on inputs up to time $k - 1$.

Synchronous programming languages like Esterel subsume asynchronous programming by permitting signals to be either present or absent at each execution of the program. If one seeks to distribute such programs without controlling execution with respect to one master clock, the potential for signals to be absent poses a conundrum. The absence of a signal may be meaningless without the master clock. While all such programs cannot be distributed in a GALS architecture, a subclass identified as the isochronous programs can be distributed without a master clock. An isochronous program pair is one where in any state pair from the two programs, unification of the two states over the present common variables implies unification over the entire set of common variables [1]. In other words, the values of signals supplied by the environment is sufficient to determine the absence of the other signals thus rendering any checking of signal presence with respect to any master clock unnecessary.

We now turn to our second question, i.e., preservation of the real-time aspect. Meeting periodic execution deadlines in DNRTS requires two kinds of schedulability. Computation must be schedulable at each computing node, and communication must be schedulable between nodes. Not only must the intercontroller channels be reliable and FIFO, they must also provide access to computing nodes at scheduled times. Alternatively, the communication channels between computing nodes must be underutilized enough to support a zero communication time abstraction, i.e., a node communicates its output to the others as soon as it is ready. A high bandwidth wired Ethernet may support such an abstraction. In this case, the compilation problem may be the same as that discussed in Sections 11.5 and 11.6.

The joint scheduling of computation and communication has been investigated over TTA [7]. TTA is well suited to the computing abstractions of digital control. It supports distributed implementations built upon a synchronous bus delivering to computing nodes a global fault-tolerant clock. Each computing node connects to the bus using a network card running the time-triggered protocol (TTP). The bus is required to have a communication schedule determined *a priori*. This is called the *message description list* (MEDL). It specifies which network node will transmit which message at which time on a common global timeline. The timeline is divided into cycles. The cycle is the unit of periodic operation, i.e., the composition of each cycle is the same. The MEDL specifies the composition of the cycle. Within a cycle, there are rounds of identical length. Each round is composed of slots. These can have different lengths. Each slot is given to a network node. Within the slot, the node sends a frame composed of one or more messages. Thus, the objective of communication scheduling when distributing over TTA is specification of the MEDL.

The approach in [7] compiles both Simulink and Lustre over TTA. Simulink programs are handled by translating them to Lustre. The control engineer is assumed to specify the allocation of components

or blocks to nodes. One or more components at each node may have input–output connections to components on other computing nodes. The compilation process starts with components annotated with deadlines, worst-case execution time assumptions, and the real-time equivalents of the logical clocks driving the component. The deadlines are relative. For example, if a component has an output y and an input u , a deadline would specify the maximum time that may elapse between the availability of u and the availability of y . The compilation process can be thought of as a two-step process with the first step handling the logical aspect and the second the real-time aspect.

The first step of compilation, called the *analyzer* [7], builds the syntax tree and global partial order across all the components in all the nodes in the system. This partial order accounts for all the dataflow dependencies between components and represents constraints that will preserve the logical aspect of the semantics. The results of this stage of compilation with the annotations previously mentioned and the partial order constraints can be passed to the next compiler called the *scheduler* because it generates the computing schedule at each node and the MEDL for the bus. The scheduling problem is a mixed integer linear program (MILP) and computationally difficult. To reduce complexity, the developers allow the fine-grain partial order over all components to be lumped into coarser grain supercomponents that include some of the original components as subcomponents. The scheduler then works with the supercomponents, moving progressively smaller supercomponents, if the coarser representations are not schedulable.

11.7.2 NNRTS

In this subsection, we discuss networked real-time systems in which intercomponent dataflows experience frequent jitter or loss. We emphasize jitter because the problems arise because of the variability of delay. DNRTS also have delay but the delay is predictable. When controllers communicate over the Internet or local wireless channels, there are no techniques to eliminate both losses and jitter. Essentially, in such networks, one must choose between loss and jitter. One cannot eliminate both. Protocols preventing loss do so by retransmission that creates variable delay or jitter.

The situation is illustrated by Figure 11.19, which shows the different kinds of NNRTS channels encountered in practice. A controller may output a periodic message stream. However, the corresponding message process observed at the input port of the receiving controller may be asynchronous. If the messages arrive through a TCP channel, they may experience different delays though all messages will be received in the right order. The TCP channel is FIFO. Alternatively, if the messages arrive through a UDP channel in a one-hop wireless LAN, some may be lost while those that arrive do so with delays small enough to be abstracted away. This combination of loss and negligible delay has received wide attention in the information theory literature, where it is referred to as the erasure channel. The fourth line in the figure illustrates

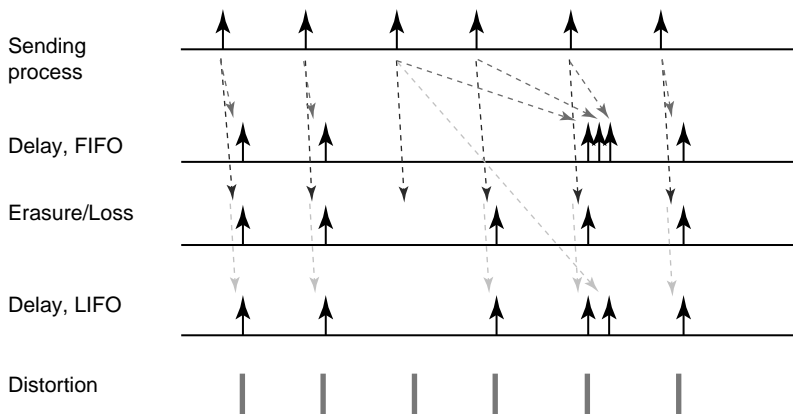


FIGURE 11.19 Message reception processes in NNRTS.

the last in first out (LIFO) channel with delays. When messages report sensor measurements or the actuator commands of other controllers, the most recent values are generally more useful than earlier ones. Therefore, it makes sense to have the network give priority to delivery of the information generated most recently instead of enforcing a first in first out order. Finally, if communication were analog, one might assume a distorted version of the transmitted signal is always received without loss or delay. While this formulation is of considerable theoretical interest [28,35], we will not discuss it further because control with analog communication is so uncommon in current engineering practice.

When intercomponent dataflow is jittery, even the logical aspect of the semantics assumed by digital control cannot be preserved in the manner used for DNRTS. Suppose we were controlling one truck to follow another at some safe distance. The control commands of the following truck would be functions of quantities like the position, velocity, or acceleration of the truck in front (see [34]), information we assume flows over a network. If we were to compile and distribute the program as prescribed for DNRTS, the module controlling the following vehicle would block until every message from the leading vehicle module is received. Since the blocking time would be as jittery as the access delay, the following vehicle would issue actuator commands at irregular intervals. Blocking real-time controllers for unpredictable durations is usually unacceptable.

Having recognized the impact of the loss or delay on control performance, control theory has moved away from the perfect intercontroller channel abstraction discussed in Section 11.2. It has introduced the physicality of intercontroller links into its semantics and produced design techniques to account for the losses or delays. As yet, it remains unclear how real-time programming might produce tools to facilitate embedded programming in these new semantics. Engineers work at a low level with C, UDP or TCP sockets, and OS system calls, rather than at the high level envisaged in the Simulink to Lustre to TTA chain described for DNRTS.

A new semantics for NNRTS is illustrated in Figure 11.20. Developments in control theory show that performance is considerably enhanced by abandoning the simplified intercontroller connection abstraction for the more sophisticated scheme in Figure 11.20 when there is a nondeterministic intercomponent dataflow. The three-vehicular system examples we have discussed all use this scheme. One puts a component at the receiver end that receives an intermittent asynchronous message stream from the UDP socket at its input but produces a periodic synchronous signal at its output. At the most basic level, this component is as an asynchronous-to-synchronous converter. It enables the entire system downstream, for example, the warning or control systems, to be designed in the classical way, e.g., as a digital control system computing, in this case, at 50 Hz.

In Figure 11.20, we refer to this converter component as an *estimator*. This is because of control theory showing it is wise to give it semantics richer than that of just an asynchronous-to-synchronous converter. For example, the component output may compute or approximate, at each time, the expected value of the message that might be received over the channel at the time conditioned on all past received values. In many cases, this is a type of Kalman Filter [22,30,33], aptly named by Sinopoli et al. as a “Kalman filter

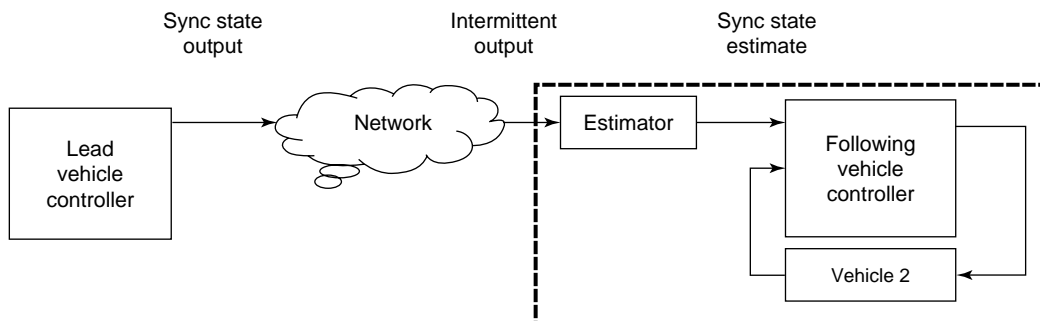


FIGURE 11.20 Controlling in the face of network losses.

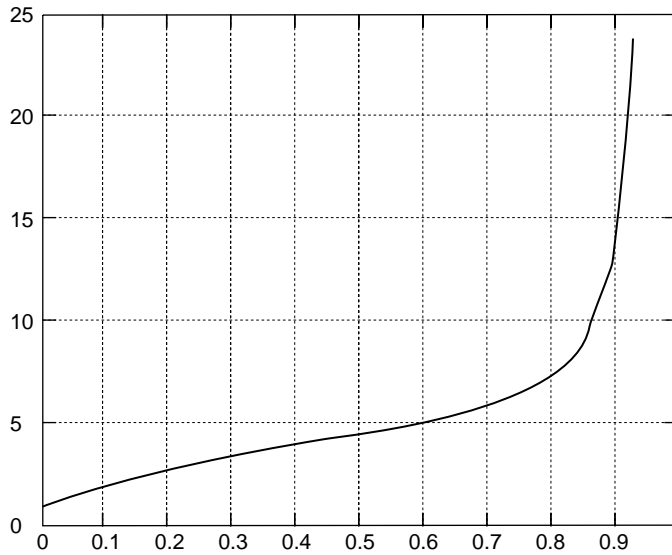


FIGURE 11.21 Performance obtained by using an estimator.

with intermittent observations.” Since messages in these systems report on the state of dynamical systems, and the state is correlated, this is a viable approach. The purpose of this converter is to help the receiver deal with the channel distorting the output of the sender.

Figure 11.21 [30] is a quantification of the performance robustness obtained by using the scheme in Figure 11.20. It is computed for one car following another. The vehicle model is a linear approximation of the demonstration vehicles. The objective of control is to maintain constant spacing between the cars. The spacing deviates from the desired constant value because of acceleration or deceleration by the leading vehicle. The following vehicle takes some time to speed up or slow down, which leads to spacing errors. The vertical axis in the figure plots:

$$\frac{\text{spacing error}}{\text{leading vehicle acceleration}}$$

One would like this ratio to be small. The horizontal axis plots the loss probability in the intercontroller communication channel. It is assumed the loss process is Bernoulli, i.e., each packet is lost with probability p or received with probability $1 - p$. The figure shows that as p gets larger the gain gets larger, i.e., the performance deteriorates. The control system computes an estimate of what might have been received whenever a message is lost and uses it to compute the actuator command. The value of doing this is made evident by the shape of the plot. The performance remains good for a long time and breaks down only at high loss rates.

Control is developing effective methods for NNRTS by breaking away from its established computing abstractions as presented in Section 11.2. The new abstractions are still debated and will take time to settle. Nevertheless, as the class of control systems distributed over the Internet or wireless LANs is growing, the real-time programming community should accelerate its effort to produce a tool chain for the clean and correct real-time programming of NNRTS.

References

1. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

2. A. Benveniste, P. Caspi, P. Guernic, H. Marchand, J. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2211 of LNCS. Springer, 2001.
3. G. Berry. *The Esterel Language Primer, version v5 91*. <http://www.esterel-technologies.com/technology/scientific-papers/>, 1991.
4. G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
5. A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.
6. G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2005.
7. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: A layered approach for distributed embedded applications. In *Proc. ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 153–162. ACM Press, 2003.
8. M. Ergen, D. Lee, R. Sengupta, and P. Varaiya. Wireless token ring protocol. *IEEE Transactions on Vehicular Technology*, 53(6):1863–1881, 2004.
9. E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–39. ACM Press, 2005.
10. L. Feng-Li, J. Moyne, and D. Tilbury. Network design consideration for distributed control systems. *IEEE Transactions on Control Systems Technology*, 10(2):297–307, 2002.
11. A. Ghosal, T.A. Henzinger, D. Iercan, C.M. Kirsch, and A.L. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*. ACM Press, 2006.
12. A. Ghosal, T.A. Henzinger, C.M. Kirsch, and M.A.A. Sanvido. Event-driven programming with logical execution times. In *Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC)*, volume 2993 of LNCS, pages 357–371. Springer, 2004.
13. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.
15. R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7), July 2003.
16. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326, 2002.
17. T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
18. T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of LNCS, pages 76–92. Springer, 2002.
19. T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In *Proc. International Conference on Embedded Software (EMSOFT)*, volume 2855 of LNCS, pages 241–256. Springer, 2003.
20. T.A. Henzinger, C.M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2005.
21. T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, February 2003.
22. R. Kalman. A new approach to liner filtering and prediction problems. *Transactions of the ASME, Journal of Basic Engineering*, 82D:34–45, 1960.
23. K.H. Kim. A non-blocking buffer mechanism for real-time event message communication. *Real-Time Systems*, 32(3):197–211, 2006.

24. C.M. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 61–75. Springer, 2002.
25. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.
26. H. Kopetz and J. Reisinger. NBW: A non-blocking write protocol for task communication in real-time systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1993.
27. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.
28. A. Sahai. *Anytime Information Theory*. PhD thesis, Massachusetts Institute of Technology, 2001.
29. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a “safe” subset of Simulink/Stateflow into Lustre. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*, pages 259–268. ACM Press, 2004.
30. P. Seiler and R. Sengupta. An H-infinity approach to networked control. *IEEE Transactions on Automatic Control*, 50(3):356–364, 2005.
31. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
32. I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 57–67, 2004.
33. B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M.I. Jordan, and S.S. Sastry. Kalman filtering with intermittent observations. *IEEE Transactions on Automatic Control*, 49(9):1453–1464, 2004.
34. D. Swaroop. *String Stability of Interconnected Systems: An Application to Platooning in Automated Highway Systems*. PhD thesis, University of California at Berkeley, 1994.
35. S. Tatikonda. *Control Under Communication Constraints*. PhD thesis, Massachusetts Institute of Technology, 2000.
36. J. Templ. TDL specification and report. Technical Report T004, Department of Computer Sciences, University of Salzburg, November 2004.
37. S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or EDF schedulers. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*, pages 353–360. ACM Press, 2005.
38. M. Zennaro and R. Sengupta. Distributing synchronous programs using bounded queues. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*. ACM Press, 2005.

12

Real-Time Java

Andy Wellings

University of York

Alan Burns

University of York

12.1	Introduction	12-1
12.2	Background on the RTSJ	12-2
12.3	Scheduling Objects and Scheduling	12-3
	Rationale • Approach • Outstanding Issues and Future Directions	
12.4	Resource Sharing and Synchronization	12-9
	Rationale • Approach • Outstanding Issues and Future Directions	
12.5	Time Values and Clocks	12-11
	Rationale • Approach • Outstanding Issues and Future Directions	
12.6	Memory Management	12-13
	Rationale • Approach • Outstanding Issues and Future Directions	
12.7	Conclusions	12-17

12.1 Introduction

Since its inception in the early 1990s, there is little doubt that Java has been a great success. However, the language does have serious weaknesses both in its overall model of concurrency and in its support for real-time systems. Consequently, it was initially treated with disdain by much of the real-time community. Nevertheless, the perceived advantages of Java (from a software engineering perspective) over languages like C and C++, coupled with the failure of Ada to make strong inroads into the broad real-time and embedded system markets, resulted in several attempts to extend the language so that it is more appropriate for a wide range of real-time systems. The simplest extension was Real-Time Java Threads (Miyoshi et al., 1997), which provided rudimentary real-time support. Another proposal was Portable Executive for Reliable Control (PERC) (Nilsen, 1998) that provided both high-level abstractions for real-time systems and low-level abstractions to access the hardware (included is real-time garbage collection). In Communication Threads for Java (Hilderink, 1998), an approach was proposed that was based on the CSP algebra, the Occam2 language, and the Transputer microprocessor. Others attempted to integrate the Java Virtual Machine (JVM) into the operating system (e.g., GVM [Back et al., 1998]) or to provide a hardware implementation of the JVM (e.g., picoJava-I [McGhan and O'Connor, 1998]).

Unfortunately, much of this early work was fragmented and lacked clear direction. In the late 1990s, under the auspices of the U.S. National Institute of Standards and Technology (NIST), approximately 50 companies and organizations pooled their resources and generated several guiding principles and a set of requirements (Carnahan and Ruark, 1999) for real-time extensions to the Java platform. Among the guiding principles was that Real-Time Java should take into account current real-time practices and facilitate advances in the state of the art of real-time systems implementation technology. There were two competing responses to the NIST requirements.

- An initiative backed by Sun Microsystems and IBM: the Real-Time Specification for Java or RTSJ (Bollella et al., 2000). RTSJ required no changes to the Java language but did require a modified JVM and a standard package of classes.
- An initiative backed by the J Consortium (HP, Siemens, Newmonics): the Real-Time Core Extension for the Java Platform (RT Core)(J Consortium, 2000). Real-time Core was based on the PERC system. It consisted of two separate APIs: the Baseline Java API for non real-time Java threads, and the Real-Time Core API for real-time tasks. Some language changes to Java were proposed.

Both of these approaches have their advantages and disadvantages. However, the language changes proposed by the J Consortium failed to find favor with the wider Java community and the work has dropped away. In contrast, the RTSJ has continued to evolve (the initial version, 1.0, has been rewritten, version 1.0.1, to make the designers' intention clearer and to tighten up the semantics) (Dibble et al., 2006), implementations have become available (e.g., the Reference Implementation by TimeSys—<http://www.timesys.com>, the Jamaic VM from aicas—<http://www.aicas.com>, OVM (an open source version)—<http://www.ovmj.org>, and Mackinac from Sun—<http://research.sun.com/projects/mackinac/>), and supporting tutorials have appeared (Dibble, 2002; Wellings, 2004). Consequently, it is the RTSJ that is the focus of this chapter.

The chapter is structured as follows. First, some background is given of the development of the RTSJ, and a number of weak areas in Java identified for enhancements. Each of the main areas is then considered in its own section: the rationale for the enhancement is given first, followed by consideration of the approach taken. Each section then ends with a discussion of the outstanding issues and future directions in that area.

12.2 Background on the RTSJ

The RTSJ has several “guiding principles” that have shaped and constrained the development of the specification. These include requirements to

- Be backward compatible with non real-time Java programs
- Support the principle of “Write Once, Run Anywhere” but not at the expense of predictability
- Address current real-time system practices and allow future implementations to include advanced features
- Give priority to predictable execution in all design trade-offs
- Require no syntactic extensions to the Java language, and
- Allow implementer's flexibility

The requirement for no syntactic enhancements to Java has had a strong impact on the manner in which the real-time facilities can be provided. In particular, all facilities have to be provided by an augmented virtual machine and a library of classes (and interfaces). In places, this has had a marked effect on the readability of real-time applications.

The RTSJ enhances Java in the following areas:

- Schedulable objects and scheduling—preemptive priority-based scheduling of real-time threads and asynchronous event handlers all within a framework allowing online feasibility analysis;
- Synchronization and resource sharing—support for the well-known priority inversion minimization algorithms;
- Time values and clocks—high-resolution time values and a real-time clock;
- Memory management—a complementary model based on the concept of memory regions (Tofte and Talpin, 1997; Tofte et al., 2004) (called scoped memory areas) allows the reclamation of objects without the vagaries of garbage collection;
- Asynchronous transfer of control—extensions to the Java thread interrupt mechanism allowing the controlled delivery and handling of asynchronous exceptions (not considered in this chapter; see Brosgol et al. [2002] and Wellings [2004]);

- Physical and raw memory access—allowing more control over allocation of objects in different types of memory and interfacing to device drivers (not considered in this chapter; see Wellings [2004]).

The intention is that the RTSJ should be capable of supporting both hard and soft real-time components. The hard real-time components should never access the Java heap but use the alternative memory management support. This way, they will never be delayed by garbage collection activities.

It should be stressed that the RTSJ is only intended to address the execution of real-time Java programs on single processor systems. It attempts not to preclude execution on shared-memory multiprocessor systems but it currently has no facilities directly to control, for example, allocation of threads to processors.

12.3 Scheduling Objects and Scheduling

12.3.1 Rationale

Scheduling of threads is a key aspect for all real-time systems. Standard Java allows each thread to have a priority that can be used by the JVM when allocating processing resources. However, Java offers no guarantees that the highest-priority runnable thread will be the one executing at any point in time. This is because a JVM may be relying on a host operating system to support its threads. Some of these systems may not support preemptive priority-based scheduling. Furthermore, Java only defines 10 priority levels, and an implementation is free to map these priorities onto a more restricted host operating system's priority range if necessary. The weak definition of scheduling and the restricted range of priorities mean that Java programs lack predictability and, hence, Java's use for real-time systems' implementation is severely limited. Consequently, this is a major area that needs to be addressed.

12.3.2 Approach

The RTSJ incorporates the notion of a *schedulable* object rather than simply considering threads. A schedulable object is any object that implements the `Schedulable` interface. The current specification essentially provides two types of objects that implement this interface, `RealtimeThreads` and `AsyncEventHandlers`.^{*} Objects that implement the `Schedulable` interface have the following associated attributes (represented by classes).

ReleaseParameters—Release requirements are specified via the `ReleaseParameters` class hierarchy. Scheduling theories often identify three types of releases: periodic (released on a regular basis), aperiodic (released at random), and sporadic (released irregularly, but with a minimum time between each release). These are represented by the `PeriodicParameters`, `AperiodicParameters`, and `SporadicParameters` classes, respectively. All release parameter classes encapsulate a cost and a deadline (relative time) value. The cost is the maximum amount of CPU time (execution time) needed to execute the associated schedulable object every time it is released. The deadline is the time at which the object must have finished executing the current release; it is specified relative to the time the object was released. `PeriodicParameters` also include the start time for the first release and the time interval (period) between releases. `SporadicParameters` include the minimum interarrival time between releases. Event handlers can be specified for the situation where a deadline is missed or where the processing resource consumed becomes greater than the cost specified. However, there is no requirement for a real-time JVM to monitor the processing time consumed by a schedulable object. If it does, then there is a requirement that a schedulable object be given no more than “cost” processing units each release.

^{*}The RTSJ allows variants of these called *no-heap* schedulable objects. These are intended to support hard real-time activities. The programmer must ensure that they never access the heap (otherwise, an exception is thrown). No-heap schedulable objects can never be delayed by garbage collection activities.

SchedulingParameters—Scheduling parameters are used by a scheduler to determine, which object is currently the most eligible for execution. This abstract class provides the root class from which a range of possible scheduling criteria can be expressed. The RTSJ defines only one criterion that is based on priority (as would be expected, it is silent on how these priorities should be assigned). **ImportanceParameters** allow an additional numerical scheduling metric to be assigned; it is a subclass of the **PriorityParameters** class. Although the RTSJ specifies a minimum range of real-time priorities, it makes no statement on the allowed values of the important parameter.

MemoryParameters—Giving the maximum amount of memory used by the object in its default memory area, the maximum amount of memory used in immortal memory, and a maximum allocation rate of heap memory. An implementation of the RTSJ may optionally enforce these maximums and throw exceptions if they are violated (see Section 12.6).

ProcessingGroupParameters—This allows several schedulable objects to be treated as a group and to have an associated period, cost, and deadline. When used with aperiodic activities, they allow their impact to be bounded.

The methods in the **Schedulable** interface can be divided into three groups.

- Methods that will communicate with the scheduler and will result in the scheduler either adding or removing the schedulable object from the list of objects it manages (called its *feasibility set*), or changing the parameters associated with the schedulable object (but only if the resulting system is feasible).
- Methods that get or set the parameter classes associated with the schedulable object (e.g., the release parameters). If the parameter object set is different from the one currently associated with the schedulable object, the previous value is lost and the new one will be used in any future feasibility analysis performed by the scheduler. Note, these methods do not result in feasibility analysis being performed and the parameters are changed even if the resulting system is not feasible.
- Methods that get or set the scheduler. For systems that support more than one scheduler, these methods allow the scheduler associated with the schedulable object to be manipulated.

Changing the parameters of a schedulable object while it is executing can potentially undermine any feasibility analysis that has been performed and cause deadlines to be missed. Consequently, the RTSJ provides methods that allow changes of parameters to occur only if the new set of schedulable objects is feasible. In these situations, the new parameters do not have an impact on a schedulable object's executions until its next release. Some applications will need certain changes to take place immediately (i.e., to affect the current release); for example, priority changes. These are catered for by changing the individual field in the current parameters.

In all cases, the scheduler's feasibility set is updated. Of course, an infeasible system may still meet all its deadlines if the worst-case loading is not experienced (perhaps the worst-case phasing between the threads does not occur or threads do not run to the worst-case execution time).

As well as generalizing threads to schedulable objects, the RTSJ also generalizes the notion of priority to one of execution eligibility. This is so that it can leave the door open for implementations to provide other schedulers (say value-based schedulers or earliest-deadline-first schedulers). All schedulers, however, must follow the framework specified by the following abstract **Scheduler** class.

```
package javax.realtime;
public abstract class Scheduler {
    // constructors
    protected Scheduler();

    // methods
    protected abstract boolean addToFeasibility(Schedulable schedulable);
    protected abstract boolean removeFromFeasibility(Schedulable schedulable);
    public abstract boolean isFeasible();
```

```

public abstract boolean setIfFeasible(Schedulable schedulable,
    ReleaseParameters release, MemoryParameters memory);
public abstract boolean setIfFeasible(Schedulable schedulable,
    ReleaseParameters release, MemoryParameters memory,
    ProcessingGroupParameters group);

public static Scheduler getDefaultScheduler();
public static void setDefaultScheduler(Scheduler scheduler);
}

```

Each scheduler maintains a set of schedulable objects that it manages. It may perform some form of feasibility analysis on that set to determine if the objects will meet their deadlines. The protected methods are called from schedulable objects (via the `Schedulable` interface) and enable those objects to be unconditionally added to or removed from the list, the Boolean return values indicate whether the resulting system is feasible. The `setIfFeasible` methods provide atomic operations that allows the parameters of a schedulable object to be changed only if it does not affect the feasibility of the whole set of objects being managed by the scheduler. The `isFeasible` method can be called to determine if the current set of objects can be scheduled to meet their deadlines.

The only scheduler that the RTSJ fully defines is a priority scheduler, which can be summarized by describing its scheduling policy, its dispatching mechanism, and its schedulability analysis.

Scheduling policy. The priority scheduler

- Schedules schedulable objects according to their release profiles and scheduling parameters
- Supports the notion of base and active priority
- Orders the execution of schedulable objects on a single processor according to the active priority
- Supports a real-time priority range of at least 28 unique priorities (the larger the value, the higher the priority)
- Requires the programmer to assign the base priorities (say, according to the relative deadline of the schedulable object)
- Allows base priorities to be changed by the programmer at runtime
- Supports priority inheritance or priority ceiling emulation inheritance for synchronized objects
- Assigns the active priority of a schedulable object to be the higher of its base priority and any priority it has inherited
- Deschedules schedulable objects when they overrun their CPU budgets
- Releases event handlers for cost overrun and deadline miss conditions
- Enforces minimum interarrival time separation between releases of sporadic schedulable objects

Dispatching mechanism. The priority scheduler

- Supports preemptive priority-based dispatching of schedulable objects—the processor resource is always given to the highest-priority runnable schedulable object
- Does not define where in the run queue (associated with the priority level) a preempted object is placed; however, a particular implementation is required to document its approach and the RTSJ recommends that it be placed at the front of the queue
- Places a blocked schedulable object that becomes runnable, or has its base priority changed, at the back of the run queue associated with its (new) active priority
- Places a schedulable object which performs a `Thread.yield` method call at the back of the run queue associated with its priority
- Does not define whether schedulable objects of the same priority are scheduled in FIFO, round-robin order or any other order

Schedulability (feasibility) analysis. The `PriorityScheduler` requires no particular analysis to be supported.

12.3.2.1 Real-Time Threads

The two types of schedulable objects supported by the RTSJ are defined by the `RealtimeThread` and `AsynEventHandler` classes. As an example of the facilities provided, this section considers real-time threads in a little more depth—an abridged version of the defining class is given below.

```
package javax.realtime;
public class RealtimeThread extends Thread
    implements Schedulable {

    // constructors
    public RealtimeThread();
    public RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release,
        MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group,
        Runnable logic);

    // Methods which implement the Schedulable interface
    ...

    // Methods which manipulate the memory area stack
    ....

    public void interrupt();
    public static void sleep(Clock clock, HighResolutionTime time)
        throws InterruptedException;
    public static void sleep(HighResolutionTime time)
        throws InterruptedException;

    public void start();
    public static RealtimeThread currentRealtimeThread();

    public static boolean waitForNextPeriod()
    public static boolean WaitForNextPeriodInterruptible()
        throws InterruptedException;

    public void deschedulePeriodic();
    public void schedulePeriodic();
    ...
}
```

As the `RealtimeThread` class implements the `Schedulable` interface, there is a set of methods for meeting its requirements. There is also a set of methods that deal with memory areas. A real-time thread can enter into one or more memory areas. These areas are stacked, and the stack can be manipulated by the owning thread only (see Section 12.6).

Although `RealtimeThread` extends `Thread`, it only overrides, overloads, or redefines a few of its methods. The `interrupt` method is redefined so that it can implement asynchronous transfer of control. Two new `sleep` methods are defined to interface with the new time and clock classes. The `start` method is also redefined so that it can record that any associated memory area that has now become active. The real-time counterpart of `Thread.currentThread` is defined to allow the currently executing real-time thread to be identified. An exception is thrown if the currently executing thread is not a real-time thread (it may be, for example, a conventional Java thread).

The final set of methods are for the special case where the real-time thread has periodic release parameters. The `waitForNextPeriod` method suspends the thread until its next release time (unless the thread has missed its deadline). The call returns true when the thread is next released.

The `deschedulePeriodic` method will cause the associated thread to block at the end of its current release (when it next calls `waitForNextPeriod`). It will then remain blocked until `schedulePeriodic` is called (by another thread). The thread is now suspended waiting for its next

period to begin, when it is released again. Of course, if the associated thread is not a periodic thread, it will not call `waitForNextPeriod` and, therefore, `deschedulePeriodic` and `schedulePeriodic` will have no effect. When a periodic thread is rescheduled in this manner, the scheduler is informed so that it can remove or add the thread to the list of schedulable objects it is managing.

12.3.2.2 Asynchronous Events and Their Handlers

An alternative to thread-based programming is event-based programming. Each event has an associated handler. When events occur, they are queued and one or more server threads take events from the queue and execute their associated handlers. When a server has finished executing a handler, it takes another event from the queue, executes the handler, and so on. The execution of the handlers may generate further events. With this model, there are only server threads. Arguably, for large reactive real-time systems, an event-based model is more efficient than one that has to dedicate a thread to event possible external event (Ousterhout, 2002; van Renesse, 1998).

In an attempt to provide the flexibility of threads and the efficiency of event handling, the RTSJ has introduced the notion of real-time asynchronous events and their associated handlers. However, the specification is silent on how these events can be implemented and how their timing requirements can be guaranteed.

The RTSJ views asynchronous events as dataless occurrences that are either fired (by the program) or associated with the triggering of interrupts (or signals) in the environment. One or more handlers can be associated with (attached to) a single event, and a single handler can be associated with one or more events. The association between handlers and events is dynamic. Each handler has a count (called `fireCount`) of the number of outstanding occurrences. When an event occurs (is fired by the application or is triggered by an interrupt), the count is atomically incremented. The attached handlers are then released for execution. As handlers support the `Schedulable` interface, they are scheduled in contention with real-time threads.

An abridged version of the `AsyncEvent` class is given below:

```
package javax.realtime;
public class AsyncEvent {

    // constructors
    public AsyncEvent();

    // methods
    public void addHandler(AsyncEventHandler handler);
    public void removeHandler(AsyncEventHandler handler);
    public void setHandler(AsyncEventHandler handler);
    public boolean handledBy(AsyncEventHandler target);
    public void bindTo(String happening);
    public void unBindTo(String happening);
    public void fire();
}
```

Most of the methods are self-explanatory. The `bindTo/unBindTo` method associates/disassociates an external “happening” (an interrupt or an operating system signal) with a particular event.

An event that is created in scoped memory and then subsequently bound to an external happening has the effect of increasing the reference count associated with that scoped memory area. The reference count is decremented when the happening is unbound. This is to ensure that the scoped memory area is active when interrupts occur.

Event handlers are defined using the `AsyncEventHandler` class hierarchy. Application-defined handlers are created either by subclassing this class and overriding the `handleAsyncEvent` method or by passing an object that implements the `Runnable` interface to one of the constructors. Once attached

to an event, in the former case, the overridden `handleAsyncEvent` method is called whenever the event's fire count is greater than zero. In the latter case, the `run` method of the `Runnable` object is called by the `handlerAsyncEvent` method.

The `AsyncEventHandler` class has several constructor methods that allow the passing of various parameter classes associated with a schedulable object (scheduling, release, memory, and processing group parameters; whether the schedulable object is a no-heap object and any memory area that should be set up as the default memory area) along with the optional `Runnable` object. An abridged version of the class is given below.*

```
package javax.realtime;
public class AsyncEventHandler implements Schedulable {

    // Example constructors
    public AsyncEventHandler();
    public AsyncEventHandler(Runnable logic);
    public AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
        MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group,
        Runnable logic);

    // Methods needed to support the Schedulable interface not shown
    ...

    // Methods needed for handling the associated event
    protected int getAndClearPendingFireCount();
    protected int getAndDecrementPendingFireCount();
    protected int getPendingFireCount();
    public void handleAsyncEvent();
    public final void run();
}
```

12.3.3 Outstanding Issues and Future Directions

The RTSJ has made a good attempt to generalize real-time activities away from real-time threads toward the notion of schedulable objects, and intends to provide a framework from within which priority-based scheduling with online feasibility analysis can be supported. The framework is, however, incomplete and current specification work is aimed at removing some of its limitations.

- The current specifications support a range of `ReleaseParameters` that can be associated with schedulable objects. Asynchronous event handlers support the full range; however, there are no appropriate mechanisms to support associating `AperiodicParameters` and `SporadicParameters` with real-time threads. This is because there is no notion of a real-time thread waiting for a next release. It can only call the `waitForNextPeriod` method, which throws an exception if the real-time thread does not have `PeriodicParameters`. This also means that it is impossible to support the deadline miss and cost overrun detection mechanisms. Also aperiodic and sporadic threads cannot be descheduled in the same way as periodic threads. The current workaround is either not to use aperiodic/sporadic real-time threads or to assume that aperiodic/sporadic real-time threads are single shot (i.e., they are released by a call to the `start` method and they complete by termination—this is the approach adopted by the current specification).

The next version of the RTSJ will provide better support for aperiodic and sporadic real-time threads.

*See Dibble (2002), Wellings (2004), and Wellings and Burns (2002) for a full discussion on the asynchronous event-handling model.

- The vast majority of real-time operating systems support fixed-priority preemptive scheduling with no online feasibility analysis. However, as more and more computers are being embedded in engineering (and other) applications, there is a need for more flexible scheduling. Broadly speaking, there are three ways to achieve flexible scheduling (Wellings, 2004):
 - *Pluggable schedulers*: In this approach, the system provides a framework into which different schedulers can be plugged. The CORBA Dynamic Scheduling (Object Management Group, 2003) specification is an example of this approach. Kernel loadable schedulers also fall into this category.
 - *Application-defined schedulers*: In this approach, the system notifies the application every time an event occurs, which requires a scheduling decision to be taken. The application then informs the system, which thread should execute next. The proposed extensions to real-time POSIX support this approach (Aldea Rivas and Gonzalez Harbour, 2002).
 - *Implementation-defined schedulers*: In this approach, an implementation is allowed to define alternative schedulers. Typically, this would require the underlying operating system or runtime support system (virtual machine, in the case of Java) to be modified. Ada 95 adopts this approach, albeit within the context of priority-based scheduling.

Support for new scheduling algorithms was a fundamental design goal of the RTSJ. Ideally, it would have supported new pluggable schedulers from third parties and end users, but in the end the goal was to let JVM implementors add new schedulers. This approach allows the specification to depend on the scheduling implemented in the underlying operating system, and it lets the new scheduler use JVM APIs that are invisible to anyone but the implementor. However, the specification does not provide a well-defined interface with which new schedulers can be easily added.

Inevitable, when other schedulers are supported, it becomes necessary to consider multiple schedulers within the same RT-JVM. The current RTSJ does not forbid cooperating schedulers, but it makes little effort to provide a useful framework for them. It can be argued that as priority-based dispatching is so ingrained in the specification, any approach to providing multiple schedulers must be from within a priority-based framework. An example of such a framework for the RTSJ is given by Zerkelidis and Wellings (2006). In the next chapter, an Ada framework will be given.

- **Cost enforcement and blocking.** Cost enforcement is critical to many real-time systems, and it is designed into the RTSJ as an optional feature (as it is not supported by many real-time operating systems). Unfortunately, the reference implementation did not implement this feature and the 1.0 version of the specification was published with that aspect untested. This is an area that has received considerable attention (Wellings et al., 2003) and the current version (1.0.1) is much more complete although it still requires some further attention during deadline overrun conditions (Marchi dos Santos and Wellings, 2005).

Blocking is another area where the RTSJ is under specified. Currently, there is no notion of the time a schedulable object is blocked when accessing resources. This is needed for all forms of feasibility analysis. Version 1.1 of the specification is likely to introduce a blocking time term into the release parameter's class.

- The goal for asynchronous event handlers is to have a lightweight concurrency mechanism. Some implementations will, however, simply map an event handler to a real-time thread and the original motivations for event handlers will be lost. It is a major challenge to provide effective implementations that can cope with heap-using and no-heap handlers, blocking and nonblocking handlers, daemon and nondaemon handlers, multiple schedulers, cost enforcement, and deadline monitoring (Dibble, 2002; Wellings and Burns, 2002).

12.4 Resource Sharing and Synchronization

12.4.1 Rationale

Although the RTSJ allows an implementation to support many different approaches to scheduling, the main focus is on priority-based scheduling. Over the last 25 years, much research has been undertaken

on the analysis of timing properties in priority-based concurrent systems. One key aspect of this involves understanding the impact of communication and synchronization between different-priority schedulable objects. In Java, communication and synchronization is based on mutually exclusive access to shared data via a monitor-like construct. Unfortunately, all synchronization mechanisms that are based on mutual exclusion suffer from priority inversion. There are two solutions to the priority inversion problem. The first is to use a priority inheritance algorithm and the second is to use nonblocking communication mechanisms. The RTSJ uses the former to limit the blocking between communicating schedulable objects, and the latter to facilitate communication between heap-using and nonheap-using threads/schedulable objects.

12.4.2 Approach

Priority inversion can occur whenever a schedulable object is blocked waiting for a resource. To limit the length of time of that blocking, the RTSJ requires the following:

- All queues maintained by the real-time virtual machine must be priority ordered. So, for example, the queue of schedulable objects waiting for an object lock (as a result of a synchronized method call or the execution of a synchronized statement) must be priority ordered. Where there is more than one schedulable object in the queue at the same priority, the order between them is not defined (although it is usual for this to be first-in-first-out (FIFO)). Similarly, the queues resulting from calls to the wait methods in the `Object` class should be priority ordered.
- Facilities for the programmer to specify the use of different-priority inversion control algorithms. By default, the RTSJ requires priority inheritance to occur whenever a schedulable object is blocked waiting for a resource (e.g., an object lock).

The programmer can change the default priority inversion control algorithm for individual objects (or for all objects) via the `MonitorControl` class hierarchy. At the root of this hierarchy is the following abstract class:

```
package javax.realtime;
public abstract class MonitorControl {
    // constructors
    protected MonitorControl();
    // methods
    public static MonitorControl getMonitorControl();
    public static MonitorControl getMonitorControl(Object monitor);
    public static MonitorControl setMonitorControl(MonitorControl policy);
    public static MonitorControl setMonitorControl(Object monitor,
                                                    MonitorControl policy);
};
```

The four static methods allow the getting/setting of the default policy and the getting/setting for an individual object (the methods return the old policy). The two RTSJ-defined policies are represented by subclasses. The default policy can be specified by passing an instance of the `PriorityInheritance` class:

```
package javax.realtime;
public class PriorityInheritance extends MonitorControl {
    // methods
    public static PriorityInheritance instance();
    // The instance is in immortal memory.
}
```

or an instance of the `PriorityCeilingEmulation` class:

```
package javax.realtime;
public class PriorityCeilingEmulation extends MonitorControl {
    // methods
    public int getCeiling(); // Returns the current priority.
    public static int getMaxCeiling();
    public static PriorityCeilingEmulation instance(int ceiling);
}
```

12.4.2.1 Wait-Free Communication

The alternative to blocking communication based on synchronized objects is to use nonblocking (wait-free) communication. The RTSJ provides two main classes that facilitate this form of communication. All are based on a queue.

`WaitFreeWriteQueue`—this class is intended for situations where a no-heap real-time thread wishes to send data to one or more heap-using threads. However, it can be used by any thread. Essentially, the writer thread is never blocked when writing even when the queue is full (the write fails). Readers can be blocked when the queue is empty. The class assumes that multiple writers provide their own synchronization. If the queue is only accessed by a single reader and a single writer, that information can be provided by one of the constructors. This allows an implementation to optimize the access protocol.

`WaitFreeReadQueue`—this class is intended for situations where a no-heap real-time thread wishes to receive data from one or more heap-using threads. However, it can be used by any thread. Essentially, the reader thread is never blocked when reading even when the queue is empty (the read fails). Writers can be blocked when the queue is full. The class assumes that multiple readers provide their own synchronization. Optimizations are again possible to support the single reader/writer case.

12.4.3 Outstanding Issues and Future Directions

Priority inheritance algorithms are reasonably well understood so the RTSJ's support in this area is unlikely to change much. However, running multiple priority inheritance algorithms in the same system is unusual. The interactions between priority ceiling emulation and simple priority inheritance was underspecified in version 1.0 of the specification. Version 1.0.1 has done much to improve the situation and the details of dynamic changing between inheritance approaches are catered for.

If the RTSJ does decide to provide better support for alternative and multiple schedulers then it is likely that this will include the generalization of priority ceiling protocols defined by Baker's preemption-level protocol (Baker, 1991). A possible approach is given by Zerzelidis and Wellings (2006).

Another area that also needs further attention is the support for communication between heap-using and no-heap real-time schedulable objects. The current mechanisms supported by wait-free queues are currently underspecified and need reevaluation in the light of user experience.

12.5 Time Values and Clocks

12.5.1 Rationale

Java supports the notion of a wall clock (calendar time). The `Date` class is intended to reflect UTC (Coordinated Universal Time); however, accuracy depends on the host system (Gosling et al., 2000).

Moreover, real-time systems often require additional functionality. In particular, they may need access to the following (Object Management Group, 2002):

1. A monotonic clock that progresses at a constant rate and is not subject to the insertion of extra ticks to reflect leap seconds (as UTC clocks are). A constant rate is required for control algorithms that need to execute on a regular basis. Many monotonic clocks are also relative to system start-up and can be used only to measure the passage of time, not calendar time. A monotonic clock is also useful for delaying a thread for a period of time, for example, while it waits for a device to respond. In addition, it may be used to provide timeouts so that the nonoccurrence of some event (e.g., an interrupt) can be detected.
2. A countdown clock that can be paused, continued, or reset (e.g., the clock that counts down to the launch of a space shuttle).
3. A CPU execution time clock that measures the amount of CPU time that is being consumed by a particular thread or object.

All of the above clocks also need a resolution that is potentially finer than the millisecond level. They may all be based on the same underlying physical clock, or be supported by separate clocks. Where more than one clock is provided, the relationship between them may be important—in particular, whether their values can drift apart. Even with a single clock, drift in relation to the external time frame will be important.

The RTSJ supports a real-time clock and an implicit CPU execution time clock. Countdown clocks are supported via timers.

12.5.2 Approach

The RTSJ introduces clocks with high-resolution time types. The `HighResolutionTime` class encapsulates time values with nanosecond granularity and an associated clock. A value is represented by a 64-bit-ms and a 32-bit-ns component. The class is an abstract class, an abridged version is shown below.

```
package javax.realtime;
public abstract class HighResolutionTime
    implements Comparable, Cloneable {

    // methods
    public Object clone();
    public int compareTo(HighResolutionTime time);
    public int compareTo(Object object);
    public boolean equals(HighResolutionTime time);
    public boolean equals(Object object);

    public Clock getClock();
    public final long getMilliseconds();
    public final int getNanoseconds();
    public void set(long millis);
    public void set(long millis, int nanos);

    public static void waitForObject(
        Object target, HighResolutionTime time)
        throws InterruptedException;
}
```

As with all classes that are essentially abstract data types, there are methods to read, write, compare, and clone time values, all of which are self-explanatory. Some methods, however, do need further consideration. The method `compareTo`, which takes an arbitrary object, is provided so that the class can implement the `java.lang.Comparable` interface. If an object passed to this method is not a `HighResolutionTime` type, the method will throw `ClassCastException`. In contrast, the `equals` method that overrides its counterpart in the `Object` class will return `-1`. Note, that

two high-resolution time objects must have the same clock if they are to be compared (otherwise, an `IllegalArgumentException` is thrown). Finally, the `waitForObject` static method is introduced to enable a schedulable object to have a high-resolution timeout while waiting for a condition to be signaled inside a monitor. This is needed because the `wait` method in the `Object` class is defined to not be overridden (i.e., it is “final”). Of course, the calling schedulable object must hold the lock on the target object when this method is called.

The `HighResolutionTime` class has three subclasses: one that represents relative time, one that represents absolute time, and one that represents “rational” time. Relative time is a duration measured by a particular clock. Absolute time is actually expressed as a time relative to some epoch. This epoch depends on the associated clock. It might be January 1, 1970, GMT for the wall clock or system start-up time for a monotonic clock. Rational time is a relative-time type, which has an associated frequency. It is used to represent the rate at which certain events occur (e.g., periodic thread execution).^{*} The RTSJ clock class defines the abstract class from which all clocks are derived.

```
package javax.realtime;
public abstract class Clock {

    // constructor
    public Clock();

    // methods
    public static Clock getRealtimeClock();
    public abstract RelativeTime getResolution();
    public abstract AbsoluteTime getTime();
    public abstract AbsoluteTime getTime(AbsoluteTime time);
    public abstract void setResolution(RelativeTime resolution);
    public abstract RelativeTime getEpochOffset();
}
```

The specification allows many different types of clocks; for example, there could be a CPU execution-time clock (although this is not required by the RTSJ). There is always one real-time clock that advances monotonically. A static method `getRealtimeClock` allows this clock to be obtained.

12.5.3 Outstanding Issues and Future Directions

The 1.0.1 version of the RTSJ clarifies the treatment of multiple clocks under the RTSJ, but it does not specify the behavior of any clock other than the default real-time clock. Future development is likely to add other clocks—for example, currently the RTSJ allows the execution time of a schedulable object to be monitored but does not expose the underlying clock mechanism for this. It is an anomaly that the RTSJ programmer can request that a cost overrun handler can be released when a schedulable object consumes more CPU time than in its stated budget; yet the programmer cannot determine how much CPU time the schedulable object has currently consumed. An explicit consumed CPU time clock would also augment the processing group facilities of the RTSJ and allow more flexible execution-time servers to be implemented, including deferrable and sporadic servers.

12.6 Memory Management

12.6.1 Rationale

Many real-time systems (particularly those that form part of an embedded system) have only a limited amount of memory available; this is either because of the cost or because of other constraints associated

^{*}This class has now been deprecated.

with the surrounding system (e.g., size, power, or weight constraints). It may, therefore, be necessary to control how this memory is allocated so that it can be used effectively. Furthermore, where there is more than one type of memory (with different access characteristics) within the system, it may be necessary to instruct the compiler to place certain data types at certain locations. By doing this, the program is able to increase performance and predictability as well as interact more effectively with the outside world.

The runtime implementations of most programming languages provide a large amount of memory (called the heap) so that the programmer can make dynamic requests for chunks to be allocated (e.g., to contain an array whose bounds are not known at compile time). An allocator (the **new** operator in Java) is used for this purpose. It returns a reference to the memory within the heap of adequate size for the program's data structures (classes). The runtime system (JVM) is responsible for managing the heap. Key problems are deciding how much space is required and deciding when allocated space can be released and reused. The first of these problems requires application knowledge (the `SizeEstimator` class in RTSJ helps here). The second can be handled in several ways, requiring

- The programmer to return the memory explicitly (e.g., via the **free** function in C)—this is error prone but is easy to implement
- The runtime support system (JVM) to monitor the memory and determine when it can logically no longer be accessed—the scope rules of a language allows its implementation to adopt this approach; when a reference type goes out of scope, all the memory associated with that reference type can be freed (e.g., Ada's **access** types)
- The runtime support system (JVM) to monitor the memory and release chunks that are no longer being used (garbage collection)—this is, perhaps, the most general approach as it allows memory to be freed even though its associated reference type is still in scope. Java, of course, supports this approach

From a real-time perspective, the above approaches have an increasing impact on the ability to analyze the timing properties of the program. In particular, garbage collection may be performed either when the heap is full (there is no free space left) or incrementally (either by an asynchronous activity or on each allocation request). In either case, running the garbage collector may have a significant impact on the response time of a time-critical thread.

All objects in standard Java are allocated on the heap, and the language requires garbage collection for an effective implementation. The garbage collector runs as part of the JVM. In general, garbage collectors for real-time systems are incremental and can be classified as work-based or time-based (Bacon et al., 2003). In a work-based collector, every application request (to allocate an object or assign a reference) performs a small amount of garbage collection work. The amount is determined by the required allocation rate. Here, the worst-case execution time of each application memory request includes a component for garbage collection. With a time-based collector, a real-time thread is scheduled to run at a given priority and with a given CPU budget in a given period. It performs as much garbage collection as it can and competes with other real-time threads for the processor. A feasibility analysis performed by the on- or offline scheduler must take into account this collector real-time thread. It is beyond the scope of this chapter to discuss individual strategies for real-time garbage collection. See Bacon et al. (2003), Chang and Wellings (2005), Henriksson (1998), Kim et al. (1999), and Siebert (1999, 2004) for possible approaches. In spite of the progress made, there is still some reluctance to rely on these techniques in time-critical systems (particularly high-integrity ones).

12.6.2 Approach

The RTSJ recognizes that it is necessary to allow memory management, which is not affected by the vagaries of garbage collection. To this end, it introduces the notion of memory areas, some of which exist outside the traditional Java heap and never suffer garbage collection. It also requires that the garbage collector can be preempted by real-time threads and that the time between a real-time thread wishing to preempt and

the time it is allowed to preempt is bounded (there should be a bounded latency for preemption to take place). The `MemoryArea` class is an abstract class from which all RTSJ memory areas are derived, an abridged version of which is shown below.

```
package javax.realtime;
public abstract class MemoryArea {

    // constructors
    protected MemoryArea(long size); // In bytes.
    protected MemoryArea(long size, Runnable logic);
    ...

    // methods
    public void enter();
        // Associate this memory area to the current schedulable
        // object for the duration of the logic.run method
        // passed as a parameter to the constructor.

    public void enter(Runnable logic);
        // Associate this memory area to the current
        // schedulable object for the duration of the
        // logic.run method passed as a parameter.

    public void executeInArea(Runnable logic);
        // Execute the logic.run method in the context of an
        // active memory area.

    public static MemoryArea getMemoryArea(Object object);
        // Get the memory area associated with the object.

    public long memoryConsumed();
        // Get the number of bytes consumed in this area.

    public long memoryRemaining();
        // Get the number of bytes remaining in this area.

    public long size();
        // Get the current size of this area.
    ...
}
```

When a particular memory area is entered, all object allocation is performed within that area (this is known as the *current allocation context*). Using this abstract class, the RTSJ defines various kinds of memory including the following:

- **HeapMemory**—Heap memory allows objects to be allocated in the standard Java heap.
- **ImmortalMemory**—Immortal memory is shared among all threads in an application. Objects created in immortal memory are never subject to garbage collection delays and behave as if they are freed by the system only when the program terminates.*
- **ScopedMemory**—Scoped memory is a memory area where objects with a well-defined lifetime can be allocated. A scoped memory may be entered explicitly or implicitly by attaching it to a real-time entity (a real-time thread or an asynchronous event handler—generically called a *schedulable object* by the RTSJ, see Section 12.3) at its creation time. Associated with each scoped memory is a

*As an optimization, the RTSJ allows objects to be collected in immortal memory as long as the collection does not cause delays to the no-heap real-time activities using the memory.

reference count. This keeps track of how many real-time entities are currently using the area. When the reference count reaches 0, all objects resident in the scoped memory have their finalization method executed, and the memory is available for reuse. The `ScopedMemory` class is an abstract class that has two subclasses:

- `VTMemory`—A subclass of `ScopedMemory` where allocations may take variable amounts of time.
- `LTMemory`—A subclass of `ScopedMemory` where allocations occur in linear time (i.e., the time taken to allocate the object is directly proportional to the size of the object).

Memory parameters can be given when real-time threads and asynchronous event handlers are created. They can also be set or changed while the real-time threads and asynchronous event handlers are active. Memory parameters specify

- The maximum amount of memory a thread/handler can consume in its default memory area
- The maximum amount of memory that can be consumed in immortal memory, and
- A limit on the rate of allocation from the heap (in bytes per second), which can be used by the scheduler as part of an admission control policy or for the purpose of ensuring adequate real-time garbage collection

Multiple memory areas can be entered via the use of nested calls to the `enter` method, and the current allocation context can be changed between active memory areas via calls to the `executeInArea` method.

12.6.3 Outstanding Issues and Future Directions

The RTSJ has established a new memory management model (via the introduction of memory areas) to facilitate more predictable memory allocation and deallocation. This is, perhaps, one of the most controversial areas of the RTSJ for two main reasons.

1. Using the model is far from straightforward and requires a programming paradigm shift.

What was simple in standard Java (where the programmer did not have to be concerned with memory allocation and deallocation strategies) has become more complex for the RTSJ programmer. It can be argued (Borg and Wellings, 2006) that this is inevitable, as efficient predictable real-time memory management requires the programmer to supply more information on how long objects live. As more experience with using scoped memory is acquired, software design patterns will be developed to ease the programming burden (these are already beginning to emerge, e.g., see Benowitz and Niessner [2003], Pizlo et al. [2004], and Raman et al. [2005]).

Scoped memory works best when an application wants space to store some temporary objects. As long as these objects can be grouped together so that they follow a stacking algorithm (last-in-first-out—or in this case, last-created-first-destroyed), the model works well. However, many algorithms require a first-in-first-out or some other order. For example, consider a simple `List` (as found in `java.util`). Now suppose objects are created and placed in the list. At a later time, they are removed from the list (say one at a time and in the order in which they were placed) and processed. Once processed, they are no longer required. Programming this using scoped memory areas (so that objects are automatically reclaimed) is far from straightforward and requires detailed knowledge of both the list structure and the lifetime of the object when it is added to the list. If this is available, then a combination of scoped memory areas and the use of the `enter` and `executeInArea` methods can usually achieve the desired result (Borg, 2004, 2006). However, the algorithms are difficult and not very elegant. Again, this is an area where design patterns will, in time, emerge. Furthermore, just as with Java, it has been found necessary to allow interactions between the program and the garbage collector (via weak references), so it is also necessary to allow

interactions between the program and the implementation of scoped memory areas (Borg and Wellings, 2003).

2. Support for the memory model incurs runtime overheads.

The deallocation strategy of each type of memory area is different and, to maintain referential integrity, there are assignment rules that must be obeyed by the programmer and that, consequently, must be checked by the implementation. Furthermore, the ability to nest memory areas means that illegal nesting must be prohibited. While it is clear that static analysis of programs can eliminate many of the runtime checks (Beebe and Rinard, 2001; Salcianu and Rinard, 2001), this requires special RTSJ-aware compilers or tools. The availability of these, along with efficient runtime implementations of the overall model, may well determine the eventual impact and take up of the RTSJ.*

Another concern with the RTSJ's memory model is the amount of information that must be placed in immortal memory. Currently, the specification requires all static variables, class objects, and interned strings be placed in a memory area that can be referenced without generating exceptions—this means heap or immortal memory. To avoid garbage collection delays when accessed by hard real-time activities, they must be placed in immortal memory. Furthermore, all static initialization must occur in the context of `ImmutableMemory` and no mention is made of the interaction between Java's class-loading architecture and the RTSJ. This is an area that is likely to receive further attention in the next version of the specification.

12.7 Conclusions

Although there is little doubt that the Java language has been immensely successful in a wide range of application areas, it has yet to fully establish itself completely in the real-time and embedded markets. These systems are mainly small (e.g., mobile phones) but can sometimes be extremely large (e.g., air traffic control systems). For small embedded applications, sequential languages like C and C++ reign supreme. For the larger real-time high-integrity systems, Ada is still important (see the next chapter). The introduction of a Real-Time Specification for Java could dramatically alter the status quo. The provision of real-time programming abstractions along with the promise of a predictable real-time Java Virtual Machine has the potential to add fuel to the already raging Java fire.

Initially (version 1.0), the RTSJ was underspecified; however, the current version (1.0.1) is a much tighter specification, and work is now progressing to extend the standard and produce high-integrity profiles. Implementation are emerging (both interpreted and compiled, propriety and open source) and beginning to mature.

One of the initial goals of the RTSJ was to support state of the practice in real-time systems development and mechanisms to allow advances in state of the art. As a result of this, the specification provides several frameworks that are only partially instantiated. The scheduler framework is one (the others being the feasibility analysis framework, the clock framework, and the physical memory framework). In the medium term, these frameworks are likely to be extended to become more usable in a standard and portable way.

Acknowledgment

The authors gratefully acknowledge the contributions made by Andrew Borg, Hao Cai, Yang Chang, Osmar Marchi dos Santos, and Alex Zerzelidis to some of the ideas presented in this chapter.

*As an example, the Jamaica VM supports compiled Java along with static analysis tools that can remove many of the runtime checks.

References

- M. Aldea Rivas and M. Gonzlez Harbour. Posix-compatible application-defined scheduling in marte os. In *Proceedings of 14th Euromicro Conference on Real-Time Systems*, pp. 67–75, 2002.
- G. Back, P. Tullmann, L. Stoller, W.C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation, uucs-98-015. Technical report, Department of Computer Science, University of Utah, 1998.
- D.F. Bacon, P. Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 285–298, 2003.
- T.P. Baker. Stack-based scheduling of realtime processes. *Real Time Systems*, 3(1): 67–91, 1991.
- W.S. Beebe and M.C. Rinard. An implementation of scoped memory for real-time Java. In *EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*, pp. 289–305. Springer, London, 2001. Springer.
- E. Benowitz and A. Niessner. A pattern catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pp. 497–507, Springer, Berlin, 2003.
- G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- A. Borg. On the development of dynamic real-time applications in the RTSJ—A model for expressing dynamic memory requirements. Technical report YCS 379, Department of Computer Science, University of York, UK, 2004.
- A. Borg. *Coarse Grained Memory Management in Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, UK, 2006.
- A. Borg and A.J. Wellings. Reference objects for RTSJ scoped memory areas. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pp. 397–410, Springer, Berlin, 2003.
- A. Borg and A.J. Wellings. Memory management: Life and times. In *Proceedings of 18th Euromicro Conference on Real-Time Systems*, pp. 237–250, 2006.
- B.M. Brosgol, R.J. Hassan, and S. Robbins. Asynchronous transfer of control in the real-time specification for Java. In *Proceedings of 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 101–108, 2002.
- L. Carnahan and M. Ruark. Requirements for real-time extensions for the Java platform. NIST, <http://www.nist.gov/rt-java>, US National Institute of Standards and Technology, 1999.
- Y. Chang and A.J. Wellings. Integrating hybrid garbage collection with dual priority scheduling. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 185–188. IEEE, IEEE Computer Society Press, August 2005.
- P. Dibble, R. Belliardi, B. Brosgol, D. Holmes, and A.J. Wellings. *The Real-Time Specification for Java: Second Edition*. Addison-Wesley, 2006.
- P.C. Dibble. *Real-Time Java Platform Programming*. Sun Microsystems Press, 2002.
- J. Gosling, B. Joy, G. Steele, and B. Bracha. *The Java Language Specification: 2nd Edition*. Addison-Wesley, 2000.
- R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund University, Sweden, 1998.
- G. Hilderink. A new Java thread model for concurrent programming of real-time systems. *Real-Time Magazine*, 1: 30–35, 1998.
- J Consortium. Real-time core extensions for the Java platform. Revision 1.0.10, J Consortium, 2000.
- T. Kim, N. Chang, N. Kim, and H. Shin. Scheduling garbage collector for embedded real-time systems. In *Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, ACM SIGPLAN Notices, pp. 55–64, 1999.
- O. Marchi dos Santos and A.J. Wellings. Cost monitoring and enforcement in the Real-Time Specification for Java—a formal evaluation. In *Proceedings of the 26th Real-Time Systems Symposium*, pp. 177–186. IEEE Computer Society Press, 2005.

- H. McGhan and M. O'Connor. picoJava: A direct execution engine for Java bytecode. *IEEE Computer*, 31(10): 22–30, 1998.
- A. Miyoshi, H. Tokuda, and T. Kitayama. Implementation and evaluation of real-time Java threads. In *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166–175, 1997.
- K. Nilsen. Adding real-time capabilities to the Java programming language. *Communications of the ACM*, pp. 44–57, 1998.
- Object Management Group. Real-time Corba 2.0. Technical report OMG Document <http://www.omg.org/docs/formal/03-11-01>, OMG, 2003.
- Object Management Group. Enhanced view of time specification enhanced view of time specification, v1.1. Technical report, Document <http://www.omg.org/docs/formal/02-05-07>, OMG, 2002.
- J. Ousterhout. Why threads are a bad idea (for most purposes). <http://www.home.pacbell.net/ouster/threads.ppt>, Sun Microsystems Laboratories, 2002.
- F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE Symposium on Object-Oriented Real-Time Distributed Computing, ISORC*, pp. 101–110, 2004.
- K. Raman, Y. Zhang, M. Panahi, J.A. Colmenares, and R. Klefsstad. Patterns and tools for achieving predictability and performance with real-time java. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pp. 247–253. IEEE, IEEE Computer Society Press, 2005.
- A. Salcianu and M.C. Rinard. Pointer and escape analysis for multithreaded programs. In *ACM PPOPP*, pp. 12–23, 2001.
- F. Siebert. Real-time garbage collection in multi-threaded systems on a single microprocessor. In *IEEE Real-Time Systems Symposium*, pp. 277–278, 1999.
- F. Siebert. The impact of real-time garbage collection on realtime Java programming. In *Proceedings of the 7th International Symposium on Object-Oriented Real-time Distributed Computing ISORC 2004*, pp. 33–44, 2004.
- M. Tofte, L. Birkedal, M. Elsmann, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3): 245–265, 2004. ISSN 1388-3690.
- M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2): 109–176, 1997.
- R. van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of the 8th ACM SIGOPS European Workshop*, also available at <http://www.cs.cornell.edu/Info/People/rvr/papers/event/event.ps>, 1998.
- A.J. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.
- A.J. Wellings, G. Bollella, P. Dibble, and D. Holmes. Cost enforcement and deadline monitoring in the Real-Time Specification for Java. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2004*, pp. 78–85. Computer Society, IEEE, May 2003.
- A.J. Wellings and A. Burns. Asynchronous event handling and real-time threads in the Real-Time Specification for Java. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 81–89, 2002.
- A. Zerkelidis and A.J. Wellings. Getting more flexible scheduling in the RTSJ. In *Proceedings of the 9th International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2006*, pp. 3–10, 2006.

13

Programming Execution-Time Servers and Supporting EDF Scheduling in Ada 2005

13.1	Introduction	13-1
13.2	The Ada 95 Version of the Language	13-3
13.3	New Ada 2005 Features	13-3
	Timing Events • Execution-Time Clocks • Execution-Time Timers • Group Budgets	
13.4	Programming Execution-Time Servers	13-8
	Deferrable Server • Sporadic Server	
13.5	Support for Deadlines	13-14
13.6	Baker's Preemption Level Protocol for Protected Objects	13-15
13.7	Supporting EDF Scheduling in Ada	13-15
	The Rules of EDF Dispatching • An Important Constraint • Example Task Set	
13.8	Mixed Dispatching Systems	13-19
13.9	Conclusion	13-20
13.10	Postscript—Ada and Java	13-20

Alan Burns

University of York

Andy Wellings

University of York

13.1 Introduction

Over the last two decades or more, a considerable volume of literature has been produced that addresses the issues and challenges of scheduling real-time systems. The focus of much of this research has been on how to effectively support a collection of distinct applications, each with a mixture of periodic and nonperiodic, and hard and soft, activities. This work is seen as an enabling technology for a wide range of applications from multimedia to robust control. Notwithstanding the quality of individual contributions, it is unfortunately true that system implementors have not, in general, taken up these results. There are a

number of reasons for this, including the normal inertia associated with technical change, but we would highlight the following [18]:

- Inconsistency in the literature—no single approach has emerged as being widely applicable, indeed there is not even any consensus over the right simulation models to use for evaluation.
- Limitations of the scheduling models and analysis—too many models have unrealistic assumptions such as ignoring runtime overheads or assuming that the worst-case execution time (WCET) is known for all tasks (or not recognizing that actual execution times can be much smaller than WCET values).
- Difficulty of implementation—often the scheduling scheme will require operating system primitives that are not available on any commercial platform.
- Lack of computational model—the scheduling results are not tied back to realistic models that applications can use to construct systems.
- Lack of design patterns that applications programmers can adopt and adapt.

The work presented in this chapter focuses on the final three problems of this group.

Over the last few years, the appropriate abstractions that are needed to support the development of flexible real-time systems have been the subject of much investigation [9,11,12,15,16,24]. Working in conjunction with the Series of International Workshop on Real-Time Ada Issues—IRTAW,* a set of mechanisms has been defined that allow a wide range of flexible real-time systems to be implemented. These have now been incorporated into the Ada 2005 standard [7]. In this chapter, we will show how these facilities allow implementations to be programmed and analyzed, and code patterns developed.

The challenges of real-time scheduling are easy to (informally) define: hard deadlines must always be met, distinct applications (or subsystems of the same application) must be isolated from each other, but any spare capacity (typically CPU resource) must be used to maximize the overall utility of the applications. The “extra” work that could be undertaken typically exceeds the spare capacity available, and hence the scheduler must decide which nonhard task to execute at any time. As processors become more complex and the analysis of WCET more pessimistic, the amount of spare capacity will increase. Indeed, the total utilization of the hard tasks will often be much less than 50%. There is continuing pressure on applications to minimize their use of hard tasks and exploit flexible scheduling to deliver dynamic behavior. A real-time system with this mixture of hard and soft (usually aperiodic) tasks has to ensure that

- All hard real-time tasks meet their deadlines even when worst-case conditions are being experienced
- Failed components do not impact on nonfailed applications by, for example, denial of service
- All aperiodic tasks have a good response time—here, good should mean that the tasks meet their soft deadlines in most of their invocations
- Other soft tasks exhibit good quality of output by gaining access to the maximum computation time available before their soft/firm deadline

One of the key building blocks for delivering this form of flexible scheduling is the use of *execution-time servers*[†] [4,6,19,20,22]. A server is in some senses a virtual processor, it provides its clients with a budget that has a defined “shelf life” and a means of replenishing the budget in a predictable and analyzable way. The servers collectively must sustain their budgets (i.e., allow the full budget to be available to clients), while the clients must ensure that the budgets are sufficient. A number of papers have addressed the scheduling issues associated with server-based systems. In this chapter, we are concerned with programming servers—or more precisely defining the language primitives that allow server abstractions to be built.

Another important building block for flexible scheduling is control over dispatching. Some tasks need the certainty provided by fixed priority-based scheduling, others need to maximize resource usage using efficient schemes such as earliest deadline first (EDF). In this chapter, we therefore also illustrate how

*Proceeding published in Ada Letters: Vol XXI, March 2001; Vol XXII, December 2002; and Vol XXIII, December 2003.

[†]In this chapter, the term *server* is used to imply an execution-time server.

Ada 2005 supports both fixed priority and EDF dispatching, and the combined use of these and other policies.

The rest of the chapter is structured as follows. The Ada 95 version of the language is introduced in the next section. New features of Ada 2005 are then introduced in Section 13.3, and examples of its use to program server abstractions given in Section 13.4. Ada 2005 support for programming with deadlines is given in Section 13.5, followed by a discussion of Baker's algorithm for preemption-level control over protected objects (in Section 13.6). Support for EDF scheduling is discussed in Section 13.7 and mixed dispatching in Section 13.8. The last section presents conclusions.

13.2 The Ada 95 Version of the Language

The Ada 95 version of the language already defines a number of expressive concurrency features that are effective in the real-time domain. Features include

- Tasks (the basic unit of concurrency)—dynamic or static creation, and flat or hieratical relationships between tasks are supported
- Absolute and relative delay statements
- A calendar (wall) clock and a “real-time” monotonic clock
- Rendezvous—a synchronous (control-oriented) means of communication between tasks
- Asynchronous transfer of control (ATC)—an asynchronous means of affecting the behavior of other tasks
- Protected types—a monitor-like object that enforces mutual exclusion over its operations, and that supports condition synchronization via a form of guarded command
- Requeue—a synchronization primitive that allows a guarded command to be prematurely terminated with the calling task placed on another guarded operation; this significantly extends the expressive power of the synchronization primitive without jeopardizing efficiency
- Exceptions—a means of prematurely terminating the execution of a sequential program segment
- Controlled objects—an OOP feature that allows the execution of code when objects enter and exit the scope of their declaration
- Fixed priority dispatching—a standard implementation, including ceiling priority protection against unbounded priority inversion
- Dynamic task priorities—a means of programming modifiable scheduling policies by changing task priorities at runtime
- Interrupt handlers—mapped on to procedures declared in protected types

In comparison with Real-Time Specification for Java (RTSJ) (see Chapter 12), Ada 95 supported the primitives from which abstractions such as periodic tasks could be programmed, rather than give such abstractions direct support.

What is missing from the Ada 95 model is explicit support for resource management and for other dispatching policies such as EDF and Round Robin.

In this chapter, we concentrate on the features that have been included into Ada 2005 for resource control and EDF dispatching. However, it should also be noted that Ada 2005 also supports the Ravenscar Profile (a profile of the tasking model) for high-integrity real-time applications; dynamic priority ceilings for protected objects; and the use of task, protected and synchronized interfaces (which integrates Ada's OOP model with its tasking model).

13.3 New Ada 2005 Features

Ada 2005 [7] is the recent amendment to the Ada 95 standard. It includes many additional features across a range of programming topics. Of particular interest here are the new features that have been added

to support the programming of real-time systems with requirements that include high performance, flexibility, and runtime protection.

In the following subsection, we introduce four important new features of Ada—timing events, execution time clocks, timers, and group budgets. Other features of Ada 2005 are described later in this chapter. Further details on all the real-time features of Ada 2005 can be found in Ref. 14. An introduction to Ada 2005, in general, is provided by Barnes [3].

13.3.1 Timing Events

Ada 2005 has introduced a new abstraction of a timing event to allow code to be executed at specified times without the need to employ a task/thread. These events are like interrupts, but they are generated by the progression of the real-time system clock. Associated with a timing event is a handler that is executed at the allotted time. An implementation should actually execute this handler directly from the interrupt handler for the clock device. This leads to a very efficient scheme. The following standard library package is defined by Ada 2005:

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is access protected
    procedure(Event : in out Timing_Event);
  procedure Set_Handler(Event : in out Timing_Event;
    At_Time : Time; Handler: Timing_Event_Handler);
  procedure Set_Handler(Event : in out Timing_Event; In_Time: Time_Span;
    Handler: Timing_Event_Handler);
  function Is_Handler_Set(Event : Timing_Event) return Boolean;
  function Current_Handler(Event : Timing_Event) return Timing_Event_Handler;
  procedure Cancel_Handler(Event : in out Timing_Event; Cancelled : out Boolean);
  function Time_Of_Event(Event : Timing_Event) return Time;
private
  -- Not specified by the language.
end Ada.Real_Time.Timing_Events;
```

The handler to be executed is a protected procedure that is passed with the timing event as a parameter when the handler is set. When the associated time is reached, the handler is executed. The timing event is a tagged type and hence can be extended by the application (an example of this is given later). The term “protected” implies it can be executed safely in a concurrent program (by the use of a ceiling protocol). Any implementation of this package must document the accuracy and granularity of mechanisms used to fire these timing events.

Many of the new features introduced into Ada 2005 take the form of event handling with a structure similar to the above.

Most server abstractions require budgets to be replenished at fixed points in time—timing events will be used to efficiently implement this requirement.

13.3.2 Execution-Time Clocks

In hard real-time systems, it is essential to monitor the execution times of all tasks and detect situations in which the estimated WCET is exceeded. This detection was usually available in systems scheduled with cyclic executives, because the periodic nature of its cycle makes it easy to check that all initiated work had been completed by the end of each cycle. In event-driven concurrent systems, the same capability should be available, and this can be accomplished with execution-time clocks and timers. In addition, many flexible real-time scheduling algorithms require the capability to measure execution time and be able to perform scheduling actions when a certain amount of execution time has been consumed.

Ada 2005 directly supports execution-time clocks for tasks, including timers that can be fired when tasks have used defined amounts of execution times:

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is
  type CPU_Time is private;
  CPU_Time_First : constant CPU_Time;
  CPU_Time_Last  : constant CPU_Time;
  CPU_Time_Unit  : constant := <implementation-defined-real-number>;
  CPU_Tick : constant Time_Span;
  function Clock(T : Ada.Task_Identification.Task_ID
    := Ada.Task_Identification.Current_Task) return CPU_Time;
  function "+" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
  function "+" (Left : Time_Span; Right : CPU_Time) return CPU_Time;
  function "-" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
  function "-" (Left : CPU_Time; Right : CPU_Time) return Time_Span;
  function "<" (Left, Right : CPU_Time) return Boolean;
  -- similar definitions for <=, > and >=
  procedure Split(T : CPU_Time; SC : out Seconds_Count; TS : out Time_Span);
  function Time_Of (SC : Seconds_Count; TS : Time_Span) return CPU_Time;
private
  -- Not specified by the language.
end Ada.Execution_Time;
```

The execution time of a task, or CPU time as it is commonly called, is the time spent by the system executing the task and services on its behalf. The clock is set to zero when the task is created, and it then monotonically increases as the task executes. The accuracy of the measurement of execution time cannot be dictated by the language specification, it is heavily dependent on the runtime support software. On some implementation (perhaps on non real-time operating systems), it may not even be possible to support this package. But if the package is supported, the range of CPU_Time must be at least 50 years and the granularity of the clock (CPU_Tick) should be no greater than 1 ms.

The following code shows how a periodic task could be constructed. On each iteration, the execution time of the task is output.

```
task body Periodic_Task is
  Interval : Time_Span := Milliseconds(30);
  -- defines the period of the task, 30ms in this example
  Next : Ada.Real_Time.Time;
  CPU,CPU2 : Ada.Execution_Time.CPU_Time;
  Used : Ada.Real_Time.Time_Span;
begin
  Next := Ada.Real_Time.Clock;
  CPU := Ada.Execution_Time.Clock;
  loop
    CPU2 := Ada.Execution_Time.Clock;
    -- code of the application
    Used := CPU2-CPU;
    -- print out Used
    CPU := CPU2;
    Next := Next + Interval;
    delay until Next;
  end loop;
end Periodic_Task;
```

In this example, the CPU time is calculated and output, the next release time is set and the task suspends itself.

13.3.3 Execution-Time Timers

Besides monitoring a task's execution time, it is also useful to trigger an event if the clock gets to some specified value. Often this is an error condition, where the task has executed for longer than was anticipated. A child package of `Execution_Time` provides support for this type of event:

```
package Ada.Execution_Time.Timers is
  type Timer(T : not null access constant
    Ada.Task_Identification.Task_ID) is tagged limited private;
  type Timer_Handler is access protected procedure(TM : in out Timer);
  Min_Handler_Ceiling : constant System.Any_Priority := <Implementation Defined>;
  procedure Set_Handler(TM : in out Timer; In_Time : Time_Span; Handler : Timer_Handler);
  procedure Set_Handler(TM : in out Timer; At_Time : CPU_Time; Handler : Timer_Handler);
  procedure Cancel_Handler(TM : in out Timer);
  function Current_Handler(TM : Timer) return Timer_Handler;
  function Time_Remaining(TM : Timer) return Time_Span;
  Timer_Resource_Error : exception;
private
  -- Not specified by the language.
end Ada.Execution_Time.Timers;
```

An execution-time timer is a “one-shot” event. It identifies the code to be executed when the task CPU time clock reaches a specified value. Obviously, it cannot reach that value again, so the handler needs to be reset for any further requirement.

To illustrate its use, the earlier code fragment is changed to stop the task if it executes for more than its WCET:

```
protected WCET_Protect is
  entry Overrun;
  procedure Overrun_Handler(T : in out Timer);
private
  Barrier : Boolean := False;
end WCET_Protect;

protected body Overrun_Handler is ...
  -- simple structures to lower barrier on the entry if the procedure is executed
end Overrun_Handler;

task body Periodic_Task is
  Interval : Time_Span := Milliseconds(30);
  Next : Ada.Real_Time.Time;
  CPU : Ada.Execution_Time.CPU_Time;
  Me : access constant Task_Id := Current_Task'Access;
  WCET_Timer : Timer(Me);
  begin
    Next := Ada.Real_Time.Clock;
    CPU := Ada.Execution_Time.Clock;
    loop
      Set_Handler(WCET_Timer, WCET, WCET_Protect.Overrun_Handler'Access);
      select
        WCET_Protect.Overrun;
        -- action to deal with WCET overrun
      then abort
        -- application code
      end select;
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Periodic_Task;
```

Note, in the case of no overrun, the call to `Set_Handler` will cancel the previous setting and install a new one.

13.3.4 Group Budgets

The support for execution-time clocks allows the CPU resource usage of individual tasks to be monitored, and the use of timers allows certain control algorithms to be programmed but again for single tasks. There are, however, situations in which the resource usage of groups of tasks needs to be managed. Typically, this occurs when distinct subsystems are programmed together but need to be protected from one another—no failures in one subsystem should lead to failures in the others. So even if a high-priority task gets into an infinite loop, tasks in other subsystems should still meet their deadlines. This requirement is usually met by execution-time servers.

Ada 2005 does not directly support servers. But it does provide the primitives from which servers can be programmed. Group budgets allow difference servers to be implemented. Note, servers can be used with fixed priority or EDF scheduling.

A typical server has a budget and a replenishment period. At the start of each period, the available budget is restored to its maximum amount. Unused budget at this time is discarded. To program a server requires timing events to trigger replenishment and a means of grouping tasks together and allocating them an amount of CPU resource. A standard package (a child of `Ada.Execution_Time`) is defined to accomplish this:

```
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;
  type Group_Budget_Handler is access protected procedure(GB : in out Group_Budget);
  type Task_Array is array(Positive range <>) of Ada.Task_Identification.Task_ID;
  Min_Handler_Ceiling : constant System.Any_Priority := <Implementation Defined>;

  procedure Add_Task(GB: in out Group_Budget; T : Ada.Task_Identification.Task_ID);
  procedure Remove_Task(GB: in out Group_Budget; T : Ada.Task_Identification.Task_ID);
  function Is_Member(GB: Group_Budget; T : Ada.Task_Identification.Task_ID) return Boolean;
  function Is_A_Group_Member(T : Ada.Task_Identification.Task_ID) return Boolean;
  function Members(GB: Group_Budget) return Task_Array;

  procedure Replenish(GB: in out Group_Budget; To : Time_Span);
  procedure Add(GB: in out Group_Budget; Interval : Time_Span);
  function Budget_Has_Expired(GB: Group_Budget) return Boolean;
  function Budget_Remaining(GB: Group_Budget) return Time_Span;

  procedure Set_Handler(GB: in out Group_Budget; Handler : Group_Budget_Handler);
  function Current_Handler(GB: Group_Budget) return Group_Budget_Handler;
  procedure Cancel_Handler(GB: in out Group_Budget; Canceled : out Boolean);

  Group_Budget_Error : exception;
private
  -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

The type `Group_Budget` represents a CPU budget to be used by a group of tasks.

The budget decreases whenever a task from the associated set executes. The accuracy of this accounting is again implementation defined. To increase the amount of budget available, two routines are provided. The `Replenish` procedure sets the budget to the amount of “real-time” given by the `To` parameter. It replaces the current value of the budget. By comparison, the `Add` procedure increases the budget by the `Interval` amount. But as this parameter can be negative, it can also be used to, in effect, reduce the budget.

The minimal budget that can be held in a `Group_Budget` is zero—represented by `Time_Span_Zero`. If a budget is exhausted, or if `Add` is used to reduce the budget by an amount greater than its current value, then the lowest budget value we can get is zero. To inquire about the state of the budget, two functions are provided. Note that when `Budget_Has_Expired` returns `True` then `Budget_Remaining` will return `Time_Span_Zero` (as long as there are no replenishments between the two calls).

A handler is associated with a group budget by use of the `Set_Handler` procedure. There is an implicit event associated with a `Group_Budget` that occurs whenever the budget goes to zero. If at that time there is a nonnull handler set for the budget, the handler will be executed.* As with timers, there is a need to define the minimum ceiling priority level for the protected object linked to any group budget handler. Also note there are `Current_Handler` and `Cancel_Handler` subprograms defined. Furthermore, the handler is permanently linked to the budget (unless it is changed or canceled). So every time the budget goes to zero, the handler is fired.

We will shortly give some examples to illustrate the use of group budgets. But it is important that one aspect of the provision is clear.

- When the budget becomes zero, the associated tasks *continue* to execute.

If action should be taken when there is no budget, this has to be programmed (it must be instigated by the handler). So group budgets are not in themselves a server abstraction—but they allow these abstractions to be constructed.

13.4 Programming Execution-Time Servers

As discussed in Section 13.1, servers are a means of controlling the allocation of processing resources to application tasks. They provide the abstraction of a virtual processor. While a server has capacity, its client tasks can execute. But when the budget of the server is exhausted, then the tasks must wait (even if they are the highest-priority tasks in the system) until the budget has been replenished. In this section, we look at two common server technologies (Deferrable [20] and Sporadic [22]) and show how they can be programmed in Ada 2005.[†]

13.4.1 Deferrable Server

In this section, we show how the deferrable server can be constructed. Here, the server has a fixed priority, and when the budget is exhausted the tasks are moved to a background priority. First, we define a type for the parameters of a server and a general-purpose interface for any general server:

```
with Ada.Real_Time, System;
use Ada.Real_Time, System;
package Execution_Servers is
  type Server_Parameters is record
    Period : Time_Span;
    Budget : Time_Span;
    Pri : Priority;
    Background_Pri : Priority;
  end record;
  type Periodic_Execution_Server is synchronized interface;
  procedure Register(ES: in out Periodic_Execution_Server) is abstract;
  type Any_Periodic_Execution_Server is access all Periodic_Execution_Server'Class;
end Execution_Servers;
```

The specific deferrable server can then be defined:

```
with System; use System;
with Ada.Real_Time_Timing_Events; use Ada.Real_Time_Timing_Events;
with Ada.Execution_Time.Group_Budgets; use Ada.Execution_Time.Group_Budgets;
with Ada.Real_Time; use Ada.Real_Time;
```

*This will also occur if the budget goes to zero as a result of a call to `Add` with a large enough negative parameter.

[†]This material is an adaptation from work previously published by the authors [13].

```

package Execution_Servers.Deferrable is
  protected type Deferrable_Server (Params : access Server_Parameters) is new
    Periodic_Execution_Server with
      pragma Interrupt_Priority (Interrupt_Priority'Last);
      procedure Register;
  private
    procedure Timer_Handler(E : in out Timing_Event);
    procedure Group_Handler(G : in out Group_Budget);
    T_Event : Timing_Event;
    G_Budget : Group_Budget;
    First : Boolean := True;
  end Deferrable_Server;
end Execution_Servers.Deferrable;

```

To illustrate the use of the server, a client aperiodic task is included:

```

task Aperiodic_Task is
  pragma Priority(Some_Value);
  -- only used to control initial execution
end Aperiodic_Task;

Control_Params : aliased Server_Parameters := (
  Period => Milliseconds(10),
  Budget => Microseconds(1750),
  Pri => 12,
  Background_Pri => Priority'First);

Con : Deferrable_Server(Control_Params'Access);

-- note the priority of the clients is set to 12, the ceiling priority
-- of the protected object is however at the highest level as it acts
-- as a handler for a timing event

task body Aperiodic_Task is
  -- local data
begin
  Con.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;

```

The body of the deferrable server follows:

```

with Ada_Dynamic_Priorities; use Ada_Dynamic_Priorities;
with Ada.Task_Identification; use Ada.Task_Identification;
package body Execution_Servers.Deferrable is
  protected body Deferrable_Server is
    procedure Register is
      begin
        if First then
          First := False;
          G_Budget.Add(Params.Budget);
          T_Event.Set_Handler(Params.Period, Timer_Handler'Access);
          G_Budget.Set_Handler(Group_Handler'Access);
        end if;
        G_Budget.Add_Task(Current_Task);
        if G_Budget.Budget_Has_Expired then
          Set_Priority(Params.Background_Pri);
          -- sets client task to background priority
        else

```

```

    Set_Priority(Params.Pri);
    -- sets client task to server's 'priority'
end if;
end Register;

procedure Timer_Handler(E : in out Timing_Event) is
    T_Array : Task_Array := G_Budget.Members;
begin
    G_Budget.Replenish(Params.Budget);
    for I in T_Array'range loop
        Set_Priority(Params.Pri,T_Array(I));
    end loop;
    E.Set_Handler(Params.Period,Timer_Handler'Access);
end Timer_Handler;

procedure Group_Handler(G : in out Group_Budget) is
    T_Array : Task_Array := G_Budget.Members;
begin
    for I in T_Array'range loop
        Set_Priority(Params.Background_Pri,T_Array(I));
    end loop;
end Group_Handler;
end Deferrable_Server;
end Execution_Servers.Deferrable;

```

The first client task to call the controller assigns the required budget and sets up the two handlers, one for a group budget and one for a timing event. The timing event is used to replenish the budget (and reset the priority of the client tasks), and a group budget is used to lower the priority of the client tasks. When a task registers, it is running outside the budget, so it is necessary to check if the budget is actually exhausted during registration. If it is, then the priority of the task must be set to the low value.

In the above example, the client tasks are switched to background priority when the budget is exhausted. An alternative algorithm is to suspend the tasks altogether until there is more resource. This can be done in Ada using the operations `Hold` and `Continue` from the predefined package `Asynchronous_Task_Control`.

13.4.2 Sporadic Server

To programme the sporadic server also requires the use of timing events and group budgets. In the following, there is a single controller for each sporadic task, so strictly speaking it is not a group budget and a timer could be used to program the server. However, the example can be expanded to support more than one task [14], and even for one task, the server is easier to construct with a group budget. The rules for the sporadic server are as follows (following the POSIX standard):

- If there is adequate budget, a task that arrives at time t and executed for time c will result in capacity c being returned to the server at time $t + T$, where T is the “period” of the server.
- If there is no budget at time t , then the calling task is delayed until the budget becomes available, say, at time s ; this capacity is returned at time $s + T$.
- If there is some budget available x (with $x < c$), then the task will immediately use this budget (and it will be returned at time $t + T$); later when at time s , say, further adequate budget becomes available then the task will continue, and $c - x$ will be returned at time $s + T$.
- If time s is before the task has used its initial budget then when it finishes, all of c is returned at time $t + T$.

In addition, we assume that the client tasks do not suspend themselves during their execution.

The sporadic task registers and then has the following straightforward structure:

```
task body Sporadic_Task is begin
  Sporadic_Controller.Register;
  -- any necessary initializations etc
  loop
    Sporadic_Controller.Wait_For_Next_Invocation;
    -- undertake the work of the task
  end loop;
end Sporadic_Task;
```

Each time the task calls its controller, the amount of computation time it used last time must be noted and returned to the budget at the appropriate time in the future. To do this (and still block the task until its release event occurs) requires the use of Ada's requeue mechanism. Although there is only one task, it may execute a number of times (using less than the budget each time), and hence there can be more than one timing event outstanding. To enable a single handler to deal with all these requires the timing event to be extended to include the amount of budget that must be returned. We will use a dynamic algorithm that defines a new timing event every time a replenish event should occur. This requires an access type:

```
type Budget_Event is new Timing_Event with record
  Bud : Time_Span;
end record;

type Bud_Event is access Budget_Event;
```

The full code for the server is as follows. Its behavior and examples of its execution will be given later.

```
protected type Sp_Server(Params : access Server_Parameters) is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Wait_For_Next_Invocation;
  procedure Register;
  procedure Release_Sporadic;
private
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
  entry Wait_For;
  TB_Event : Bud_Event;
  G_Budget : Group_Budget;
  Start_Budget : Time_Span;
  Release_Time : Real_Time.Time;
  ID : Task_ID;
  Barrier : Boolean := False;
  Task_Executing : Boolean := True;
end Sp_Server;
```

```
Sporadic_Controller : SP_Server(P'Access);
  -- for some appropriate parameter object P
```

```
protected body Sp_Server is
  procedure Register is
  begin
    ID := Current_Task;
    G_Budget.Add_Task(ID);
    G_Budget.Add(Params.Budget);
    G_Budget.Set_Handler(Group_Handler'Access);
    Release_Time := Real_Time.Clock;
    Start_Budget := Params.Budget;
  end Register;
  entry Wait_For_Next_Invocation when True is
```



```

begin
    -- work out how much budget used, construct the timing event and set the handler
    Start_Budget := Start_Budget - G_Budget.Budget_Remaining;
    TB_Event := new Budget_Event;
    TB_Event.Bud := Start_Budget;
    TB_Event.Set_Handler(Release_Time+Params.Period, Timer_Handler'Access);
    Task_Executing := False;
    requeue Wait_For with abort;
end Wait_For_Next_Invocation;

entry Wait_For when Barrier is
begin
    if not G_Budget.Budget_Has_Expired then
        Release_Time := Real_Time.Clock;
        Start_Budget := G_Budget.Budget_Remaining;
        Set_Priority(Params.Pri,ID);
    end if;
    Barrier := False;
    Task_Executing := True;
end Wait_For;

procedure Release_Sporadic is
begin
    Barrier := True;
end Release_Sporadic;

procedure Timer_Handler(E : in out Timing_Event) is
    Bud : Time_Span;
begin
    Bud := Budget_Event(Timing_Event'Class(E)).Bud;
    if G_Budget.Budget_Has_Expired and Task_Executing then
        Release_Time := Real_Time.Clock;
        Start_Budget := Bud;
        G_Budget.Replenish(Bud);
        Set_Priority(Params.Pri,ID);
    elsif not G_Budget.Budget_Has_Expired and Task_Executing then
        G_Budget.Add(Bud);
        Start_Budget := Start_Budget + Bud;
    else
        G_Budget.Add(Bud);
    end if;
end Timer_Handler;

procedure Group_Handler(G : in out Group_Budget) is
begin
    -- a replenish event required for the budget used so far
    TB_Event := new Budget_Event;
    TB_Event.Bud := Start_Budget;
    TB_Event.Set_Handler(Release_Time+Params.Period,Timer_Handler'Access);
    Set_Priority(Params.Background_Pri,ID);
    Start_Budget := Time_Span_Zero;
end Group_Handler;
end Sp_Server;

```

To understand how this algorithm works, consider a sporadic server with a budget of 4 ms and a replenishment interval of 20 ms. The task will first call `Register`; this will set up the group budget, add the task to this group budget, and note the time of the registration (for illustrative purposes, assume the

clock ticks at 1 ms intervals and that the registration took place at time 1). It also notes that its starting budget is 4 ms. After the execution of other initialization activities, the task will call in to await its release event. Assume this initial phase of execution takes 1 ms. Within `Wait_For_Next_Invocation`, a timing event is constructed that will trigger at time 21 with the budget parameter set at 1 ms.

The external call to `Release_Sporadic` now occurs at time 15. The task is released and, as there is budget available, the release time of the task is noted (15) as is the current capacity of the budget (which is 3). If the task can execute immediately (i.e., no high-priority task is runnable) then it will start to use the budget. If its CPU requirement is, say, 4 ms, then it will execute for 3 ms and then, at time 18, the budget handler will be triggered as the budget has been exhausted. In this handler, a timing event is again constructed; its trigger time is 35 and its budget parameter is 3 ms. The task is given a background priority where it may or may not be able to execute. Assume first that it does not execute (i.e., there are other runnable task will priorities above the background). The next event will be the triggering of the first timing event at time 21. This will add 1 ms to the budget and will allow the task to continue to execute at its higher-priority level. The time of release is noted (21) and the budget outstanding (1 ms). If the task terminates within this 1 ms capacity, then it will call `Wait_For_Next_Invocation` again and a timing event will be created for triggered at time 41 (with parameter 1 ms).

If the task while at the background priority was able to execute, then this would not impact on the budget (which is zero). If it gets as far as completing its execution, then when it calls `Wait_For_Next_Invocation`, a timing event with parameter 0 ms will be constructed—this is inefficient, but a rare event and hence probably better to allow rather than test for the nonzero parameter.

To illustrate just one further feature of the algorithm, assume the call of `Release_Sporadic` occurred at time 19 (rather than 15). Now, the task will start executing at time 19 and run until it uses up the available budget at time 22. However, during that interval, the triggering of the timing event will occur at time 21. This will add 1 ms to the budget but have no further effects as the budget is nonzero at that time. The task can now run to completion and, when it calls `Wait_For_Next_Invocation`, a timing event with a parameter of 4 ms and a triggering time of 39 will be constructed. It may appear that this replenishment time is too soon (should it not be 40?), but the analysis of the sporadic server algorithm does allow this optimization.

The above code has a single client and uses dynamically created timing objects. Adding more clients is straightforward. The alternative to dynamically creating timing objects is to reuse a finite pool defined within the controller.

As CPU time monitoring (and hence budget monitoring) is not exact, the above algorithm is likely to suffer from drift—in the sense that the budget returned is unlikely to be exactly the equivalent of the amount used. To counter this, some form of reasserting the maximum budget is required. For example, the `Timer_Handler` routine could be of the form:

```
procedure Timer_Handler(E : in out Timing_Event) is
begin
    ...
    G_Budget.Add(Bud);
    if Budget_Remaining(G_Budget) > Params.Budget then
        Replenish(G_Budget, Params.Budget);
    end if;
    ...
end Timer_Handler;
```

This will prevent too much extra budget being created. To counter budget leakage, it would be necessary to identify when the task has not run for time `Period` and then make sure the budget was at its maximum level (using `Replenish`). This could be achieved with a further timing event that is set when the task is blocked on `Wait_For` and triggered at this time plus `Period` unless it is canceled in `Release_Sporadic`.

13.5 Support for Deadlines

EDF is a popular paradigm as it has been proven to be the most effective scheme available, in the sense that if a set of tasks is schedulable by any dispatching policy then it will also be schedulable by EDF. To support EDF requires two language features:

- Representation of deadline for a task
- Representation of preemption level for a protected object

The first is obviously required; the second is the EDF equivalent of priority ceilings and allows protected objects to be “shared” by multiple tasks [2] (see the next two sections).

A deadline is usually defined to be *absolute* if it refers to a specific (future) point in time; for example, there is an absolute deadline on the completion of the first draft of this chapter by July 1, 2006 (presumably at 23:59:59). A *relative* deadline is one that is anchored to the current time: “We have one month to finish this chapter.” Obviously, if the current time is known, then a relative deadline can be translated into an absolute one. Repetitive (periodic) tasks often have a static relative deadline (e.g., 10 ms after release) but will have a different absolute deadline every time they are released. The EDF scheme is more accurately expressed as *earliest absolute deadline first*.

It is an anomaly that Ada 95’s support for real time does not extend as far as having a direct representation for “deadline,” but Ada 2005 has rectified this by providing the following package:

```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline := Ada.Real_Time.Time_Last;
  procedure Set_Deadline(D : in Deadline; T : in Ada.Task_Identification.Task_ID :=
    Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline(Delay_Until_Time : in Ada.Real_Time.Time;
    TS : in Ada.Real_Time.Time_Span);
  function Get_Deadline(T : in Ada.Task_Identification.Task_ID :=
    Ada.Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

The meaning of most of the operations of this package should be clear. A call of `Delay_Until_And_Set_Deadline` delays the calling task until time `Delay_Until_Time`. When the task becomes runnable again, it will have deadline `Delay_Until_Time + TS`. The inclusion of this procedure reflects a common task structure for periodic activity. Examples of use are given later in this chapter. Note all tasks are given a deadline—if one has not been explicitly set, then `Time_Last` (the far distant future) is given as the default.

A pragma is also provided to give an initial relative deadline to a task to control dispatching during activation.

A program can manipulate deadlines and even catch a deadline overrun via the ATC (select-then-abort) feature:

```
loop
  select
    delay until Ada.Dispatching.EDF.Get_Deadline;
    -- action to be take when deadline missed
  then abort
    -- code
  end select;
  -- set next release condition
  -- and next absolute deadline
end loop;
```

However, note that an asynchronous change to the deadline while the “code” is being executed will not be reflected in the current loop.

13.6 Baker’s Preemption Level Protocol for Protected Objects

One of the major features that Ada 95 provided is the support for monitors via the introduction of the protected object. Although the rendezvous is a powerful synchronization and communication primitive, it does not easily lead to tight scheduling analysis. Rather a more asynchronous scheme is desirable for real-time applications. The protected object provides this form of communication. With this construct, and the use of fixed priorities for tasks and a priority ceiling protocol for protected objects, it is possible to predict worst-case completion times for tasks.

With standard fixed priority scheduling, *priority* is actually used for two distinct purposes:

- To control dispatching
- To facilitate an efficient and safe way of sharing protected data

The latter is known as the priority ceiling protocol. In Baker’s stack-based protocol, two distinct notions are introduced for these policies [2]:

- EDF to control dispatching*
- Preemption levels to control the sharing of protected data

With preemption levels (which is a very similar notion to priority), each task is assigned a static preemption level, and each protected object is assigned a ceiling value that is the maximum of the preemption levels of the tasks that call it. At runtime, a newly released task, τ_1 , can preempt the currently running task, τ_2 , if and only if the

- Absolute deadline of τ_1 is earlier (i.e., sooner) than deadline of τ_2
- Preemption level of τ_1 is higher than the preemption of any locked protected object

With this protocol, it is possible to show that the mutual exclusion (over the protected object on a single processor) is ensured by the protocol itself (in a similar way to that delivered by fixed priority scheduling and ceiling priorities). Baker also showed, for the classic problem of scheduling a fixed set of periodic tasks, that if preemption levels are assigned according to each task’s relative deadline, then a task can suffer at most a single block from any task with a longer deadline. Again, this result is identical to that obtained for fixed priority scheduling. Note preemption levels must be assigned inverse to relative deadline (the smaller the relative deadline, the higher the preemption level).

Although EDF scheduling results in higher resource usage and Baker’s protocol furnishes efficient data sharing between tasks, certain forms of analysis and application characteristics are more complex than the fixed priority equivalent. However, response time analysis, for example, does exist for EDF scheduling [21,23].

13.7 Supporting EDF Scheduling in Ada

Ada 95 did not support EDF scheduling and hence new facilities needed to be added to Ada 2005; however, it is desirable for fixed priority and EDF scheduling to work together (see Section 13.8). To keep language changes to a minimum, Ada 2005 does not attempt to define a new locking policy but uses the existing ceiling locking rules without change. Priority, as currently defined, is used to represent preemption levels. EDF scheduling is defined by a new dispatching policy.

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

*His paper actually proposes a more general model of which EDF dispatching is an example.

With Ada 95, dispatching is defined by a model that has a number of ready queues, one per priority level. Each queue, for the standard model, is ordered in a FIFO manner. Tasks have a base priority (assigned by pragma `Priority` and changed, if desired, by the use of `Dynamic_Priority.Set_Priority`). They may also have a higher active priority if they inherit such a value during, for example, a rendezvous or an execution within a protected object. Preemptive behavior is enforced by requiring a context switch from the current running task if there is a higher-priority nonempty ready queue. This is known, within the Ada Reference Manual, as a *dispatching point*. At any dispatching point, the current running task is returned to its ready queue, and another task (or indeed the same task if appropriate) is taken from its ready queue, and executed. It should be noted that this is an abstract model of the required behavior; an implementation does not need to deliver the required semantics in this way. This also applies to the new model defined below.

13.7.1 The Rules of EDF Dispatching

The basic preemption rule given earlier for Baker's protocol, while defining the fundamental behavior, does not give a complete model. For example, it is necessary to define what happens to a newly released task that is not entitled to preempt. A complete model, within the context of Ada's ready queues, is defined by the following rules. (Note that if `EDF_Across_Priorities` is defined then all ready queues within the range `Priority'First ... Priority'Last` are ordered by deadline.)

- Whenever a task T is added to a ready queue, other than when it is preempted, it is placed on the ready queue with the highest priority R, if one exists, such that:
 - Another task, S, is executing within a protected object with ceiling priority R;
 - Task T has an earlier deadline than task S; and
 - The base priority of task T is greater than R.

If no such ready queue exists, the task is added to the ready queue for `Priority'First`.

- When a task is chosen for execution, it runs with the active priority of the ready queue from which the task was taken. If it inherits a higher active priority, it will return to its original active priority when it no longer inherits the higher level.

It follows that if no protected objects are in use at the time of the release of T, then T will be placed in ready queue at level `Priority'First` at the position dictated by its deadline.

A task dispatching point occurs for the currently running task T whenever

- A change to the deadline of T takes effect
- A decrease to the deadline of any task on a ready queue for that processor takes effect and the new deadline is earlier than that of the running task
- There is a nonempty ready queue for that processor with a higher priority than the priority of the running task

So dispatching is preemptive, but it may not be clear that the above rules implement a form of Baker's algorithm (as described in Section 13.6). Consider three scenarios. Remember in all these descriptions, the running task is always returned to its ready queue whenever a task arrives. A task (possible the same task) is then chosen to become the running task following the rules defined above.

The system contains four tasks: T1, T2, T3, and T4 and three resources that are implemented as protected objects: R1, R2, and R3. Table 13.1 defines the parameters of these entities.

We are concerned with just a single invocation of each task. The arrival times have been chosen so that the tasks arrive in order of the lowest preemption level task first, etc. We assume all computation times are sufficient to cause the executions to overlap.

The resources are all used by more than one task, but only one at a time and hence the ceiling values of the resources are straightforward to calculate. For R1, it is used by T1 and T4, hence the ceiling preemption level is 4. For R2, it is used by T2 and T3, hence the ceiling value is 3. Finally, for R3, it is used by T1 and T2, the ceiling equals 2 (see Table 13.2).

TABLE 13.1 A Task Set (Time Attributes in Milliseconds)

Task	Relative Deadline D	Preemption Level L	Uses Resources	Arrives at Time	Absolute Deadline A
T1	100	1	R1,R3	0	100
T2	80	2	R2,R3	2	82
T3	60	3	R2	4	64
T4	40	4	R1	6	46

TABLE 13.2 Ceiling Values

Protected Object	Ceiling Value
R1	4
R2	3
R3	2

To implement this, the set of tasks and resources will require ready queues at level 0 (value of `PriorityFirst` in this example) and values up to 4.

Scenario 1

At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running task. This is illustrated in the following where “Level” is the priority level, “Executing” is the name of the task that is currently executing, and “Ready Queue” shows the other nonblocked tasks in the system.

Level	Executing	Ready Queue
0	T1	

At time 2, T2 arrives and is added to ready queue 0 in front of T1 as it has a shorter absolute deadline. Now T2 is chosen for execution.

0	T2	T1
---	----	----

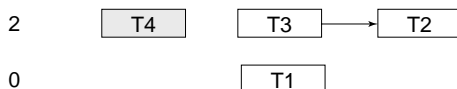
Assume at time 3, T2 calls R3. Its active priority will raise to 2.

2	T2	
0		T1

At time 4, T3 arrives. Task T2 is joined by T3 on queue 2, as T3 has an earlier deadline and a higher preemption level; T3 is at the head of this queue and becomes the running task.

2	T3	T2
0		T1

At time 6, T4 arrives. Tasks T3 and T2 are now joined by T4 as it has a deadline earlier than T2 and a higher preemption level (than 2). Task T4 now becomes the running task and will execute until it completes; any calls it makes on resources R1 will be allowed immediately as this resource is free.

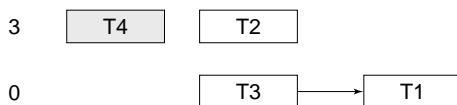


Some time later, T4 will complete then T3 will execute (at priority 2, or 4 if it locks R2) and when it completes, T2 will execute (also at priority 2) until it releases resource R3, at which point its priority will drop to 0, but it will continue to execute. Eventually, when T2 completes, T1 will resume (initially at priority 0—but this will rise if it accesses either of the resources it uses).

Scenario 2

Here we make the simple change that, at time 3, T2 calls R2 instead of R3. Its active priority will raise to 3. Now when T3 arrives at time 4, it will not have a high enough preemption level to join ready queue 3 and will be placed on the lowest queue at level 0 (but ahead of T1). Task T2 continues to execute.

At time 6, T4 arrives. It passes both elements of the test and is placed on queue at level 3 ahead of T2 and therefore preempts it.



At some time later, T4 will complete then T2 will execute (at priority 3) until it releases resource R2, at which point its priority will drop to 0. Now T3 will preempt and becomes the running task.

Scenario 3

For a final example, return to the first scenario but assume T3 makes use of resource R2 before T4 arrives:

At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running task.

At time 2, T2 arrives and is added to ready queue 0 in front of T1.

Assume at time 3, T2 calls R3. Its active priority will raise to 2.

At time 4, T3 arrives and becomes the running task at priority level 2.

At time 5, T3 calls R2 (note all resource requests are always to resources that are currently free) and thus its active priority raises to 3.

At time 6, T4 arrives. There is now one task on queue 0, one on queue 2 (T2 holding resource R3), and one task on queue 3 (T3 holding R2). The highest ready queue that the dispatch rules determine is that at level 3, and hence T4 joins T3 on this queue—but at the head and hence becomes the running task.

13.7.2 An Important Constraint

The above rules and descriptions are, unfortunately, not quite complete. The ready queue for *PriorityFirst* plays a central role in the model as it is, in some senses, the default queue. If a task is not entitled to be put in a higher queue, or if no protected objects are in use, then it is placed in this base queue. Indeed, during the execution of a typical program, most runnable tasks will be in the *PriorityFirst* ready queue most of the time. However, the protocol only works if there are no protected objects with a ceiling at the *PriorityFirst* level. Such ceiling values must be prohibited, but this is not a significant constraint. Ceilings at this level are rare (all user tasks would need to have priority *PriorityFirst*) and the use of $\text{PriorityFirst} + 1$ will not have a major impact. A common practice is to set any ceiling at maximum usage + 1.

13.7.3 Example Task Set

To illustrate the minimal changes that have to be made to the application code to move from one scheduling paradigm to another, consider a simple periodic task scheduled according to the standard fixed priority method. This task has a period of 10 ms.

```
task A is
  pragma Priority(5);
end A;

task body A is
  Next_Release: Ada.Real_Time.Time;
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- application code
    Next_Release := Next_Release + Ada.Real_Time.Milliseconds(10);
    delay until Next_Release;
  end loop;
end A;
```

If EDF dispatching is required (perhaps to make the system schedulable), then very little change is needed. The priority levels of the task and protected objects remain exactly the same. The task's code must, however, now explicitly refer to deadlines:

```
task A is
  pragma Priority(5);
  pragma Relative_Deadline(10); -- gives an initial relative
                                -- deadline of 10 milliseconds
end A;

task body A is
  Next_Release: Ada.Real_Time.Time;
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- application code
    Next_Release := Next_Release + Ada.Real_Time.Milliseconds(10);
    Delay_Until_And_Set_Deadline(Next_Release, Ada.Real_Time.
      Milliseconds(10));
  end loop;
end A;
```

Finally, the dispatching policy must be changed from

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);

to

pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

13.8 Mixed Dispatching Systems

All the above descriptions have assumed that EDF dispatching is an alternative to fixed priority dispatching. The current language definition however goes further and allows mixed systems to be programmed. The system's priority range can be split into a number of distinct nonoverlapping bands. In each band, a specified dispatching policy can be defined. So, for example, there could be a band of fixed priorities on top of a band of EDF with a single round-robin level for non real-time tasks at the bottom.

Obviously, to achieve this, the properties assigned above to level `Priority'First` would need to be redefined for the base of whatever range is used for the EDF tasks. Also rules need to be defined for tasks

communicating across bands (via protected objects or rendezvous) and for tasks changing bands (via the use of `Dynamic_Priorities.Set_Priority`). These rules are defined for Ada 2005.

If, for example, in a five-task system, tasks A and B were critical while tasks C, D, and E were not, then A and B would be assigned priorities 5 and 4, while the other three tasks would be allocated priorities (preemption levels) in the range 1 to 3.

```
pragma Priority_Specific_Dispatching(FIFO_Within_Priorities, 5, 4);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 3, 1);
```

The two integer parameters indicate upper and lower priorities of the band.

There are also usage paradigms in the real-time area that run all tasks in an EDF queue, but then move any “hard” task to a higher fixed priority if there is any danger of its deadline being missed [5,10,18]. Such algorithms are simple to achieve with the facilities defined in Ada 2005.

13.9 Conclusion

Many of the new features of Ada 2005 take the form of events that are fired by the program’s execution environment. Example events are

- When a task terminates
- When a task’s execution time reaches a defined value
- When time reaches a defined value
- When a group budget reaches zero, and
- When an interrupt occurs (existing Ada 95 feature)

All these events trigger the execution of an application-defined handler programmed as a protected procedure with a ceiling priority that determines the priority at which the handler is executed.

One focus of this chapter has been on the construction of execution-time server abstractions using these new Ada 2005 facilities. Simple servers such as the deferrable server are straightforward and need just a simple timing event and group budget. The sporadic server in contrast is quite complicated and its implementation is a testament to the expressive power of the language.

EDF scheduling is the policy of choice in many real-time systems, particularly those that are less critical and where minimizing resource usage is critical (e.g., mobile embedded systems and multimedia applications). The features described in this chapter show how full support for EDF can be achieved in a manner fully compatible with the fixed priority scheduling model. Preemption level control over the use of Protected Objects is achieved with no change to the ceiling locking policy. The inclusion of EDF and mixed scheduling within the Ada language significantly increases the expressive power, usability, and applicability of the language. It is a strong indication of the continuing relevance of Ada. Together with other features in Ada 2005 (e.g., Round-Robin scheduling), there is now a language platform that can deliver modern real-time scheduling theory to the systems engineer.

In this chapter, two major parts of Ada 2005’s new provisions have been described: programming servers and EDF dispatching. The material on servers is all presented within the context of fixed priority dispatching. There are however a number of other server algorithms for fixed priority scheduling and others, such as the bandwidth preserving server, that are defined to work with EDF [1,17]. Space restrictions prevent any further examples in this chapter, but servers can be produced to work with EDF dispatching.

13.10 Postscript—Ada and Java

This and the previous chapter have described the two main concurrent programming languages for implementing real-time systems. The expressive power of Java (with the RTSJ) and Ada 2005 far exceeds that provided by any other engineering languages. Although the style of the syntax and the programming

approaches differ, there are a large number of similarities between the two languages. For example, the use of and support for concurrency, clocks, delay statements, monitor-like structures for mutual exclusion, ATC, fixed priority preemptive scheduling with a priority ceiling protocol, dynamic priorities, CPU budgeting, and enforcement. Taken together, these features (and others) allow a wide range of real-time systems to be constructed ranging from the traditional simple hard real-time system to the more flexible and open soft systems.

One area where the RTSJ provides significant extra support is in its approach to memory management. The use of distinctive patterns of memory accesses within real-time OO software coupled with the limited amounts of physical memory often associated with embedded systems requires language features that can address reuse and garbage collection. The RTSJ is the first language to try and address these issues although, as described in the previous chapter, there are still a number of outstanding issues with the approach adopted.

Ada is a much more mature language, in that it has gone through a number of definitions over the last 30 years. The latest Ada 2005 version has included a number of new features that have emerged from the real-time research community. For example, the use of EDF scheduling and the ability to program servers mean that Ada remains a key language for real-time system implementation. It is likely in the future that the RTSJ (with its defined processing group parameters) will also move to support these features and facilities, and hence there remains a strong conduit from research results to industrial practice via these two languages.

Acknowledgments

The authors wish to thank the members of ISO committee ARG, attendees of the IRTAW series, and members of the ARTIST project for their input to the issues discussed in this chapter.

References

1. L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Florence, Italy, June 1999.
2. T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1), pp. 67–99, March 1991.
3. J.G.P. Barnes. *Programming in Ada 2005*. Addison-Wesley, Reading, MA, 2006.
4. G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the Real-Time Systems Symposium*, pp. 328–335, Lisbon, Portugal, 2004. Computer Society, IEEE.
5. G. Bernat and A. Burns. Combining (n/m)-hard deadlines and dual priority scheduling. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 46–57, 1997.
6. G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 68–78, 1999.
7. R. Brukardt (ed). Ada 2005 reference manual. Technical Report, ISO, 2006.
8. A. Burns and G. Bernat. Implementing a flexible scheduler in Ada. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Leuven*, pp. 179–190. Springer, Heidelberg, LNCS 2043, 2001.
9. A. Burns, M. González Harbour, and A.J. Wellings. A round robin scheduling policy for Ada. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, pp. 334–343. Springer, Heidelberg, LNCS 2655, 2003.
10. A. Burns and A.J. Wellings. Dual priority scheduling in Ada 95 and real-time POSIX. In *Proceedings of the 21st IFAC/IFIP Workshop on Real-Time Programming, W RTP’96*, pp. 45–50, 1996.
11. A. Burns and A.J. Wellings. Accessing delay queues. In *Proceedings of IRTAW11, Ada Letters*, Vol XXII(4), pp. 72–76, 2002.

12. A. Burns and A.J. Wellings. Task attribute-based scheduling—extending Ada’s support for scheduling. In T. Vardenega, editor, *Proceedings of the 12th International Real-Time Ada Workshop*, Vol XXIII, pp. 36–41. ACM Ada Letters, 2003.
13. A. Burns and A.J. Wellings. Programming execution-time servers in Ada 2005. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2006.
14. A. Burns and A.J. Wellings. *Concurrency and Real-Time Programming in Ada*. Cambridge University Press, 2007.
15. A. Burns, A.J. Wellings, and T. Taft. Supporting deadlines and EDF scheduling in Ada. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, pp. 156–165. Springer, Heidelberg, LNCS 3063, 2004.
16. A. Burns, A.J. Wellings, and T. Vardanega. Report of session: Flexible scheduling in Ada. In *Proceedings of IRTAW 12, Ada Letters*, Vol XXIII(4), pp. 32–35, 2003.
17. M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2001.
18. R.I. Davis and A.J. Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 100–109, 1995.
19. J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pp. 110–123, 1992.
20. J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pp. 261–270, 1987.
21. J.C. Palencia and M. González Harbour. Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pp. 200–209, 2003.
22. B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1: 27–69, 1989.
23. M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, 1996.
24. A.J. Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? In *Proceedings of IRTAW 12, Ada Letters*, Vol XXIII(4), pp. 16–21, 2003.

14

Synchronous Programming*

Paul Caspi

CNRS, Verimag Laboratory

Pascal Raymond

CNRS, Verimag Laboratory

Stavros Tripakis

*CNRS/Verimag and Cadence
Berkeley Labs*

14.1	Introduction	14-1
14.2	From Practice	14-2
	Programming Practices of Control and Electronic Engineers • The Interest of the Approach • Limitations of the Approach	
14.3	To Theory	14-3
	Milner's Synchronous Calculus of Communicating Systems • Synchrony as a Programming Paradigm • The Causality Problem in Synchronous Composition	
14.4	Some Languages and Compilers	14-5
	Esterel • Lustre • Causality Checking • Code Generation	
14.5	Back to Practice	14-14
	Single-Processor, Preemptive Multitasking Implementations • Distributed Implementations	
14.6	Conclusions and Perspectives	14-19

14.1 Introduction

Synchronous programming is a school of thought in embedded software, which has been in the air for a while but without being always given clear and thorough explanations. This has led to many misunderstandings as, for instance, opinions like: “we do not use synchronous programming because null execution time is not realistic and infinitely fast machines do not exist.” At the same time many practitioners adopted (implicitly or not) the synchronous programming principles and, by now, synchronous programming is used in real-world safety-critical systems like the “fly-by-wire” systems of Airbus commercial aircrafts (A340, A380). This discrepancy between a valuable industrial usage and charges of unrealism clearly shows that better explanations are urgently needed. This chapter hopefully aims at contributing to a better understanding.

To provide these mandatory explanations, we propose to take a historical perspective by showing that synchronous programming has arisen from the interaction between

1. The practices of control and electronic engineers in the early times of digital computers when they adopted this new technology in replacement of analog devices (Section 14.2) and
2. The concerns of some computer theoreticians who were trying to better account for the modeling of reactive computing systems (Section 14.3)

Indeed, synchronous languages arose from the (unlikely) convergence of these two very different concerns and we shall explain how this convergence took place. Then we describe some of the languages in

*This work has been partially supported by the European project ARTIST2 (IST-004527).

Section 14.4. As we shall see, these languages are both quite useful and yet with very restrictive implementation capabilities, and in Section 14.5 we describe the possible extensions that have been brought, through time, to the initial concepts.

14.2 From Practice . . .

14.2.1 Programming Practices of Control and Electronic Engineers

In the early eighties, microprocessors appeared as very efficient and versatile devices, and control and electronic engineers were quickly moving to using them. Yet, these devices were very different from the ones they were used to, that is, analog and discrete components coming along with a “data sheet” basically describing some “input–output” behavior. Here, on the contrary, the device was coming along with a larger user manual, encompassing a so-called “instruction set” that users had to get used to. It should be noted that these users were almost totally ignorant of elementary computer engineering concepts like operating systems, semaphores, monitors, and the like. Consequently, after some trial and error, most of them finally settled with a very simple program structure, triggered by a single periodic real-time clock, as shown in Table 14.1.

14.2.2 The Interest of the Approach

Though this program structure may appear as very basic, it still has many advantages in the restricted context it is applied to.

It requires a very simple operating system, if any. In fact, it uses a single interrupt, the real-time clock, and from the real-time requirement, this interrupt should occur only when the loop body is over, that is to say, when the computer is idle. Thus, there is no need for context switching and this kind of program can even be implemented on a “bare” machine.

Timing analysis is the simplest one. Timing analysis amounts to checking that the worst-case execution time (WCET) C of the loop body is smaller than the clock period T . To make this check even easier, people took care of forbidding the use of unbounded loop constructs in the loop body as well as forbidding the use of dynamic memory and recursion so as to avoid many execution errors. Despite these restrictions, the problem of assessing WCET has been constantly becoming harder because of more and more complex hardware architectures with pipelines, caches, etc. Still, this is clearly the simplest possible timing analysis.

It perfectly matches the needs of sampled-data control and signal processing. Periodic sampling has a long-lasting tradition in control and signal processing and comes equipped with well-established theories like the celebrated Shannon–Nyquist sampling theorem or the sampled-data control theory [1].

All this is very simple, safe, and efficient. Safety is also very important in this context, as many of these control programs apply to safety-critical systems such as commercial aircraft flight control, nuclear plant emergency shutdown, railway signaling, and so on. These systems could not have experienced the problems met by the Mars Pathfinder due to priority inversion [2] in programs based on tasks and semaphores. As a matter of fact, operating systems are extremely difficult to debug and validate. Certification authorities in

TABLE 14.1 Basic Synchronous Program Structure

```
initialize state;
loop
  wait clock-tick;
  read inputs;
  compute outputs and state;
  emit outputs
end-loop
```

safety-critical domains are aware of it and reluctantly accept to use them. Even the designers of these systems are aware of it and issue warnings about their use, like the warning about the use of priorities in Java [3].*

14.2.3 Limitations of the Approach

Thus, this very simple execution scheme should be used as much as possible. Yet, there are situations where this scheme is not sufficient and more involved ones must be used. We can list some of these situations here:

Multiperiodic systems. When the system under control has several degrees of freedom with different dynamics, the execution periods of the corresponding controls may be different too and, then, the single-loop scheme is rather inefficient. In this case, an organization of the software into periodic tasks (one task for each period) scheduled by a preemptive scheduler provides a much better utilization of the processor. In this case, checking the timing properties is somewhat harder than in the single-loop case, but there exist well-established schedulability checking methods for addressing this issue [4].

Yet, a problem remains when the different tasks need to communicate and exchange data. It is here where mixing tasks and semaphores raises difficult problems. We shall see in Section 14.5.1, some ways of addressing this problem in a way, which is consistent with the synchronous programming philosophy.

Discrete-event and mixed systems. In many control applications, discrete events are tightly combined with continuous control. A popular technique consists of sampling these discrete events in the same way as continuous control.[†] When such sampling is performed, software can be structured as described previously in the periodic and multiperiodic cases.

There are still more complex cases when some very urgent tasks are triggered by some nonperiodic event. In these cases, more complex scheduling methods need to be used as, for instance, deadline monotonic scheduling [5].

In every case but the single-period one, the problem of communication remains and has to be addressed (see Section 14.5.1).

Distributed systems. Finally, most control systems are distributed, for several reasons, for instance, sensor and actuator location, computing power, redundancy linked with fault tolerance, etc. Clearly, most of the computing structures considered above for single computers get more complicated when one moves toward distribution. We shall present in Section 14.5.2.1 a solution to this problem for the case of a synchronous distributed execution platform.

14.3 To Theory

14.3.1 Milner's Synchronous Calculus of Communicating Systems

In the late seventies, computer theorists were thinking of generalizing usual formalisms like the λ -calculus and language theory so as to encompass concurrency, which was urgently needed in view of the ever-growing interest in reactive systems. In this setting, C.A.R. Hoare [6] proposed *rendezvous* as the structuring concept for both communication and synchronization between concurrent computing activities. Slightly later, R. Milner [7] used this concept to build a general algebraic concurrent framework called *Calculus of Communicating Systems* (CCS) generalizing both λ -calculus and the calculus of regular expressions of language theory. This approach was very successful and yielded valuable by-products like the discovery of the bisimulation concept.

But it soon appeared that this framework was not expressive enough in that it did not allow the expression of such a current object of reactive systems as a simple *watchdog*. Thus, Milner went back to work and invented the *Synchronous Calculus of Communicating Systems* (SCCS) [8], which could overcome this

*... use thread priority only to affect scheduling policy for efficiency purposes; do not rely on it for algorithm correctness.

[†]Although we can remark that there does not exist a sampling theory of discrete-event systems as well established as the one, which exists for continuous control.

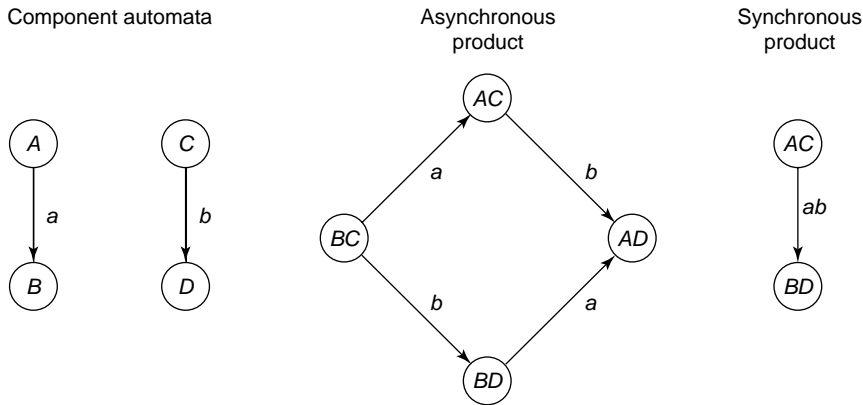


FIGURE 14.1 Asynchronous and synchronous products of automata.

drawback. These two calculi are based on a different interpretation of the parallel construct, that is, of what takes place when two computing agents operate concurrently. This difference is shown in Figure 14.1, while in the asynchronous version of the calculus, at each time step, one agent is chosen nondeterministically to perform a step, in the synchronous case, each agent performs a step at each time. Milner also showed that SCCS can simulate CCS and thus that CCS is a *subtheory* of SCCS. This, in our opinion, provides some theoretical support to control practices: this means that synchronous primitives are stronger than and encompass asynchronous ones.

14.3.2 Synchrony as a Programming Paradigm

To this landscape, G. Berry and the Esterel team added several interesting remarks [9]:

- First they remarked that the synchronous product is *deterministic* and yields less states than the asynchronous one. This has two valuable consequences: programs are more deterministic and thus yield easier debugging and test and have less state explosion and thus yield easier formal verification.
- They also remarked that a step in the synchronous parallel composition corresponds exactly to one step of each of the component computing agents. This provides a “natural” notion of *logical time* shared by all components, which allows an easier and clearer reasoning about time. In this setting, several activities can take place *simultaneously*, that is, *at the same logical time instant*. This in turn, gave birth to the celebrated *infinitely fast machine* and *null execution time* mottos.
- Finally, they remarked that simultaneity provided for an important communication scheme, *instant broadcast*: in the same time step, an agent can emit a message and several other agents can receive it.

14.3.3 The Causality Problem in Synchronous Composition

These features were appealing. But they also yielded many problems because they allowed the so-called *instant dialogs*: an agent can, at the same step, emit and receive messages and this can give rise to paradoxical situations when some events (receptions and emissions) are conditioned to some other events. Examples of these paradoxes are

- An event occurs if it does not (there is no solution) and
- An event occurs if it occurs (there can be two solutions: either the event occurs or it does not)

This was a general problem of synchronous parallel composition, the *causality problem*. This problem arose in many frameworks even for those, which did not recognize themselves in the synchronous programming school, that is, each time concurrent communicating activities take place synchronously.

Thus, in any of these frameworks, solutions had to be found and, indeed, many of them had been proposed. Let us list some of them:

Impose unit delay constraints on instant dialogs. This is the most conservative solution as it suppresses the problem. This solution has been adopted in Lustre [10], SL [11], and several execution modes of discrete-time Simulink.*

Rely on an arbitrary ordering of actions within a time step. This solution has often been taken in simulation-based frameworks. This is the case of some Simulink simulation modes and, most notably, of Stateflow. The advantage comes from the fact that no restrictions need to be imposed on instant dialogs. The main drawback is that it provides an obscure semantic in which the behavior may depend on strange features such as, for instance, the lexicographic order of block names or the graphical position of blocks in a drawing [12].

Order activities within a time step according to a small-step semantic. This is the approach followed by the STATEMATE semantics of Statecharts [13].† The difference with the previous approach is that it does not guarantee the uniqueness of solutions.

Reject nonunique solutions. This is the approach taken in Signal [14] and Argos [15]. At each instant, the different possible emission and receptions patterns are gathered and the design is rejected if there is no solution or if there is more than one solution. The drawback of this proposal is that it requires solving, at each time step, an NP-complete problem akin to solving Boolean equations.

Reject nonunique constructive solutions. This is the approach taken by the Esterel team. It is based on a deep criticism of the previous solution which, according to G. Berry, does not allow designers to understand and get insight on why their design is accepted or not: a designer does not solve Boolean equations in his head when designing a system [16]. Moreover, constructiveness is easier to solve and is also consistent with the rules of electric stabilization in hardware design.

As we can see, there are many possible solutions to this problem and it is not yet clear whether a satisfactory one has already emerged.

14.4 Some Languages and Compilers

Although they are based on the same principles, different synchronous languages propose different programming styles:

- Dataflow languages are inspired by popular formalisms like block diagrams and sequential circuits. Programs are described as networks of operating nodes connected by wires. The actual syntax can be textual, like in Lustre [10] or Signal [14,17], or graphical-like in the industrial tool SCADE [18,19]. Lustre and SCADE are purely functional, and provide a simple notion of clock, while Signal is relational and provides a more sophisticated clock calculus.
- Imperative languages are those where the program structure reflects the control flow. Esterel [9] is such a language, providing not only control structures resembling the ones of classical imperative languages (sequence, loops, etc.) but also interrupts and concurrency. In graphical languages, the control structure is described with finite automata. Argos [15] and SynchCharts [20], both inspired by Statecharts, are such languages, where programs consist of hierarchical, concurrent Mealy machines.

This section gives a flavor of two languages: Esterel as an example of an imperative language and Lustre as an example of a dataflow language. It also presents some classical solutions for the causality problem, and the basic scheme for code generation.

*<http://www.mathworks.com/products/simulink/>.

†Note that many semantics have been proposed for Statecharts, showing the intrinsic difficulty of the problem.

14.4.1 Esterel

This section presents a flavor of Esterel, a more detailed introduction can be found in Ref. 21. We use an example adapted from that paper: a *speedometer*, which indefinitely computes and outputs the estimated speed of a moving object.

Signals. Inputs/outputs are defined by means of *signals*. At each logical time instant, a signal can be either *present* or *absent*. In the speedometer example, inputs are *pure signals*, that is, events occurring from time to time without carrying any value:

- *sec* is an event, which occurs each second, it is typically provided by some real-time clock;
- *cm* is an event, which occurs each time the considered object has moved by 1 cm, it is typically provided by some sensor.

According to the synchrony hypothesis, the program runs on a discrete clock defining a sequence of logical instants. This clock can be kept abstract: the synchronous hypothesis simply states that this base clock is “fast enough” to capture any occurrence of the inputs *sec* and *cm*.

The output of the program is the estimated speed. It can be present or absent, but when present, a numerical information must be provided (the computed speed). It is thus a *valued signal*, holding (for instance) a floating point value.

The header of the program (a module in Esterel terminology) is then

```
module SPEEDOMETER:
input sec, cm;           % pure signals
output speed: float;    % valued signal
```

Basic statements. Basic statements are related to signals. The presence of a signal in the current logical instant can be tested with the *present S then ... else ...* statement. More sophisticated, the *await S* statement is used to wait for the next logical instant when *S* occurs.

An output signal is present in the current logical instant if and only if it is broadcast by the *emit* statement. Pure signals are emitted using *emit S*, while valued signals are emitted using *emit S(val)*, where *val* is a value of the suitable type.

Variables, sequence, and loops. The other statements provided by Esterel resemble the ones of a classical imperative language. A program can use imperative variables, that is, memory locations. In the example, we use a numerical variable to count the occurrences of *cm*.

Statements can be put in sequence with the semicolon operator. This sequence is *logically* instantaneous, which means that both parts are occurring in the same logical instant. For instance, *emit X; emit Y* means that *X* and *Y* are both present in the current instant, and thus, it is equivalent to *emit Y; emit X*. However, the sequence reflects a *causality* precedence as soon as imperative variables are concerned: *x := 1; x := 2* results in *x = 2*, which is indeed different from *x := 2; x := 1* (result *x = 1*).

The unbounded loop is the (infinite) repetition of the sequence: *loop P end*, where *P* is a statement, is equivalent to *P; P; P; ...*. The loop statement may introduce instantaneous loops where the logical instant never ends. For instance, *loop emit X end* infinitely emits *X* in the same logical instant. Such programs are rejected by the compiler: each control path in a loop must pass through a statement, which “takes” logical time, typically an *await* statement.

Interrupts. Suppose for the time being that the event *cm* is more frequent than *sec*. Then, we can approximate the speed by the number of *cm* received between two occurrences of *sec*. As a consequence, the result will not be accurate if the speed is too low (<1 cm/s).

The normal behavior consists in waiting for a *cm*, then incrementing the counter and so on. This behavior is interrupted when a second occurs: the output *speed* is then emitted with the current value of the counter, and then, the whole process repeats (see Figure 14.2).

Parallel composition. As said before, the result of *speedometer* is not accurate if the speed is too low. In this case, an approximation is the inverse of the number of *sec* between two occurrences of *cm*.

```

module SPEEDOMETER:
input sec, cm;           % pure signals
output speed : double;   % valued signal
loop                    % Infinite behavior
  var cpt := 0 : double in % initialize an imperative variable
    abort                % Aborts the normal behavior ...
    loop                % repeat
      await cm ;         % wait a centimeter,
      cpt := cpt + 1.0    % increment the counter
    end loop
  when sec do            % ... on the next occurrence of second, then
    emit speed(cpt)      % emit speed with the current value of cpt
  end abort
end var
end loop.                % and so on ...

```

FIGURE 14.2 First version of the speedometer in Esterel.

In a new version of the program, two processes are running concurrently. One is based on a centimeter counter and is suitable for speeds > 1 . The other is based on a *sec* counter and is suitable for speeds < 1 . An actual null speed results in not emitting *speed* at all. This example illustrates a problem owing to synchrony, since two concurrent processes are supposed to produce the same output (*speed*). In Esterel this problem is solved as follows: a signal is present if it is emitted by at least one concurrent process, and, in case of a valued signal of type τ , the several values emitted at the same logical instant are combined with an associative operator $\Gamma : \tau \times \tau \mapsto \tau$. The user has to specify this operator when declaring the signal, otherwise concurrent emission will be forbidden by the compiler.

This standard solution can be used in our example, since we know that, whenever both *sec* and *cm* are present, at most one counter is relevant (≥ 0), while the other is equal to 0. As a consequence, the values emitted on the signal *speed* can be safely combined with $+$ (see Figure 14.3). However, this solution is somewhat *ad hoc*, and one may prefer to program a suitable “combinator” process, running concurrently with the other processes.

14.4.2 Lustre

Lustre programs as dataflow networks. A Lustre program denotes a network of operators connected by wires. Figure 14.4 shows both the graphical view and the textual (Lustre) version: *x* and *y* are input wires, that is, free variables. Other variables correspond to wires connected to some source. In Lustre, they are either output variables (*m*) or “hidden” local variables (*c*). Each linked variable must be defined by a unique equation. Note that it is not necessary to name every wire; in the example, the local variable *c* can be avoided by simply writing *m* = if *x* \geq *y* then *x* else *y*. Moreover, the order of the equations is not relevant; a Lustre equation is not an assignment: it expresses a global invariant (e.g., the value of *c* is always the sum of the values of *x* and *y*).

Semantics. In Lustre, every variable denotes a synchronous *flow*, that is, a function from the discrete time (\mathbb{N}) to a declared domain of values. As a consequence, the semantics of the example above is obvious: the program takes two real flows *x* and *y*, and computes, step by step, the flow *m* such that

$$m_t = \text{if } x_t \geq y_t \text{ then } x_t \text{ else } y_t \quad \forall t \in \mathbb{N}$$

Temporal operators. The Lustre language provides the basic data types *bool*, *int*, and *real*, and all the standard arithmetic and logic operators (*and*, *or*, *not*, $+$, $*$, etc.). Constants (e.g., 42, 3.14, and *true*) are interpreted as constant functions over discrete time.

```

module SPEEDOMETER:
input sec, cm;
output speed : combine double with +;
[
loop
  var cpt := 0.0 : double in
    abort
      loop await cm ; cpt := cpt + 1.0 end loop
    when sec do
      if (cpt > 0.0) then % null speed is no longer relevant
        emit speed(cpt)
      end if
    end abort
  end var
end loop
||
loop
  var cpt := 0.0 : double in
    abort
      loop await sec ; cpt := cpt + 1.0 end loop
    when cm do
      if (cpt > 0.0) then % null speed is no longer relevant
        emit speed(1.0/cpt)
      end if
    end abort
  end var
end loop
].

```

FIGURE 14.3 Second version of the speedometer in Esterel.

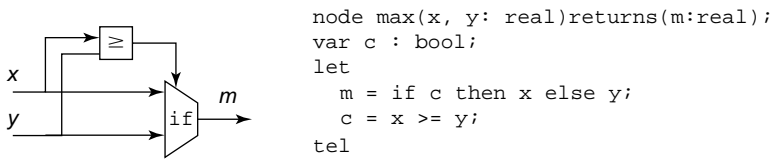


FIGURE 14.4 A dataflow graph and the corresponding Lustre program.

With this subset, one can write combinational programs, which are simply scalar functions extended pointwise on flows. To actually operate on flows, the language provides the `pre` operator:

$$(\text{pre } X)(1) = \perp \quad (\text{pre } X)(t) = X(t-1) \quad \forall t > 1$$

This operator is similar to a noninitialized memory: its value is initially undefined (represented by \perp), and then, forever, it holds the previous value of its argument.

Some mechanism is necessary to properly initialize flows. This is achieved by the arrow operator:

$$(X \rightarrow Y)(1) = X(1) \quad (X \rightarrow Y)(t) = Y(t) \quad \forall t > 1$$

By combining `pre` and `arrow`, one can define recursive flows. For instance, the alternating Boolean flow

$$\text{alt} = \text{false} \rightarrow \text{not pre alt}$$

denotes false, true, false, true, ...; the counter:

$$i = 0 \rightarrow \text{pre } i + 1$$

denotes the sequence of integers 0, 1, 2, 3, ...

A simple example. In the dataflow paradigm, it is natural to represent events as infinite Boolean flows according to the equivalence $\text{present} = \text{true}/\text{absent} = \text{false}$. The program in Figure 14.5 counts the occurrences of its input x since the last occurrence of reset , or the beginning of time if reset has never occurred.

The speedometer in Lustre. Lustre is modular: user-defined nodes can be reused to program more and more complex systems. The syntax is functional-like and the semantics is simple; instantiating a node is equivalent to inlining the definition of the node (provided local variables are renamed with fresh identifiers). Note that this does not mean that inlining is necessary for code generation; under some restrictions the code generation can be modular and thus, reflect the structure of the source code (see Section 14.4).

For instance, the node `counter` can be reused to program the speedometer. Just like in the Esterel program (Figure 14.3), we use two counters running concurrently. The results of the counters are used to compute the fast speed (`sp1`) and the slow speed (`sp2`). By construction, at most one of these speeds is relevant (which means ≥ 0), as a consequence, the output can be obtained by computing the maximum value out of `sp1` and `sp2` (Figure 14.6).

14.4.3 Causality Checking

Among classical static checks (type checking, references etc.), the compilation of synchronous languages requires to check causality. As explained in Section 14.3.3, the logical simultaneity of causes and consequences may introduce logical loops. This is the case

- In Esterel, when the status of a signal depends on itself, e.g., `present X then emit X` and
- In Lustre, when the definition of a flow is instantaneously recursive, e.g., `X = X or Y`

```
node counter(x, reset: bool) returns (c: int);
var lc : int;
let
  c = lc + (if x then 1 else 0);
  lc = if (true -> pre reset) then 0 else pre c;
tel
```

FIGURE 14.5 A simple counter with delayed reset in Lustre.

```
node speedometer(sec, cm: bool) returns (speed: real);
var
  cpt1, cpt2 : int;
  sp1, sp2 : real;
let
  cpt1 = counter(cm, sec);
  sp1 = if sec then real(cpt1) else 0.0;
  cpt2 = counter(sec, cm);
  sp2 = if (cm and (cpt2 > 0)) then 1.0/(real(cpt2)) else 0.0;
  speed = max(sp1, sp2);
tel
```

FIGURE 14.6 A speedometer in Lustre.

Whatever the language, the problem is in fact similar; instantaneous dependencies may contain loops that must be checked in some manner. We present here three solutions adopted in actual compilers.

Syntactic reject. This is somehow the most strict approach, it consists in statically rejecting any program containing a recursion in the instantaneous dependencies relation. This solution is adopted in Lustre (and SCADE), and the criterion is purely syntactic. For instance, the following program is rejected, even if it is impossible to have a causality loop at runtime (if condition C is true then X depends on Y , if C is false then Y depends on X):

$$\begin{aligned} X &= \text{if } C \text{ then } f(Y) \text{ else } T; \\ Y &= \text{if } C \text{ then } U \text{ else } g(X); \end{aligned}$$

This choice in Lustre is purely pragmatic; it is widely accepted that combinational loops in dataflow descriptions are error-prone and should be avoided even in the case where the loop can actually be solved by some reasoning.

Boolean causality. This solution is the opposite to the previous one in the sense that it aims at accepting any “safe” program, regardless to syntactic loops. Intuitively, this analysis accepts any program that can be proved to be both *reactive* and *deterministic*, that is, to have exactly one possible reaction (pair of next state and outputs), for each set of inputs, no matter what the current state is.

More precisely, instantaneous dependencies are represented by a system of logical equations of the form $V = F(V, I)$, where V is the local and output variables and I the inputs. The system is considered as correct if, for any valuation of I , the system admits a unique solution for V . For instance, supposing that $V = \{x, y\}$, $I = \{c\}$, and f, g are Boolean functions, the system:

$$\begin{aligned} x &= c \wedge f(y) \\ y &= \neg c \wedge g(x) \end{aligned}$$

admits a unique solution for both $c = 1$ ($x = f(0)$, $y = 0$) and $c = 0$ ($x = 0$, $y = g(0)$), and thus, it is accepted.

This solution rejects programs that clearly make no sense. For instance, consider an Esterel program where the only emission of X appears in the statement `present X else emit X`. Instantaneous dependencies give the equations $X = \neg X$, which has no logical solution; the program is not *reactive* and thus, it is rejected.

Consider now another example where the unique emission of X appears in “`present [X or A] then emit X end.`” The underlying logical equation is then $X = X \vee A$, which has a unique solution if $A = 1$, but two solutions when $A = 0$. The corresponding program is then rejected as *nondeterministic*.

Boolean causality has been adopted in Argos and in early versions of the Esterel compiler. However, it suffers from several drawbacks:

- It accepts “dubious” programs. For instance, consider the system ($X = X$, $Y = X \wedge \neg Y$); this system seems to have all the bad characteristics: X is not constrained, while Y depends on its own negation. However, the system admits a unique solution $X = Y = 0$, and thus, the corresponding program is accepted. Note that this system, if implemented as a combinational circuit in the straightforward way, yields an *unstable* circuit, that is, a circuit with races.
- Boolean causality requires satisfiability checking, which is NP-complete. As a consequence, the static check is likely to become intractable as the number of variables grows.

Constructive causality. This solution has been proposed [22,23] to circumvent the problems of Boolean causality, which means to reject the doubtful programs, and to allow a (much) more efficient static check. The idea is to solve recursive equations of the form $V = F(I, V)$ according to constructive (or *intuitionistic*) logic rather than classical Boolean logic.

Roughly speaking, constructive logic is classical logic without the *tertium non datur* principle (i.e., without the axiom $\forall x, x \vee \neg x$). In constructive logic, the system ($X = X$, $Y = X \wedge \neg Y$) has no solution, but $x = c \wedge f(y)$, $y = \neg c \wedge g(x)$ still admits a unique solution (as a function of c): $x = c \wedge f(0)$, $y = \neg c \wedge g(0)$.

Concretely, constructive logic behaves as a propagation of facts in the four-values lattice $\{\top, 0, 1, \perp\}$, where \top is the erroneous value (both true and false), and \perp the unknown value (either true or false). If, during the evaluation, some variable becomes \top , the system has no solution. If the evaluation stops while a variable is still \perp , the system has several solutions.

Solving closed systems in constructive logics is similar to partial evaluation, and thus, has a polynomial cost. The complexity still grows exponentially by adding free variables such as inputs (the system must be solved for any valuation of free variables). However, efficient symbolic algorithms exist [23], and, in practice, constructive causality is much more efficient than Boolean causality.

We have shown that constructive causality rejects programs that are obviously wrong, or at least dubious. But what exactly are the right programs? and does the method accept them all? G. Berry [22] pointed out a natural argument in favor of constructive causality by stating that it reflects what actually happens in a digital circuit; recursive systems that are constructively correct are those which, when mapped to hardware, give circuits that eventually stabilize.

14.4.4 Code Generation

The importance of this topic is due to the very peculiar position of synchronous languages, somewhere between modeling tools and programming languages. According to which definition is chosen, one would address the topic as “compilation” or as “program synthesis.” The truth is somewhere in-between: code generation for synchronous languages is in general harder than usual compilation but still easier than program synthesis. It should be noted that code generation is still met with resistance; in many cases, even if people use synchronous modeling tools (e.g., Simulink/Stateflow), they still prefer to manually recode the models, mainly for efficiency reasons, related either to performance or code length or memory width. This situation reminds of the old times of assembly versus high-level languages. There is little doubt that, sooner or later, code generation from high-level models will become mainstream. This is why this question is addressed here.

We only consider here the basic problem, which is the generation of purely sequential code, suitable for a single-task, monoprocessor implementation, as shown in Table 14.1.

Synchronous compilers do not actually produce the full program, they identify the necessary memory (and its initial value) and produce a procedure implementing a single step of execution (the `compute outputs and states` in Table 14.1). In other terms, compilers only provide the functionality of the system, and some main loop program should be written to define the actual execution rate (e.g., event-driven or periodically sampled, depending on the application).

14.4.4.1 From Dataflow to Sequential Code

Consider the example of the counter (Figure 14.5). Obtaining sequential code from the set of Lustre equations is rather simple. It requires (1) to introduce variables for implementing the `pre` operators (in the example `pre_reset`, `pre_c`) and (2) to sort equations to respect data dependencies. Note that a suitable order exists as soon as the program has been accepted by the causality checking (in the example, `lc` must be computed before `c`).

Following those principles, the target code is a simple sequence of assignments, as shown in the left of Figure 14.7. The main advantage of this somehow naive algorithm is that it produces a code, which is neither better nor worse than the source code: both the size and the execution time are linear with respect to the size of the source code. This one-to-one correspondence between source and target code is particularly appreciated in critical domains such as avionics, and it has been adopted by the SCADE compiler.

However, some optimizations can be made:

- To minimize the number of buffers (in the example, `pre_c` is redundant)
- To speed up execution by building a control structure

An optimized code is shown in the right of Figure 14.7.

<pre> bool pre_reset = true; int pre_c = 0, c; // c: output bool x, reset; // inputs void counter_step() { int lc; lc = (pre_reset)? 0 : pre_c; c = lc + (x)? 1 : 0; pre_c = c; pre_reset = reset; } </pre>	<pre> bool pre_reset = true; int c; bool x, reset; void counter_step() { if (pre_reset) c = 0; if (x) c++; pre_reset = reset; } </pre>
---	---

FIGURE 14.7 Simple and optimized C code for the Lustre counter.

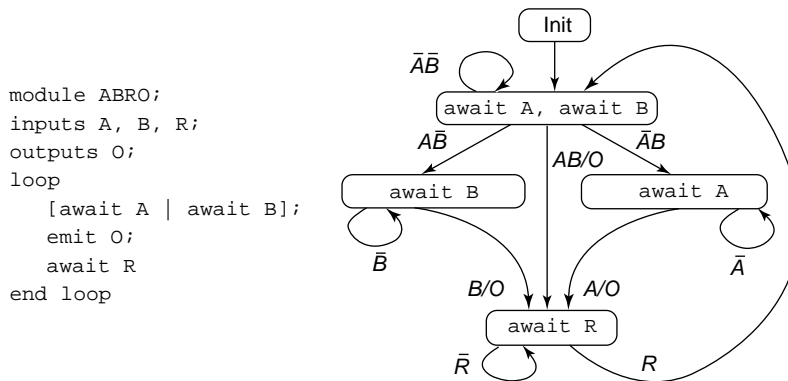


FIGURE 14.8 The ABRO system and its automaton.

14.4.4.2 Explicit Automaton

For imperative languages like Esterel, and even more for languages based on concurrent Mealy machines, it may appear a good idea to generate a code whose structure is an explicit automaton. The principle of automaton generation is illustrated in Figure 14.8: the program ABRO awaits in parallel one A and one B, then it emits Output, and waits for a Reset. For building the automaton, it is necessary to first identify the *halt-points*: halt-points are the statements where the logical time passes (the *await* statements in the example). A state of the automaton corresponds to a configuration of halt-points. Each transition is labeled by a condition on input signals (\bar{A} means A is absent).

Producing a “next” procedure from such an automaton is trivial. Figure 14.9 shows a possible C code: the procedure is a switch on the enumerated variable implementing the current state. In this example, inputs/outputs are performed using procedures that are not detailed.

Compilation into an explicit automaton produces code, which is very efficient in terms of execution time; a typical call of the next procedure only requires a few tests and assignments. The drawback is indeed the size of the code: flattening a hierarchical, parallel composition of Mealy machines may result in code exponentially larger than the source program. As a consequence, this compilation scheme was soon abandoned for methods providing a more realistic compromise between code size and execution time.

14.4.4.3 Implicit Automaton

Several techniques for generating efficient and compact code for Esterel have been proposed [24–27]. All these methods are far too sophisticated to be presented in a few lines. However, they all share the same characteristic, namely, that the state is encoded using several Boolean variables instead of a single

```

enum AbroState = {Init, awaitAB, awaitA, awaitB, awaitR};
static AbroState state;
void AbroNext(){
    switch(state){
        case Init: state = awaitAB; break;
        case awaitAB:
            if (presentA()) {
                if (presentB()) { emitO(); state = awaitR; }
                else state = awaitB;
            } else if (presentB()) state = awaitA; }
        break;
        case awaitA:
            if (presentA()) { emitO(); state = awaitR; }
        break;
        case awaitB: if (presentB()) { emitO(); state = awaitR; } break;
        case awaitR: if (presentR()) { state = awaitAB; } break;
    }
}

```

FIGURE 14.9 A typical C code for the ABRO automaton.

```

next_Init = false
next_awaitA = Init or (awaitA and not A) or (awaitR and R)
next_awaitB = Init or (awaitB and not B) or (awaitR and R)
next_awaitR = not (next_awaitA or next_awaitB)
0 = next_awaitR and not awaitR

```

FIGURE 14.10 Implicit automaton of ABRO, as a set of equations.

enumerated variable. Note that an automaton with n states can be encoded using $\log_2(n)$ Boolean variables. It is a bad idea to first generate the explicit automaton and then try to encode it with Booleans. However, a trivial encoding is obtained directly from the source program by taking one Boolean variable for each control point.

Consider the ABRO example. We first introduce four Boolean variables, one for each halt-point (Init, awaitA, awaitB, awaitR). Initially, all these variables are false except Init. Then, the next values of these variables are defined by a set of equations obtained by analyzing the program, as shown in Figure 14.10. Finally, we have obtained an implicit automaton, which is equivalent to a sequential circuit, or, equivalently, a Lustre program.

One can note that, while the memory required is now clearly linear with respect to the source size, the number of required operations seems to have grown dramatically. Thus, some work remains to actually obtain concise and efficient code, for instance

- Identify common subexpressions, to compute things once
- Embed the whole computation step into a local control structure to avoid useless computation
- Or even, try to build a new encoding giving a better compromise between the number of state variables and the size of code

Figure 14.11 shows a possible optimized code for the ABRO system.

In practice, efficient Esterel compilers [25,28] are able to produce code whose size is reasonable (most of the time linear, quadratic in the worst case) and whose execution time is similar to what can be obtained by “handmade” code.


```

if (Init) {
    awaitA = awaitB = 1; Init = 0;
} else if (awaitR) {
    awaitA = awaitB = R;
    awaitR = !R;
} else {
    awaitA = (awaitA && !A);
    awaitB = (awaitB && !B);
    if (!awaitA and !awaitB) {
        awaitR = 1; emitO();
    }
}

```

FIGURE 14.11 Optimized C code for ABRO.

14.5 Back to Practice

We have seen several examples of synchronous languages. We have also seen how a *single-processor* and *single-task* (or, equivalently, *single-process* or *single-thread*) implementation can be automatically generated from a synchronous program. Such an implementation is simple, it consists of initialization code followed by a read–compute–write loop triggered by some external event (clock “tick” or other). Despite its advantages, discussed above, this “classical” implementation method also has limitations, as discussed in Section 14.2.3 above: it is not well-suited for multirate applications (multiperiodic or mixed time- and event-triggered) and it is not applicable to implementations on distributed execution platforms.

Multirate applications and distributed implementations are common in industrial practice. To deal with these cases, new methods have been developed, which allow to implement synchronous programs under various software or hardware architectures, for instance, involving many tasks or many processors connected with a *bus*. We review some of these methods in the subsections that follow.

14.5.1 Single-Processor, Preemptive Multitasking Implementations

In this subsection, we will explain how synchronous programs can be implemented on a single processor that is equipped with a *real-time operating system* (RTOS) employing some type of *preemptive scheduling* policy, such as *static priority* or *earliest deadline first* (EDF). In such a case, we can generate a *multitask* instead of a single-task implementation. This means that there will be many tasks, each corresponding to a part of the synchronous program. The RTOS will schedule the tasks for execution on the processor.

To justify the interest behind multitask implementations, let us provide a simple example. Consider a synchronous program consisting of two parts, or tasks, P_1 and P_2 , that must be executed every 10 and 100 ms, respectively. Suppose the WCET of P_1 and P_2 is 2 and 10 ms, respectively, as shown in Figure 14.12. Then, generating a single task P that includes the code of both P_1 and P_2 would be problematic. P would have to be executed every 10 ms, since this is required by P_1 . Inside P , P_2 would be executed only once every 10 times (e.g., using an internal counter modulo 10). Assuming that the WCET of P is the sum of the WCETs of P_1 and P_2 (note that this is not always the case), we find that the WCET of P is 12 ms, that is, greater than its period. In practice, this means that every 10 times, the task P_1 will be delayed by 2 ms. This may appear harmless in this simple case, but the delays might be larger and much less predictable in more complicated cases.

Until recently, there has been no rigorous methodology for handling this problem. In the absence of such a methodology, industrial practice consists in “manually” modifying the synchronous design, for instance, by “splitting” tasks with large execution times, like task P_2 above. Clearly, this is not satisfactory as it is both tedious and error-prone.

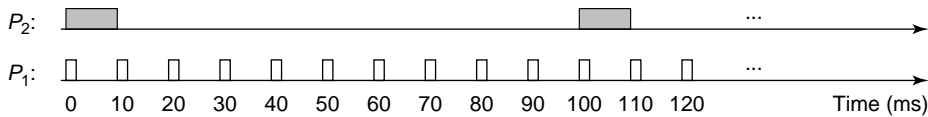


FIGURE 14.12 Two periodic tasks.

When an RTOS is available, and the two tasks P_1 and P_2 do not communicate with each other (i.e., do not exchange data in the original synchronous design), there is an obvious solution: generate code for two *separate* tasks, and let the RTOS handle the scheduling of the two tasks. Depending on the scheduling policy used, some parameters need to be defined. For instance, if the RTOS uses a static-scheduling policy, as this is the case most of the time, then a priority must be assigned to each task prior to execution. During execution, the highest-priority task among the tasks that are ready to execute is chosen. In the case of *multiperiodic* tasks, as in the example above, the *rate-monotonic* assignment policy is known to be optimal in the sense of *schedulability* [4]. This policy consists in assigning the highest priority to the task with the highest rate (i.e., smallest period), the second highest priority to the task with the second highest rate, and so on.

This solution is simple and works correctly as long as the tasks do not communicate with each other. However, this is not a common case in practice. Typically, there will be data exchange between tasks of different periods. In such a case, some intertask communication mechanism must be used. On a single processor, this mechanism usually involves some form of *buffering*, that is, shared memory accessed by one or more tasks. Different buffering mechanisms exist:

- Simple ones, such as a buffer for each pair of writer/reader tasks, equipped with a *locking* mechanism to avoid corruption of data because of simultaneous reads and writes;
- Same as above but also equipped with a more sophisticated protocol, such as a *priority inheritance* protocol to avoid the phenomenon of *priority inversion* [2], or a *lock-free* protocol to avoid blocking upon reads or writes [29,30]; and
- Other shared-memory schemes, like the *publish-subscribe* scheme used in the PATH project [31,32], which allows *decoupling* of writers and readers.

None of these buffering schemes, however, guarantees preservation of the original synchronous semantics.* This means that the sequence of outputs produced by some task at the implementation may not be the same as the sequence of outputs produced by the same task at the original synchronous program. Small discrepancies between semantics and implementation can sometimes be tolerated, for instance, when the task implements a *robust* controller, which has built-in mechanisms to compensate for errors. In other applications, however, such discrepancies may result in totally wrong results with catastrophic consequences. Having a method that guarantees equivalence of semantics and implementation is then crucial. It also implies that the effort spent in simulation or verification of the synchronous program need not be duplicated for the implementation. This is an extremely important cost factor.

To show why preservation of synchronous semantics is not generally guaranteed, consider the example shown in Figure 14.13. The timeline on the top of the figure shows the arrivals (releases) of three tasks: τ_i , τ_j , and τ_q . Task τ_j (the *reader*) reads the output produced by τ_i (the *writer*). τ_q is a third task not communicating with the other two, its role will become clear in what follows. Task τ_i is released at times r_k^i and r_{k+1}^i , and produces outputs y_k^i and y_{k+1}^i , respectively. Task τ_j is released at time r_m^j and reads from τ_i a value x_m^j . According to the synchronous semantics, x_m^j must be equal to y_k^i , since all tasks execute in *zero time*.

*Many of these schemes guarantee *freshest-value* semantics, where the reader always gets the latest value produced by the writer. Freshness is desirable in some cases, in particular in control applications, where the more recent the data, the more accurate they are.

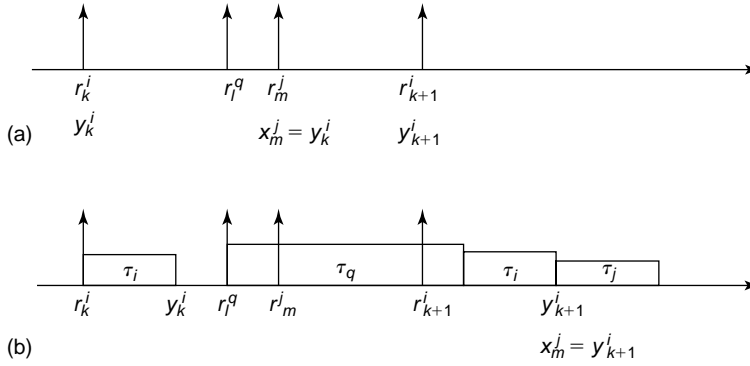


FIGURE 14.13 In the semantics, $x_m^j = y_k^i$, whereas in the implementation, $x_m^j = y_{k+1}^i$. (a) Ideal semantics, (b) Simple implementation.

Setting: A high-priority writer task τ_i communicates data to a low-priority reader task τ_j . Task τ_j maintains a double buffer $B[0, 1]$ and two Boolean pointers *current*, *next*. Initially, *current* = *next* = 0 and $B[0] = B[1] = \text{some default value}$. During execution:

- When τ_i is released: if *current* = *next*, then *next* := not *next*.
- While τ_i executes it writes to $B[\text{next}]$.
- When τ_j is released: *current* := *next*.
- While τ_j executes it reads from $B[\text{current}]$.

FIGURE 14.14 High-to-low buffering protocol.

The timeline on the bottom of Figure 14.13 shows a possible behavior of a simple implementation of the three tasks. We suppose that each task is executed as a separate process on an RTOS that uses static-priority preemptive scheduling. We suppose that task τ_q has higher priority than τ_i and τ_i has higher priority than τ_j . The execution periods of the different tasks are represented by the “boxes” shown in the figure. It can be seen that task τ_q , because of its higher priority, continues to execute when τ_j and τ_i are released. In that way, it “masks” the order of arrivals of τ_i and τ_j , which results in an inverse execution order: τ_i executes before τ_j , because it has higher priority. As a result, task τ_j reads the wrong output of τ_i : y_{k+1}^i instead of y_k^i .

In the rest of this subsection, we will present a novel implementation method, which permits to systematically build multitask implementations of synchronous programs while preserving the synchronous semantics. This method has been developed in a number of recent works [33–35]. We only sketch the main ideas of the method here, and refer the reader to the above publications for details.

The method consists in a set of buffering schemes that mediate data between writer and reader tasks. Each of these schemes involves a set of *buffers* shared by the writer and the readers, a set of *pointers* pointing to these buffers, and a *protocol* to manipulate buffers and pointers. The essential idea behind all protocols is that *the pointers must be manipulated not during the execution of the tasks, but upon the arrival of the events triggering these tasks*. In that way, the order of arrivals can be “memorized” and the original semantics can be preserved.

Let us describe one of these protocols, called the *high-to-low buffering protocol* (described in Figure 14.14). This protocol is used when the writer has higher priority than the reader (for simplicity we assume a single reader, but the protocol can be extended to any number of readers, with the addition of extra buffers, see Ref. 35). In this protocol, the reader τ_j maintains a double buffer $B[0, 1]$. The reader

also maintains two Boolean variables `current`, `next`. `current` points to the buffer currently being read by τ_j and `next` points to the buffer that the writer must use when it arrives next, in case it preempts the reader. The two buffers are initialized to a default value (which the reader reads if it is released before any release of the writer) and both bits are initialized to 0.

When the reader task is released, it copies `next` into `current`, and reads from `B[current]` during its execution. When the writer task is released, it checks whether `current` is equal to `next`. If they are equal, then a reader might still be reading from `B[current]`, therefore, the writer must write to the other buffer, in order not to corrupt this value. Thus, the writer toggles the `next` bit in this case. If `current` and `next` are not equal, then this means that one or more instances of the writer have been released before any reader was released, thus, the same buffer can be reused. During execution, the writer writes to `B[next]`.

Let us see how this protocol copes with the problem illustrated in Figure 14.13. Suppose that the release of the writer at time r_k^i is the first release. Then, since at that time we have `current` = `next` = 0, `next` will be toggled to 1. Upon release of the reader at time r_m^j , `current` is set to `next`, that is, to 1 also. Upon the next release of the writer at time r_{k+1}^i , `next` is toggled to 0. Thus, when the writer executes after the end of task τ_q , it writes to `B[next] = B[0]`. When the reader executes after the end of the writer, it reads from `B[current] = B[1]`, which holds the correct value.

14.5.2 Distributed Implementations

In this subsection, we will explain how synchronous programs can be implemented on distributed architectures, consisting of a number of computers connected via some type of network. Most embedded applications today involve such distributed architectures. For instance, high-end cars today may contain up to 70 electronic control units connected by many different busses. The implementation of a synchronous program consists of one or more tasks per computer, plus extra “glue code” that handles intertask communication. In the rest of the section, we present two distributed implementation methods.

14.5.2.1 Implementation on the Time-Triggered Architecture

The time-triggered architecture [36] or TTA is a distributed, synchronous, fault-tolerant architecture. It includes a set of computers (or *nodes*) connected via a *synchronous bus*. The bus is synchronous in the sense that it implements a clock-synchronization protocol. This implies a global time frame for all nodes. Access to the bus is done using a statically defined *time-triggered* schedule: each node is allowed to transmit during a specified time slot. The schedule repeats periodically as shown in Figure 14.15. The schedule ensures that no two nodes transmit at the same time (provided their clocks are synchronized, which is the responsibility of the TTA bus controller). Therefore, no online arbitration is required, contrary to the case of the CAN bus [37]. A set of tasks is executing at each TTA node. The latter is equipped with the *OSEKtime* operating system [38], which allows tasks to be also scheduled in a time-triggered manner.

Generating a TTA implementation from a synchronous program means the following:

- *Decomposing* the synchronous program into a set of tasks.
- *Assigning* each task to a TTA node where it is going to execute.
- *Scheduling* the tasks on each node and the messages on the TTA bus.
- *Generating* code for each task and glue code for the messages.

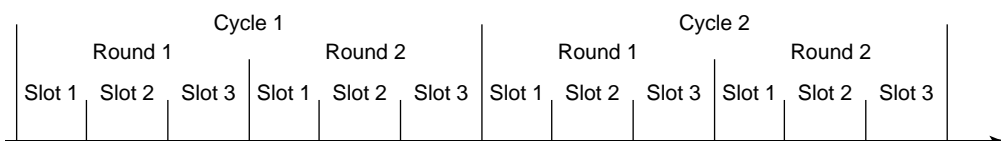


FIGURE 14.15 TTA bus schedule.

Notice that the first two steps are not particular to an implementation on TTA. Any distributed implementation of a synchronous program requires at least these two steps. The scheduling steps are specific to TTA. However, implementations on other types of platforms will probably involve similar steps. For instance, instead of producing a time-triggered schedule of tasks and messages, priorities might need to be assigned to tasks (e.g., when the RTOS uses static-priority preemptive scheduling) or messages (e.g., when the bus is CAN).

In theory, the first three steps above can be formulated in terms of an optimization problem that can be solved automatically. In practice, the degrees of freedom (unknowns of the optimization problem) are far too many for the problem to be tractable (the problem is NP-hard). Thus, some choices must be made by the human user of the method. Fortunately, in practice the user has often a good intuition about many of these choices. For instance, the decomposition of the program into tasks and the assignment of tasks to processors is often dictated by the application and topology constraints (e.g., a task sampling a sensor must be executed on the computer directly connected to the sensor).

In Ref. 39, a method is proposed to generate TTA implementations from Lustre programs. In fact, the method uses a version of Lustre extended with annotations that allow the user to specify total or partial choices on the decomposition and assignment steps, as well as real-time assumptions (on WCETs) and requirements (deadlines). The method uses a *branch-and-bound* algorithm to solve the scheduling problem. Decomposition is handled using the following strategy: start with a *coarse* decomposition; if this fails to produce a valid schedule then *refine* some of the tasks, that is, “split” them into more than one task. Which tasks should be refined is determined based on feedback from the scheduler and on heuristics (e.g., task with the largest WCET and task with longest blocking time). We refer the reader to the above paper for details.

14.5.2.2 Static Scheduling of Synchronous Dataflow Graphs on Multiprocessors

An older method for producing distributed implementations of synchronous designs is the one proposed in Refs. 40 and 41. This method concerns the *synchronous dataflow* model (SDF). SDF can be viewed as a subclass of synchronous languages such as Lustre in the sense that only multiperiodic designs can be described in SDF. In contrast, SDF descriptions are more “compact” and must generally be “unfolded” before being transformed into a synchronous program. Algorithms to perform this unfolding are provided in Ref. 40.

Before proceeding to review the implementation of SDF, let us give an example of an SDF graph and its translation into a synchronous language. The SDF graph is shown in Figure 14.16. It consists of two nodes A and B. The arrow from A to B denotes that A produces a sequence of *tokens* that are consumed by B. The number 2 in the source of the arrow denotes that A produces two tokens every time it is invoked. The number 1 at the destination of the arrow denotes that B consumes one token every time it is invoked. These numbers imply that, in a periodic execution of A and B, B must be executed twice as many times as A.

In Lustre, the above SDF graph could be described as follows:

```
c = periodic_clock(0, 2);          /* clock 1/2 with phase 0 */
(a1,a2) = A(in when c);           /* A runs at 1/2 and produces two tokens */
(b1,b2) = current (a1,a2);        /* buffer made explicit */
out = B( if c then b1 else b2 );
```

The `periodic_clock(k,p)` macro constructs a Boolean flow corresponding to a clock of period `p` and initial phase `k`.

Let us now turn to the implementation of SDF on multiprocessor architectures proposed in Ref. 40. This consists essentially in constructing a *periodic admissible parallel schedule*, or PAPS. The schedule is

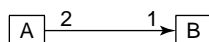


FIGURE 14.16 A simple SDF graph.

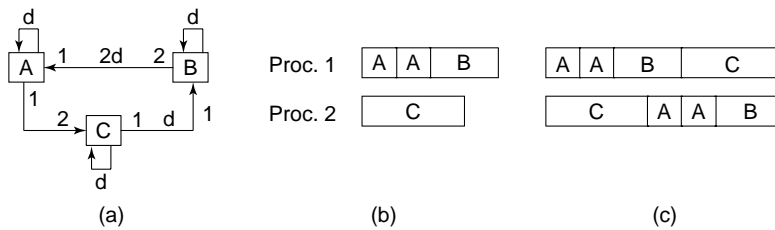


FIGURE 14.17 An SDF graph and two periodic schedules on two processors. (Taken from E. Lee and D. Messerschmitt. *IEEE Transactions on Computers*, 36(1): 24–35, January 1987. With permission.)

admissible in the sense that (1) node precedences are respected, that is, a node cannot execute before all required input tokens are available, (2) execution does not block, and (3) a finite amount of buffering is required to store tokens until they are consumed. The architecture considered in Ref. 40 consists of a number of *homogeneous* computers in the sense that a node may be executed on any computer and it requires the same execution time on any computer. Communication delays between nodes are considered negligible. Finally, some synchronization mechanism is assumed to be available, which allows all processors to resynchronize at the end of a global period to restart the schedule.

Let us provide an example, taken from Ref. 40. Figure 14.17 shows an SDF graph (a), and two PAPS (b) and (c). In the SDF graph there are three nodes A, B, and C, with execution times 1, 2, and 3, respectively. The notation “d” means there is a *unit-delay* in the dataflow link (similarly, “2d” means there are two unit-delays). The schedules are for two processors.

The schedule shown in Figure 14.17b corresponds to a single unfolding of the SDF graph. Notice that A must be executed twice so that the two tokens needed by C are available. Also note that B may start before C finishes, since there is a unit-delay in the link from C to B. Finally, note that the period of this schedule cannot be less than 4; in other words, the rate of this implementation is $\frac{1}{4}$. The schedule shown in Figure 14.17c corresponds to two unfoldings of the SDF graph. This schedule achieves a better rate, namely, $\frac{2}{7}$. It also fully utilizes the two processors (whereas in the previous schedule Processor 2 remains idle for 1 time unit every 4 time units).

It should come as no surprise that constructing an optimal (i.e., rate-maximizing) PAPS is a hard (combinatorial) problem. Ref. 40 proposes a heuristic algorithm based on the so-called level-scheduling algorithm [42].

14.6 Conclusions and Perspectives

Synchronous programming has been founded on the widely adopted practice of simple control-loop programs and on the synchrony hypothesis, which permits to abstract from implementation details by assuming that the program is sufficiently fast to “keep up” with its real-time environment. This spirit is in full accordance with modern thinking about system design, sometimes called model-based design, which advocates the use of high-level models with “ideal,” implementation-independent semantics, and the separation of concerns between design and implementation. New methods that allow the high-level semantics to be preserved by different types of implementations on various execution platforms are constantly being developed and gaining ground in the industry.

A number of challenges remain for the future. To list only a few of them:

- There is currently no good way to describe execution platforms. Although the complete synthesis of semantics-preserving implementations starting from a high-level model (e.g., a synchronous program) plus a formal description of the execution platform seems too far-reaching today (because of the complexity involved), less-ambitious goals such as reconfiguring the code when reconfiguring the platform ultimately require a well-defined description of execution platform or its variations.

- Regarding the preservation of semantics itself, many options are open. Preservation in the “strict” sense such as the one described in Section 14.5 may not always be necessary. For instance, a mixed discrete/continuous controller may require strict preservation of determinism in what concerns control-flow decision points and looser, freshest-value semantics in what concerns robust, continuous control. Finding ways to express the preservation requirements and methods to achieve them are challenging topics for future research.
- UML [43] is fastly moving toward establishing as a standard in embedded system design. Owing to the fact that synchronous programming is also a popular approach, this seems to call for a thorough comparison between these two approaches and, maybe attempts to make them converge.

References

1. K. Åström and B. Wittenmark. *Computer Controlled Systems*. Prentice-Hall, Upper Saddle River, NJ, 1984.
2. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, September 1990.
3. M. Campione, K. Walrath, and A. Huml. *The Java(TM) Tutorial: A Short Course on the Basics*. Sun Microsystems, 3rd edition, Boston, MA, 2001.
4. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1): 46–61, January 1973.
5. N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, 1991.
6. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Upper Saddle River, NJ, 1985.
7. R. Milner. *A Calculus of Communicating Systems*, volume 92 of LNCS. Springer, Berlin, 1980.
8. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3): 267–310, 1983.
9. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2): 87–152, 1992.
10. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9): 1305–1320, September 1991.
11. F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4): 256–266, 1996.
12. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a “safe” subset of Simulink/Stateflow into Lustre. In G. Buttazzo, editor, *4th International Conference on Embedded Software, EMSOFT04*. ACM, Pisa, Italy, 2004.
13. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4): 293–333, 1996.
14. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal: a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2): 362–374, 1986.
15. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proceedings of CONCUR’92*, volume 630 of LNCS. Springer, Berlin, August 1992.
16. G. Berry. The Foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pp. 425–454. MIT Press, Cambridge, MA, 2000.
17. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9): 1321–1336, 1991.
18. B. Dion and J. Gartner. Efficient development of embedded automotive software with IEC 61508 objectives using SCADE Drive. <http://www.esterel-technologies.com/files/Efficient-Development-of-Embedded-SW.pdf>.
19. J. Camus and B. Dion. Efficient development of airborne software with SCADE Suite™, 2003. http://www.esterel-technologies.com/files/SCADE_DO-178B_2003.zip.

20. C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.
21. G. Berry. The Esterel v5 Language Primer. <ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf>.
22. G. Berry. The constructive semantics of Esterel. Draft book available by ftp at <ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps>, 1999.
23. T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.
24. G. Berry. Hardware and software synthesis, optimization, and verification from Esterel programs. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pp. 1–3, Springer-Verlag, London, 1997.
25. D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *CASES'2000*, San Jose, November 2000.
26. S. Edwards. Compiling Esterel into sequential code. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pp. 322–327, June 5–9, 2000. ACM/IEEE New York.
27. S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(2): 169–183, 2002.
28. CEC: The Columbia Esterel Compiler. <http://www1.cs.columbia.edu/~sedwards/cec/>.
29. J. Chen and A. Burns. A three-slot asynchronous reader-writer mechanism for multiprocessor real-time systems. Technical Report YCS-286, Department of Computer Science, University of York, May 1997.
30. H. Huang, P. Pillai, and K. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX'02*, 2002.
31. A. Puri and P. Varaiya. Driving safely in smart cars. In *IEEE American Control Conference*, 1995.
32. S. Tripakis. Description and schedulability analysis of the software architecture of an automated vehicle control system. In *Embedded Software (EMSOFT'02)*, volume 2491 of *LNCS*. Springer, Berlin, 2002.
33. N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro Conference on Real-Time Systems (ECRTS'04)*, 2004.
34. S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or EDF schedulers. In *5th ACM International Conference on Embedded Software (EMSOFT'05)*, pp. 353–360, 2005.
35. C. Sofronis, S. Tripakis, and P. Caspi. A dynamic buffering protocol for preservation of synchronous semantics under preemptive scheduling. Technical Report TR-2006-2, Verimag Technical Report, 2006.
36. H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer, Norwell, MA, 1997.
37. ISO. Road vehicles—Controller area network (CAN)—Part 1: Data link layer and physical signalling. Technical Report ISO 11898, 2003.
38. OSEK/VDX. Time-Triggered Operating System—Version 1.0, 2001.
39. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, 2003.
40. E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1): 24–35, January 1987.
41. E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9): 1235–1245, 1987.
42. T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6): 841–848, 1961.
43. Open Management Group. Response to the OMG RFP for schedulability, performance, and time revised submission. OMG Document number: ad/2001-06-14, June 18, 2001.

III

Operating Systems and Middleware for Real-Time and Embedded Systems

15

QoS-Enabled Component Middleware for Distributed Real-Time and Embedded Systems

Gan Deng

Vanderbilt University

Douglas C. Schmidt

Vanderbilt University

Christopher D. Gill

Washington University

Nanbor Wang

Tech-X Corporation

15.1	Introduction	15-1
15.2	R&D Challenges for DRE Systems	15-2
15.3	Comparison of Middleware Paradigms	15-2
	Conventional DOC Middleware • QoS-Enabled DOC Middleware • Conventional Component Middleware • QoS-Enabled Component Middleware	
15.4	Achieving QoS-Enabled Component Middleware: CIAO, DAnCE, and CoSMIC	15-5
	Integration of Real-Time CORBA and Lightweight CCM Capabilities in CIAO • The Design of DAnCE • Optimizing Static Configuration in DAnCE • Model-Driven Deployment and Configuration with CoSMIC	
15.5	Applications of CIAO, DAnCE, and CoSMIC	15-9
	Avionics Mission Computing Applications • Shipboard Computing Applications • Inventory Tracking Applications • Lessons Learned	
15.6	Related Work	15-13
	QoS-Enabled DOC Middleware • QoS-Enabled Component Middleware	
15.7	Concluding Remarks	15-14

15.1 Introduction

Component middleware technologies are playing an increasingly important role in the development of distributed real-time and embedded (DRE) systems in a variety of application domains ranging from military shipboard computing [1] to commercial inventory tracking [2]. The challenges of designing, implementing, and evaluating component middleware that can meet the needs of such diverse DRE systems have motivated several important advances in the state of the art. This chapter summarizes those

advances and uses examples from several application domains to show how these resulting technologies can be applied to meet the needs of DRE systems.

Section 15.2 surveys R&D challenges for DRE systems. Section 15.3 describes other middleware paradigms for DRE systems to motivate our work on quality of service (QoS)-enabled component middleware for DRE systems. Section 15.4 describes how we developed the *component-integrated ACE ORB* (CIAO) and the *deployment and configuration engine* (DAnCE) QoS-enabled component middleware to address key challenges for DRE systems described in Section 15.2. Section 15.5 describes three application domains in which CIAO has been applied: avionics mission computing [3], shipboard computing [1], and inventory tracking [2]. Section 15.6 surveys related work on middleware for DRE systems and Section 15.7 presents concluding remarks.

15.2 R&D Challenges for DRE Systems

Some of the most challenging R&D problems are those associated with producing software for DRE systems where computer processors control physical devices for sensing and actuation. Examples of such systems include avionics mission computing systems, which help pilots plan and execute airborne missions; shipboard computing systems, which help crews maintain situational awareness of the ship and its surroundings; and inventory tracking system that help to maintain an up-to-date picture of the location and status of parts and products needed by complex and dynamic supply chains. Despite advances in standards-based commercial-off-the-shelf (COTS) middleware technologies, several challenges must be addressed before COTS software can be used to build mission-critical DRE systems effectively and productively. In particular, as DRE systems have increased in scale and complexity over the past decade, a tension has arisen between stringent performance requirements and the ease with which systems can be developed, deployed, and configured to meet those requirements.

DRE systems require different forms of configuration—both at design time and at runtime—to allow customization of reusable components to meet QoS requirements for applications being developed. In addition to being configured individually, components must be assembled to form a complete application and deployed across a set of endsystems upon which the applications will run. Characteristics of the components, their interdependencies when assembled, the endsystems onto which components are deployed, and the networks over which they communicate can vary *statically* (e.g., due to different hardware/software platforms used in a product-line architecture) and *dynamically* (e.g., due to damage, changes in mission modes of the system, or due to differences in the real versus expected behavior of applications during actual operation). When these variabilities are combined with the complex requirements of many DRE systems and the dynamic operating environments in which they operate, it becomes tedious and error-prone to configure operational characteristics of these systems, particularly when components are deployed manually or using middleware technologies that do not provide appropriate configuration support.

Developers of complex DRE systems therefore need middleware technologies that offer (1) explicit configurability of policies and mechanisms for QoS aspects such as priorities, rates of invocation, and other real-time properties, so that developers can meet the stringent requirements of modern DRE systems; (2) a component model that explicitly separates QoS aspects from application functionality so that developers can untangle code that manages QoS and functional aspects, resulting in systems that are less brittle and costly to develop, maintain, and extend; and (3) configuration capabilities that can customize functional and QoS aspects of each DRE system, flexibly and efficiently at different stages of the system life cycle.

15.3 Comparison of Middleware Paradigms

This section examines previous approaches to middleware for DRE systems and explains why only the QoS-enabled component middleware addresses all the challenges described in Section 15.2.

15.3.1 Conventional DOC Middleware

Standards-based COTS middleware technologies for distributed object computing (DOC), such as the Object Management Group (OMG)'s common object request broker architecture (CORBA) [4] and Sun's Java RMI [5], shield application developers from low-level platform details, provide standard higher-level interfaces to manage system resources, and help to amortize system development costs through reusable software frameworks. These technologies significantly reduce the complexity of writing client programs by providing an object-oriented programming model that decouples application-level code from the system-level code that handles sockets, threads, and other network programming mechanisms. Conventional DOC middleware standards, however, have the following limitations for DRE systems [6]:

- *Only functional and not QoS concerns are addressed.* Conventional DOC middleware addresses *functional* aspects, such as how to define and integrate object interfaces and implementations, but does not address crucial QoS aspects, such as how to define and enforce deadlines for method execution. The code that manages these QoS aspects often becomes entangled with application code, making DRE systems brittle and hard to evolve.
- *Lack of functional boundaries.* Application developers must explicitly program the connections among interdependent services and object interfaces, which can make it difficult to reuse objects developed for one application in a different context.
- *Lack of generic server standards.* Server implementations are *ad hoc* and often overly coupled with the objects they support, which further reduce the reusability and flexibility of applications and their components.
- *Lack of deployment and configuration standards.* The absence of a standard way to distribute and start up implementations remotely results in applications that are hard to maintain and even harder to reuse.

15.3.2 QoS-Enabled DOC Middleware

New middleware standards, such as real-time CORBA [7] and the Real-Time Specification for Java [8], have emerged to address the limitations of conventional DOC middleware standards described above. These technologies support explicit configuration of QoS aspects, such as the priorities of threads invoking object methods. However, they do not support generic server environments or component deployment and configuration, which can lead to tangling of application logic with system concerns, similar to the problems seen with conventional DOC middleware.

For example, although real-time CORBA provides mechanisms to configure and control resource allocations of the underlying endsystem to meet real-time requirements, it lacks the flexible higher-level abstractions that component middleware provides to separate real-time policy configurations from application functionality. Manually integrating real-time aspects within real-time CORBA application code is unduly time consuming, tedious, and error-prone [3]. It is hard, therefore, for developers to configure, validate, modify, and evolve complex DRE systems consistently using QoS-enabled DOC middleware.

15.3.3 Conventional Component Middleware

Conventional component middleware technologies, such as the CORBA Component Model (CCM) [9] and Enterprise JavaBeans (EJB) [10], have evolved to address the limitations of conventional DOC middleware in addressing the functional concerns of DRE systems by (1) separating application components so that they interact with each other only through well-defined interfaces, (2) defining standard mechanisms for configuring and executing components within generic containers and component servers that divide system development concerns into separate aspects, such as implementing application functionality versus configuring resource management policies, and (3) providing tools for deploying and configuring assemblies of components.

In CCM, for example, components interact with other components through a limited set of well-defined interface called *ports*. CCM ports include (1) *facets*, which provide named interfaces that service method

invocations from other components; (2) *receptacles*, which provide named connection points to facets provided by other components; (3) *event sources*, which indicate a willingness to send events to other components; and (4) *event sinks*, which indicate a willingness to receive events from other components. Components can also have *attributes* that specify named parameters, which are configurable at various points in an application's life cycle.

Components are implementation entities that can be installed and instantiated independently in standard component server runtime environments stipulated by the CCM specification. For each type of component, a *home* interface provides life-cycle management services. A *container* provides the server runtime environment for component implementations, and handles common concerns such as persistence, event notification, transaction, security, and load balancing.

These component middleware features ease the burden of developing, deploying, and maintaining complex large-scale systems by freeing developers from connecting and configuring numerous distributed subsystems manually. Conventional component middleware, however, is designed more for the needs of enterprise business systems, rather than for the more complex QoS provisioning needs of DRE systems. Developers are therefore often forced to configure and control these mechanisms imperatively within their component implementations.

While it is possible for component developers to embed QoS provisioning code directly within a component implementation, doing so often prematurely commits each implementation to a specific QoS provisioning strategy, making component implementations harder to reuse. Moreover, many QoS capabilities, such as end-to-end provisioning of shared resources and configuring connections between components, cannot be implemented solely within a component because the concerns they address span multiple components.

15.3.4 QoS-Enabled Component Middleware

Owing to the limitations described above, it is necessary to extend standard middleware specifications so that they provide better abstractions for controlling and managing both functional and QoS aspects of DRE systems. In particular, what is needed is *QoS-enabled component middleware* that preserves existing support for heterogeneity in conventional component middleware, yet also provides multiple dimensions of QoS provisioning and enforcement and offers alternative configuration strategies to meet system-specific QoS requirements across the diverse operating environments of DRE systems.

The remainder of this chapter describes how we have extended the standard lightweight CCM specification to support QoS-enabled component middleware that can more effectively compose real-time behaviors into DRE systems and help make it easier to develop, verify, and evolve applications in these systems. Specifically, we cover:

- How real-time CORBA policies can be composed into lightweight CCM applications during their development life cycle, using an XML-based metadata format for describing how real-time policies can be coupled with existing CCM metadata that define application assemblies.
- How the DAnCE framework we developed can translate an XML-metadata specification into the deployment and configuration actions needed by an application, and how static configuration optimizations we developed for DAnCE optimize that capability to support applications with constraints on configuration times or on real-time operating system features.
- How these added capabilities can be used to develop DRE systems via examples from several application domains.
- How extending various CCM metadata to document behavioral characteristics of components and applications can help streamline the optimization of DRE systems.

As we describe in Section 15.4, the vehicles for our R&D activities on QoS-enabled component middleware are (1) the CIAO and the DAnCE, which are based on the OMG's lightweight CORBA component model (lightweight CCM) specification [11]; and (2) the component synthesis of model integrated computing (CoSMIC) [12], which is a model-driven engineering (MDE) [13] tool chain.

15.4 Achieving QoS-Enabled Component Middleware: CIAO, DAnCE, and CoSMIC

This section presents the design of CIAO and DAnCE, which are open-source QoS-enabled component middleware developed at Washington University and Vanderbilt University atop *The ACE ORB* (TAO) [14]. TAO in turn is an open-source real-time CORBA DOC middleware that implements key patterns [15] for DRE systems. CIAO and DAnCE enhance TAO to simplify the development of applications in DRE systems by enabling developers to provision QoS aspects declaratively end to end when deploying and configuring DRE systems using CCM components. We also describe how the CoSMIC MDE tools support the deployment, configuration, and validation of component-based DRE systems developed using CIAO and DAnCE.

15.4.1 Integration of Real-Time CORBA and Lightweight CCM Capabilities in CIAO

CIAO supports lightweight CCM mechanisms that enable the specification, validation, packaging, configuration, and deployment of component assemblies and integrates these mechanisms with TAO’s real-time CORBA features, such as thread pools, lanes, and client-propagated and server-declared policies. The architecture of CIAO is shown in Figure 15.1. CIAO extends the notion of lightweight CCM component assemblies to include both client- and server-side QoS provisioning and enables the declarative configuration of QoS aspects of endsystem ORBs, such as prioritized thread pools [16], via XML metadata. It also allows QoS provisioning at the component-connection level, e.g., to configure thread pools with priority lanes.

To support the composition of real-time behaviors, CIAO allows developers of component-based DRE systems to specify the desired *real-time QoS policies* and associate them with components or component assemblies. For example, application components can be configured to support different priorities and rates of invocation of methods among components. These configurations are declaratively specified via metadata, and once they are deployed, the properties of CIAO’s real-time component server and container runtime environment will be automatically initialized based on the definition of QoS mechanism plug-ins. In particular, this metadata can configure CIAO’s QoS-aware containers, which provide common interfaces for managing components’ QoS policies and for interacting with the mechanisms that enforce those QoS policies.

To support the configuration of QoS aspects in its component servers and containers, CIAO defines a new file format—known as the *CIAO server resource descriptor* (CSR)—and adds it to the set of XML

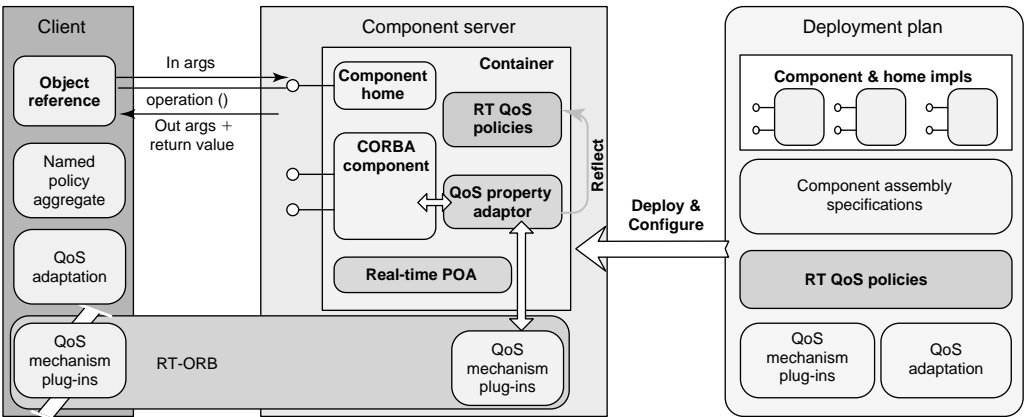


FIGURE 15.1 The architecture of CIAO.

descriptors that can be used to configure an application assembly. A CSR file defines *policy sets* that specify policies for setting key QoS behaviors such as real-time thread priorities and configures resources to enforce them. The resources and policies defined in CIAO's CSR files can be specified for individual component instances. System developers then can use deployment and configuration tools, such as the DAnCE middleware and CoSMIC MDE tools described in next section, to deploy the resulting application assembly onto platforms that support the specified realtime requirements. The middleware and MDE tools can also configure the component runtime infrastructure that will enforce these requirements.

15.4.2 The Design of DAnCE

DAnCE is a middleware framework we developed for CIAO based on the OMG's deployment and configuration (D&C) specification [17], which is part of the lightweight CCM specification [11]. This specification standardizes many deployment and configuration aspects of component-based distributed applications, including component configuration, assembly, and packaging; package configuration and deployment; and resource configuration and management. These aspects are handled via a *data model* and a *runtime model*. The data model can be used to define/generate XML schemas for storing and interchanging metadata that describes component assemblies and their deployment and configuration attributes, such as resource requirements. The runtime model defines a set of services/managers that process the metadata described in the data model to deploy, execute, and control application components in a distributed manner.

DAnCE implements a data model that describes (1) the DRE system component instances to deploy, (2) how the components are connected to form component assemblies, (3) how these components and component assemblies should be initialized, and (4) how middleware services are configured for the component assemblies. In addition, DAnCE implements a spec-compliant runtime model as a set of runtime entities.

The architecture of DAnCE is shown in Figure 15.2. The various entities of DAnCE are implemented as CORBA objects* that collaborate as follows: By default, DAnCE runs an `ExecutionManager` as a daemon to manage the deployment process for one or more *domains*, which are target environments consisting of *nodes*, *interconnects*, *bridges*, and *resources*. An `ExecutionManager` manages a set of `DomainApplicationManagers`, which in turn manage the deployment of components within a single domain. A `DomainApplicationManager` splits a deployment plan into multiple subplans, one for each node in a domain. A `NodeManager` runs as a daemon on each node and manages the deployment of all components that reside on that node, irrespective of the particular application with which they are associated. The `NodeManager` creates the `NodeApplicationManager`, which in turn creates the `NodeApplication` component servers that host application-specific containers and components.

To support additional D&C concerns not addressed by the OMG D&C specification, but which are essential for DRE systems, we enhanced the specification-defined data model by describing additional deployment concerns, which include fine-grained system resource allocation and configuration. To enforce real-time QoS requirements, DAnCE extends the standards-based data model by explicitly defining (1) component server resource configuration as a set of *policies*; and (2) how components bind to these policies, which influence end-to-end QoS behavior.

Figure 15.3 shows the policies for server configurations that can be specified using the DAnCE *server resources XML schema*. With such enhancements, the DAnCE data model allows DRE system developers to configure and control *processor resources* via thread pools, priority mechanisms, intraprocess mutexes, and a global scheduling service; *communication resources* via protocol properties and explicit bindings; and *memory resources* via buffering requests in queues and bounding the size of thread pools.

DAnCE's metadata-driven resource configuration mechanisms help ensure efficient, scalable, and predictable behavior end to end for DRE systems. Likewise, DAnCE helps enhance reuse since component QoS configuration can be performed at a much later phase, i.e., just before the components are deployed into

*The DAnCE deployment infrastructure is implemented as CORBA objects to avoid the circular dependencies that would ensue if it was implemented as components, which would have to be deployed by DAnCE itself!

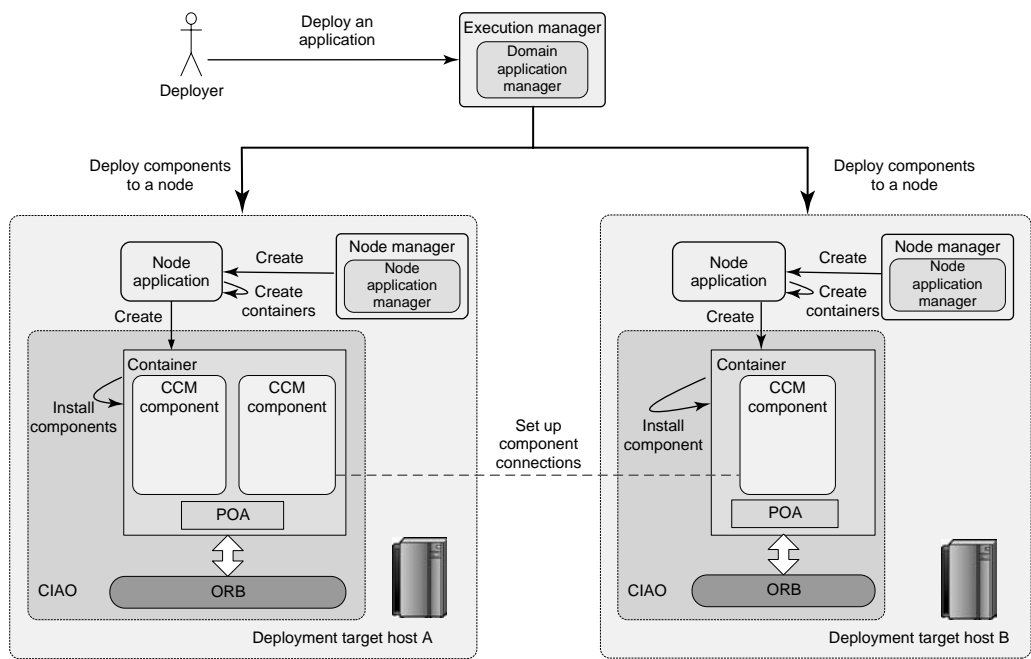


FIGURE 15.2 The architecture of DAnCE.

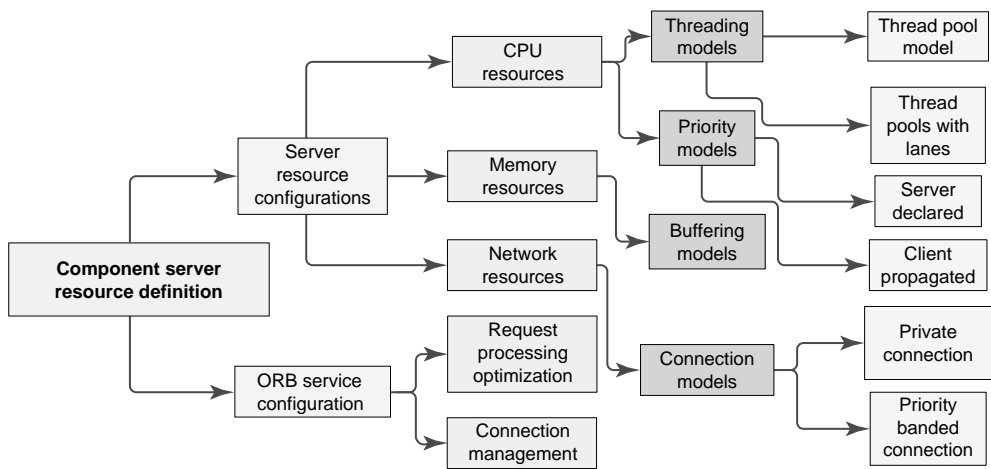


FIGURE 15.3 Specifying RT-QoS requirements.

the target environment. Server resource specifications can be set via the following options: (1) *object request broker (ORB) command line options*, which control TAO's connection management models, protocol selection, and optimized request processing; and (2) *ORB service configuration options*, which specify ORB resource factories that control server concurrency and demultiplexing models. Using this XML schema, a system deployer can specify the designated ORB configurations. The ORB configurations defined by the data model are used to configure *NodeApplication* runtime entities that host components, thereby providing the necessary resources for the components to operate.

To fulfill the QoS requirements defined by the data model, DAnCE extends the standards-based runtime model as follows: An *XMLConfigurationHandler* parses the deployment plan and stores the

information as interface definition language (IDL) data structures that can transfer information between processes efficiently and enables the rest of DAnCE to avoid the runtime overhead of parsing XML files repeatedly. The IDL data structure output of the `XMLConfigurationHandler` is input to the `ExecutionManager`, which propagates the information to the `DomainApplicationManager` and `NodeApplicationManager`. The `NodeApplicationManager` uses the server resource configuration options in the deployment plan to customize the containers in the `NodeApplication` it creates. These containers then use other options in the deployment plan to configure TAO's real-time CORBA support, including thread pool configurations, priority propagation models, and connection models.

15.4.3 Optimizing Static Configuration in DAnCE

To extend our QoS-enabled component middleware to a wider range of applications and platforms, we have implemented static techniques and mechanisms for deployment and configuration where as much of the configuration life cycle as possible is performed offline. This approach is especially important for systems where availability of platform features or stringent constraints on configuration times would otherwise limit our ability to support component deployment and configuration services. The fundamental intuition in understanding our static configuration approach is that stages of the overall DRE system configuration life cycle similar to those in the dynamic approach must still be supported. In our static approach, however, several stages of the life cycle are *compressed* into the compile-time and system-initialization phases so that (1) the set of components in an application can be identified and analyzed before runtime, and (2) the overheads for runtime operation following initialization are minimized and made predictable.

Owing to the nuances of the platforms traditionally used for deploying DRE systems, not all features of conventional platforms are available or usable. In particular, dynamically linked libraries and online XML parsing are often either unavailable or too costly in terms of performance. We, therefore, move several configuration phases earlier in the configuration life cycle. We also ensure that our approach can be realized on highly constrained real-time operating systems, such as VxWorks or LynxOS, by refactoring the configuration mechanisms to use only mechanisms that are available and efficient.

In CIAO's static configuration approach, the same automated build processes that manage compilation of the application and middleware code first insert a code-generation step just before compilation. In this step, CIAO's XML configuration files (`.cdp`, `.csr`) are translated into C++ header files that are then compiled into efficient runtime configuration drivers, which in turn are linked statically with the main application. These runtime drivers are invoked during application (re)initialization at runtime and rapidly complete the remaining online system configuration actions at that point.

With CIAO's static configuration approach, all XML parsing is moved before runtime to the offline build process, and all remaining information and executable code needed to complete configuration is linked statically into the application itself. Each endsystem can then be booted and initialized within a single address space, and there is no need for interprocess communication to create and assemble components.

An important trade-off with this approach is that the allocation of component implementations and configuration information to specific endsystems must be determined in advance. In effect, this trade-off shifts support for runtime flexibility away from the deployment steps and toward the configuration steps of the system life cycle. Our static configuration approach in CIAO thus maintains a reasonable degree of configuration flexibility, while reducing the runtime cost of configuration.

15.4.4 Model-Driven Deployment and Configuration with CoSMIC

Although DAnCE provides mechanisms that make component-based DRE systems more flexible and easier to develop and deploy, other complexities still exist. For example, using XML descriptors to configure the QoS aspects of the system reduces the amount of code written imperatively. XML, however, also introduces new complexities, such as verbose syntax, lack of readability at scale, and a high degree of accidental complexity and fallibility.

To simplify the development of applications based on lightweight CCM and to avoid the complexities of handcrafting XML, we developed CoSMIC, which is an open-source MDE tool chain that supports the

deployment, configuration, and validation of component-based DRE systems. A key capability supported by CoSMIC is the definition and implementation of *domain-specific modeling languages* (DSMLs). DSMLs use concrete and abstract syntax to define the concepts, relationships, and constraints used to express domain entities [18]. One particularly relevant DSML in CoSMIC is the *quality of service policy modeling language* (QoSPML) [19], which configures real-time QoS aspects of components and component assemblies. QoSPML enables graphical manipulation of modeling elements and performs various types of generative actions, such as synthesizing XML-based data model descriptors defined in the OMG D&C specification and the enhanced data model defined in DAnCE. It also enables developers of component-based DRE systems to specify and control the QoS policies via visual models, including models that capture the syntax and semantics of priority propagation model, thread pool model, and connection model. QoSPML's model interpreter then automatically generates the XML configuration files, which allows developers to avoid writing applications that use the convoluted XML descriptors explicitly, while still providing control over QoS aspects.

15.5 Applications of CIAO, DAnCE, and CoSMIC

This section describes our experiences using CIAO, DAnCE, and CoSMIC in several application domains, including avionics mission computing, shipboard computing, and inventory tracking. It also summarizes our lessons learned from these experiences.

15.5.1 Avionics Mission Computing Applications

Avionics mission computing applications [20] have two important characteristics that affect how components are deployed and configured in those systems. First, components are often hosted on embedded single-board computers within an aircraft, which are interconnected via a high-speed backplane such as a VME-bus. Second, interacting components deployed on different aircraft must communicate via a low-and-variable bandwidth wireless communication link, such as Link-16. Figure 15.4 illustrates how image selection, download, and display components may be deployed in an avionics mission computing system.

The timing of the interactions between components within an aircraft is bound by access to the CPU, so that configuring the rates and priorities at which components invoke operations on other components is crucial. Component connections involving receptacles that make prioritized calls to facets can be configured as standard real-time CORBA calls in CIAO, which maps those configuration options directly onto TAO features, as described in Section 15.4.1. Moreover, since components are deployed statically in these systems, the static deployment and configuration techniques described in Section 15.4.3 are applicable to these systems. In fact, they are often necessary due to stringent constraints on system initialization times and limitations on platform features, e.g., the absence of dynamic library support on some real-time operating systems.

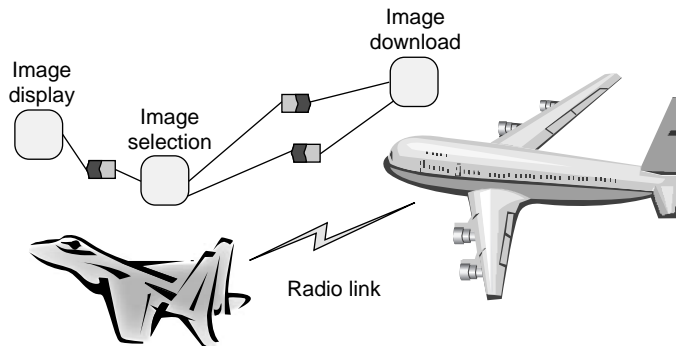


FIGURE 15.4 Component interactions within and between aircraft.

Even within an aircraft, however, it is often appropriate to use more dynamic forms of resource management, such as using hybrid static/dynamic scheduling to add noncritical processing to the system without violating deadlines for critical processing [21]. These more dynamic forms of resource management require configuration of more sophisticated interconnection services between component ports, such as TAO’s real-time event channel [22]. We are currently expanding the configuration options exposed by CIAO to include attributes of the event channel and scheduling services provided by TAO.

Applications whose component interactions span multiple aircraft *mandate* even more dynamic forms of resource management, such as using adaptive scheduling together with adaptive control to manage QoS properties of application data transmission and processing [20,23] across a low-and-variable wireless communication link. We are currently adding configurable adaptive resource management capabilities to CIAO so that system developers can configure QoS properties end to end in component-based DRE systems.

15.5.2 Shipboard Computing Applications

Modern shipboard computing environments consist of a grid of computers that manage many aspects of a ship’s power, navigation, command and control, and tactical operations that support multiple *simultaneous* QoS requirements, such as survivability, predictability, security, and efficient resource utilization. To meet these QoS requirements, *dynamic resource management* (DRM) [24,25] can be applied to optimize and (re)configure the resources available in the system to meet the changing needs of applications at runtime. The primary goal of DRM is to ensure that large-scale DRE systems can adapt dependably in response to dynamically changing conditions (e.g., *evolving multimission priorities*) to ensure that computing and networking resources are best aligned to meet critical mission requirements. A key assumption in DRM technologies is that the levels of service in one dimension can be coordinated with and traded off against the levels of service in other dimensions to meet mission needs, e.g., the security and dependability of message transmission may need to be traded off against latency and predictability.

In The U.S. Defense Advanced Research Projects Agency (DARPA)’s *adaptive and reflective middleware systems* (ARMS) program, DRM technologies were developed and applied to coordinate a computing grid that manages and automates many aspects of shipboard computing. As shown in Figure 15.5, the ARMS DRM architecture integrates resource management and control algorithms supported by QoS-enabled component middleware infrastructure, and is modeled as a layered architecture comprising components at different levels of abstraction, including the DRM services layer, the resource pool layer, and the physical

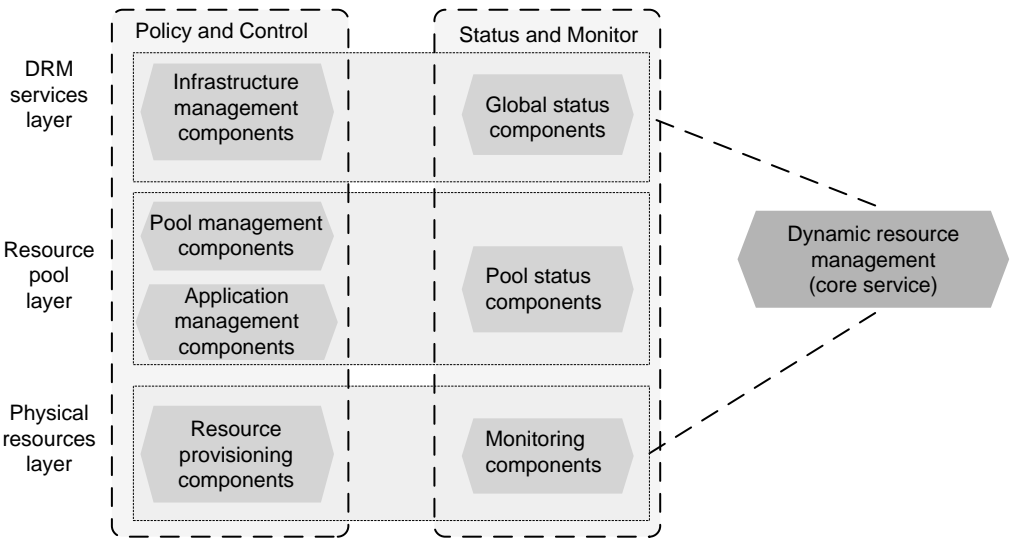


FIGURE 15.5 DRM layered architecture in a shipboard computing environment.

resources layer. The top-level *DRM services layer* is responsible for satisfying global shipboard computing missions, such as mission mode changes. It then decomposes global mission requests into a coordinated set of requests on resource pools within the *resource pool layer*, which is an abstraction for a set of computer nodes managed by a local resource manager. The *physical resources layer* deals with specific instances of resources within a single node in the system.

To simplify development and enhance reusability, all the three layers of the ARMS DRM infrastructure are implemented using CIAO, DANCE, and CoSMIC, which simplifies and automates the (re)deployment and (re)configuration of DRE system components in a shipboard computing environment. The ARMS DRM system has hundreds of different types and instances of infrastructure components written in ~300,000 lines of C++ code and residing in ~750 files developed by different teams from different organizations. ARMS DRM research and experiments [26] show that DRM using standards-based middleware technologies is not only feasible, but can (1) handle dynamic resource allocations across a wide array of configurations and capacities, (2) provide continuous availability for critical functionalities—even in the presence of node and pool failures—through reconfiguration and redeployment, and (3) provide QoS for critical operations even in overload conditions and resource-constrained environments.

15.5.3 Inventory Tracking Applications

An *inventory tracking system (ITS)* is a warehouse management infrastructure that monitors and controls the flow of items and assets within a storage facility. Users of an ITS include couriers (such as UPS, DHL, and Fedex), airport baggage handling systems, and retailers (such as Walmart and Target). An ITS provides mechanisms for managing the storage and movement of items in a timely and reliable manner. For example, an ITS should enable human operators to configure warehouse storage organization criteria, maintain the inventory throughout a highly distributed system (which may span organizational boundaries), and track warehouse assets using decentralized operator consoles. In conjunction with colleagues at Siemens [2], we have developed and deployed an ITS using CIAO, DANCE, and CoSMIC.

Successful ITS deployments must meet both the functional and QoS requirements. Like many other DRE systems, an ITS is assembled from many independently developed reusable components. As shown in Figure 15.6, some ITS components (such as an `OperatorConsole`) expose interfaces to end users, i.e., ITS operators. Other components (such as a `TransportUnit`) represent hardware entities, such as cranes, forklifts, and belts. Database management components (such as `ItemRepository` and `StorageFacility`) expose interfaces to manage external backend databases (such as those tracking item inventories at storage facilities). Finally, the sequences of events within the ITS is coordinated by control-flow components (such as the `WorkflowManager` and `StorageManager`). Our QoS-enabled component middleware and MDE tools allow ITS components to be developed independently.

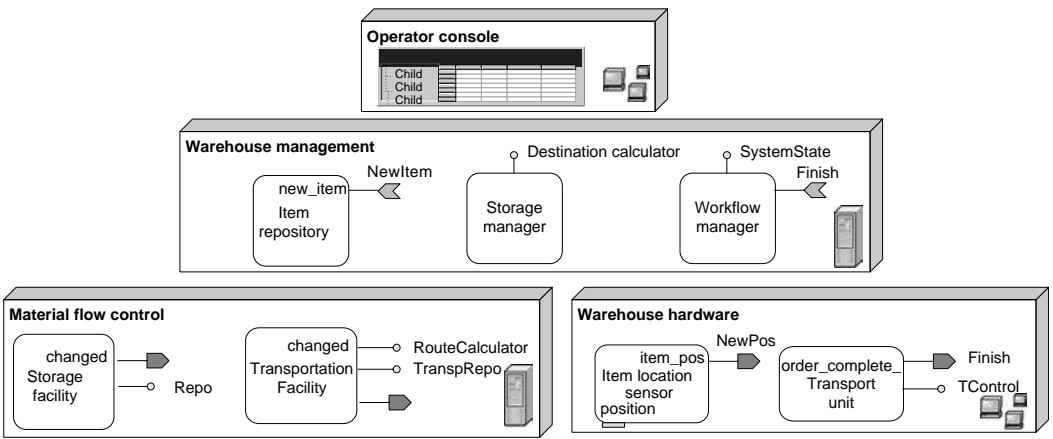


FIGURE 15.6 The architecture of an inventory tracking system.

After individual components are developed, there are still a number of crosscutting concerns that must be addressed when assembling and configuring an ITS, including (1) identifying dependencies between component implementation artifacts, such as the `OperatorConsole` component having dependencies on other ITS components (e.g., a `WorkflowManager` component) and other third-party libraries (e.g., the QT library, which is a cross-platform C++ GUI library compatible with the Embedded Linux OS); (2) specifying the interaction behavior among ITS components; (3) specifying components to configure and control various resources, including processor resources, communication resources, and memory resources; and (4) mapping ITS components and connections to the appropriate nodes and networks in the target environment where the ITS will be deployed. We used the CoSMIC MDE tool chain to simplify the assembly of ITS components and then used DAnCE to deploy and configure them.

QoS requirements (such as latency and jitter) are important considerations of an ITS. For instance, in an ITS, whenever a hardware unit (such as a conveyor belt) fails, the component that controls and monitors this hardware unit must notify another ITS component (such as a `WorkflowManager` component) in realtime to avoid system damage and to minimize overall warehouse delay. Likewise, other components that monitor the location of items, such as an `ItemLocationSensor`, must have stringent QoS requirements since they handle real-time item delivery activities. To satisfy these real-time QoS requirements, the CIAO NodeApplications hosting `ItemLocationSensor` components can be configured declaratively with a server-declared priority model at the highest real-time CORBA priority, with thread pools having preset static threads, and with priority-banded connections.

15.5.4 Lessons Learned

The following are our lessons learned from applying CIAO, DAnCE, and CoSMIC to the three kinds of DRE systems described above:

- *Developing complex DRE systems requires a sophisticated and integrated engineering paradigm.* While modern DRE systems are increasing in size and complexity, creating stove-piped one-of-a-kind applications is unsatisfying, although that has been the state of the practice until recently. Component middleware is designed to enhance the quality and productivity of software developers by elevating the level of abstraction used to develop distributed systems. Conventional component middleware, however, lacks mechanisms to separate QoS aspects from component functional aspects, and it also lacks mechanisms to handle deployment and configuration for such concerns in DRE systems. We therefore designed and integrated CIAO, DAnCE, and CoSMIC based on advanced software principles, such as separation of concerns, metaprogramming, component-based software engineering, and model-driven engineering. We incorporated these concepts as key design principles underlying these technologies and used them to develop representative applications in a variety of DRE system domains.
- *MDE tools alleviate complexities associated with component middleware.* Although component middleware elevates the abstraction level of middleware to enhance software developer quality and productivity, it also introduces new complexities. For example, the OMG lightweight CCM and D&C specifications have a large number of configuration points. To alleviate these complexities, we designed and applied the CoSMIC MDE tools. Our experiences in applying CIAO and DAnCE to various application domains showed that when component deployment plans are incomplete or must change, the effort required is significantly less than using component middleware without MDE tool support since applications can evolve from the existing domain models. Likewise, if the component assemblies are the primary changing concerns in the system, little new application code must be written, yet the complexity of the MDE tool remains manageable due to the limited number of well-defined configuration “hot spots” exposed by the underlying infrastructure.
- *Automated MDE tools are needed to configure and deploy DRE systems effectively.* Despite the fact that the CIAO and DAnCE QoS-enabled component middleware facilitate the configuration of complex DRE systems based on real-time CORBA and lightweight CCM, developers are still faced with the question of what constitutes a “good” configuration. Moreover, they are still ultimately

responsible for determining the appropriate configurations. We observed that scheduling analysis and verification tools would be helpful in performing this evaluation and should be integrated into MDE tool suite to help system designers address such challenges. In addition, despite the benefits of using visual MDE tools to describe different aspects of the large-scale DRE systems, it is still tedious and error-prone to specify the details manually for all the components in large-scale DRE systems. This observation motivates the need for further research in automating the synthesis of large-scale DRE systems based on the different types of metainformation about assembly units, such as high-level design intention or service requirements.

15.6 Related Work

This section compares our work on CIAO and DAnCE with other related work on middleware for DRE systems. In particular, we compare two different categories of middleware for DRE systems: QoS-enabled DOC middleware and QoS-enabled component middleware.

15.6.1 QoS-Enabled DOC Middleware

The *quality objects* (QuO) framework [27,28] uses aspect-oriented software development [29] techniques to separate QoS aspects from application logic in DRE systems. It allows application developers to explicitly declare system QoS characteristics, and then use a specialized compiler to generate code that integrates those characteristics into the system implementation. It therefore provides higher-level QoS abstractions on top of conventional DOC middleware technologies such as CORBA and Java RMI or on top of QoS-enabled DOC middleware such as real-time CORBA.

The *dynamicTAO* [30] project applies reflective techniques to reconfigure ORB components at runtime. Similar to *dynamicTAO*, the *Open ORB* project [31] also aims at highly configurable and dynamically reconfigurable middleware platforms to support applications with dynamic requirements. This approach may not be suitable for some DRE systems, however, since dynamic loading and unloading of ORB components can incur unpredictable overheads and thus prevent the ORB from meeting application deadlines. Our work on CIAO allows MDE tools, such as Cadena [32] and QoS PML [19], to analyze the required ORB components and their configurations and ensure that a component server contains only the required ORB components.

Proactive [33] is a distributed programming model for deploying object-oriented grid applications and is similar to CIAO and DAnCE since it also separately describes the target environment using XML descriptors, but CIAO goes further to allow application to be explicitly composed with a number of components, and CIAO and DAnCE together ensure end-to-end system QoS at deployment time.

15.6.2 QoS-Enabled Component Middleware

The container architecture from conventional component middleware provides a vehicle for applying metaprogramming techniques for QoS assurance control. Containers can also help apply aspect-oriented software development techniques to plug in different systemic behaviors [34]. This approach is similar to CIAO in that it provides mechanisms to inject aspects into applications at the middleware level.

Ref. 35 further develops the state of the art in QoS-enabled containers by extending a QoS EJB container interface to support a `QoSContext` interface that allows the exchange of QoS-related information among component instances. To take advantage of the QoS container, a component must implement `QoSBean` and `QoSNegotiation` interfaces. This requirement, however, adds an unnecessary dependency to component implementations.

The QoS-enabled distributed objects (Qedo) project [36] is another effort to make QoS support an integral part of CCM. Qedo's extensions to the CCM container interface and component implementation framework (CIF) require component implementations to interact with the container QoS interface and negotiate the level of QoS contract directly. While this approach is suitable for certain applications

where QoS is part of the functional requirements, it inevitably tightly couples the QoS provisioning and adaptation behaviors into the component implementation, and thus hampers the reusability of the component. In comparison, CIAO explicitly avoids this coupling and *composes* the QoS aspects into applications declaratively.

The OpenCCM [37] project is a Java-based CCM implementation. The OpenCCM Distributed Computing Infrastructure (DCI) federates a set of distributed services to form a unified distributed deployment domain for CCM applications. OpenCCM and its DCI infrastructure, however, do not support key QoS aspects for DRE systems, including real-time QoS policy configuration and resource management.

15.7 Concluding Remarks

This chapter described how the CIAO and DAnCE QoS-enabled component middleware and CoSMIC MDE tools have been applied to address key challenges when developing DRE systems for various application domains. Reusability and composability are particularly important requirements for developing large-scale DRE systems. With QoS-enabled DOC middleware, such as real-time CORBA and the real-time specification for Java, configurations are made *imperatively* via calls to programmatic configuration interfaces with which the component code is thus tightly coupled. In contrast, QoS-enabled component middleware allows developers to specify configuration choices *declaratively* through metaprogramming techniques (such as XML descriptor files), thus enhancing reusability and composability.

Our future work will focus on (1) developing a computational model to support configuration and dynamic reconfiguration of DRE systems at different levels of granularity to improve both system manageability and reconfiguration performance, and developing a platform model to execute the computational model predictably and scalably; (2) applying specialization techniques (such as partial evaluation and generative programming) to optimize component-based DRE systems using metadata contained in component assemblies; and (3) enhancing DAnCE to provide state synchronization and component recovery support for a fault-tolerant middleware infrastructure, such as MEAD [38].

TAO, CIAO, and DAnCE are available for download at <http://deuce.doc.wustl.edu/Download.html>, and CoSMIC is available for download at <http://www.dre.vanderbilt.edu/cosmic>.

References

1. D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," *CrossTalk—The Journal of Defense Software Engineering*, Nov. 2001.
2. A. Nechypurenko, D. C. Schmidt, T. Lu, G. Deng, and A. Gokhale, "Applying MDA and Component Middleware to Large-Scale Distributed Systems: A Case Study," in *Proceedings of the 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application* (Enschede, Netherlands), IST, Mar. 2004.
3. D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.
4. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Dec. 2002.
5. A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, pp. 265–290, Nov./Dec. 1996.
6. N. Wang, D. C. Schmidt, and C. O’Ryan, "An Overview of the CORBA Component Model," in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, MA: Addison-Wesley, 2000.
7. Object Management Group, *Real-Time CORBA Specification*, OMG Document formal/02-08-02 ed., Aug. 2002.

8. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*, Reading, MA: Addison-Wesley, 2000.
9. Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.
10. Sun Microsystems, "Enterprise JavaBeans Specification." java.sun.com/products/ejb/docs.html, Aug. 2001.
11. Object Management Group, *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., May 2003.
12. A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-Time and Embedded Applications," *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2007 (to appear).
13. D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
14. D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.
15. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, vol. 2. New York: Wiley, 2000.
16. I. Pyrali, D. C. Schmidt, and R. Cytron, "Techniques for Enhancing Real-Time CORBA Quality of Service," *IEEE Proceedings Special Issue on Real-Time Systems*, vol. 91, July 2003.
17. Object Management Group, *Deployment and Configuration Adopted Submission*, Document ptc/03-07-08 ed., July 2003.
18. G. Karsai, J. Sztipanovits, A. Ledeczki, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE*, vol. 91, pp. 145–164, Jan. 2003.
19. S. Paunov, J. Hill, D. C. Schmidt, J. Slaby, and S. Baker, "Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System QoS," in *Proceedings of the 13th Annual International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06)* (Potsdam, Germany), IEEE, Mar. 2006.
20. C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," *Journal of Real-Time Systems*, vol. 29, nos. 2–3, pp. 101–130, 2005.
21. C. Gill, D. C. Schmidt, and R. Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, vol. 91, Jan. 2003.
22. T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," in *Proceedings of OOPSLA '97* (Atlanta, GA), pp. 184–199, ACM, Oct. 1997.
23. X. Wang, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill, "CAMRIT: Control-Based Adaptive Middleware for Real-Time Image Transmission," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (Toronto, Canada), May 2004.
24. L. Welch, B. Shirazi, B. Ravindran, and C. Bruggeman, "DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable, Real-Time Systems," in *Proceedings of the 15th IFAC Workshop on Distributed Computer Control Systems (DCCS'98)*, Sept. 1998.
25. J. Hansen, J. Lehoczky, and R. Rajkumar, "Optimization of Quality of Service in Dynamic Systems," in *Proceedings of the 9th International Workshop on Parallel and Distributed Real-Time Systems*, Apr. 2001.
26. J. Balasubramanian, P. Lardieri, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano, "A Multi-Layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems," *Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-Time Systems*, 2007 (to appear).
27. R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging Quality of Service Control Behaviors for Reuse," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)* (Crystal City, VA), pp. 375–385, IEEE/IFIP, Apr./May 2002.

28. J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.
29. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp. 220–242, June 1997.
30. F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.
31. G. Blair, G. Coulson, F. Garcia, D. Hutchinson, and D. Shepherd, "The Design and Implementation of Open ORB 2." dsonline.computer.org/0106/features/bla0106_print.htm, 2001.
32. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-Based Systems," in *Proceedings of the 25th International Conference on Software Engineering* (Portland, OR), May 2003.
33. F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere, "Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications," in *Proceedings of the 11th International Symposium on High Performance Distributed Computing (HPDC'02)* (Edinburgh, UK), July 2002.
34. D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel, "Integration of Non-Functional Properties in Containers," *Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP)*, 2001.
35. M. A. de Miguel, "QoS-Aware Component Frameworks," in *The 10th International Workshop on Quality of Service (IWQoS 2002)* (Miami Beach, Florida), May 2002.
36. FOKUS, "Qedo Project Homepage." <http://qedo.berlios.de/>.
37. A. Flissi, C. Gransart, and P. Merle, "A Component-Based Software Infrastructure for Ubiquitous Computing," in *Proceedings of the 4th International Symposium on Parallel and Distributed Computing*, July 2005.
38. P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava, "MEAD: Support for Real-Time Fault-Tolerant CORBA," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 12, pp. 1527–1545, 2005.

16

Safe and Structured Use of Interrupts in Real-Time and Embedded Software

16.1	Introduction	16-1
16.2	Interrupt Definitions and Semantics	16-2
	Definitions • Semantics	
16.3	Problems in Interrupt-Driven Software	16-4
	Stack Overflow • Interrupt Overload • Real-Time Analysis	
16.4	Guidelines for Interrupt-Driven Embedded Software	16-9
	Part 1: Scheduling • Part 2: Callgraph • Part 3: Time Correctness • Part 4: Stack Correctness • Part 5: Concurrency Correctness	
16.5	Conclusions	16-12

John Regehr
University of Utah

16.1 Introduction

While developing embedded and real-time systems, it is usually necessary to write code that handles interrupts, or code that interacts with interrupt handlers. Interrupts are indispensable because they use hardware support to reduce both the latency and overhead of event detection, when compared to polling. Furthermore, modern embedded processor cores support wake-on-interrupt and can save energy by shutting down when idle, leaving only simple circuitry such as timers running. For example, a TelosB [13] sensor network node drains a pair of AA batteries in a few days by running the processor continuously but can run a useful application for months if the duty cycle is kept appropriately low.

Interrupts have some inherent drawbacks from a software engineering point of view. First, they are relatively nonportable across compilers and hardware platforms. Second, they lend themselves to a variety of severe software errors that are difficult to track down since they manifest only rarely. These problems give interrupts a bad reputation for leading to flaky software: a significant problem where the software is part of a highly available or safety-critical system.

The purpose of this chapter is to provide a technical introduction to interrupts and the problems that their use can introduce into an embedded system, and also to provide a set of design rules for developers of interrupt-driven software. This chapter neither addresses interrupts on shared-memory multiprocessors nor does it delve deeply into concurrency correctness: the avoidance of race conditions and deadlocks. Concurrency correctness is the subject of a large body of literature. Although the vast majority of this

literature addresses problems in creating correct thread-based systems, essentially all of it also applies to interrupt-driven systems.

16.2 Interrupt Definitions and Semantics

This section presents interrupt terminology and semantics in more detail than is typically found in an operating systems textbook or an embedded systems trade publication.

16.2.1 Definitions

Neither processor and operating system reference manuals nor the academic literature provides a consistent set of terminology for interrupts and their associated issues. This section provides a few definitions that are largely consistent with the prevailing usage among academics and practitioners.

The term *interrupt* has two closely related meanings. First, it is a hardware-supported asynchronous transfer of control to an interrupt vector based on the signaling of some condition external to the processor core. An *interrupt vector* is a dedicated or configurable location in the memory that specifies the address to which execution should jump when an interrupt occurs. Second, an interrupt is the execution of an *interrupt handler*: code that is reachable from an interrupt vector. It is irrelevant whether the interrupting condition originates on-chip (e.g., timer expiration) or off-chip (e.g., closure of a mechanical switch). Interrupts usually, but not always, return to the flow of control that was interrupted. Typically, an interrupt changes the state of main memory and of device registers but leaves the main processor context (registers, page tables, etc.) of the interrupted computation undisturbed.

An *interrupt controller* is a peripheral device that manages interrupts for a processor. Some embedded processor architectures (such as AVR, MSP430, and PIC) integrate a sophisticated interrupt controller, whereas others (such as ARM and x86) include only a rudimentary controller, necessitating an additional external interrupt controller, such as ARM's VIC or x86's PIC or APIC. Note that an external interrupt controller may be located on-chip but even so it is logically distinct from its processor.

Shared interrupts are those that share a single CPU-level interrupt line, necessitating demultiplexing in either hardware or software. In some cases, software must poll all hardware devices sharing an interrupt line on every interrupt. On some platforms, interrupts must be *acknowledged* by writing to a special device register; on other platforms this is unnecessary.

An interrupt is *pending* when its firing condition has been met and noticed by the interrupt controller, but when the interrupt handler has not yet begun to execute. A *missed interrupt* occurs when an interrupt's firing condition is met, but the interrupt does not become pending. Typically, an interrupt is missed when its firing condition becomes true when the interrupt is already pending. Rather than queueing interrupts, hardware platforms typically use a single bit to determine whether an interrupt is pending or not.

An interrupt is *disabled* when hardware support is used to prevent the interrupt from firing. Generally, it is possible to disable all interrupts by clearing a single bit in a hardware register, the *master interrupt enable bit*. Additionally, most interrupts have dedicated interrupt enable bits.

An interrupt fires only when the following conditions are true:

1. The interrupt is pending.
2. The processor's master interrupt enable bit is set.
3. The individual enable bit for the interrupt is set.
4. The processor is in between executing instructions, or else is in the middle of executing an interruptible instruction (see below). On a pipelined processor, the designers of the architecture choose a pipeline stage that checks for pending interrupts and a strategy for dealing with architectural state stored in pipeline registers.
5. No higher-priority interrupt meets conditions 1–3.

Interrupt *latency* is the length of the interval between the time at which an interrupt's firing condition is met and the time at which the first instruction of the interrupt handler begins to execute. Since an

interrupt only fires when all five conditions above are met, all five can contribute to interrupt latency. For example, there is often a short hardware-induced delay between the time at which a firing condition is met and the time at which the interrupt's pending bit is set. Then, there is typically a short delay while the processor finishes executing the current instruction. If the interrupt is disabled or if a higher-priority interrupt is pending, then the latency of an interrupt can be long. The *worst-case interrupt latency* is the longest possible latency across all possible executions of a system. Although the “expected worst-case latency” of an interrupt can be determined through testing, it is generally the case that the true worst-case latency can only be determined by static analysis of an embedded system's object code.

Nested interrupts occur when one interrupt handler preempts another, whereas a *reentrant* interrupt is one where multiple invocations of a single interrupt handler are concurrently active.* The distinction is important: nested interrupts are common and useful, whereas reentrant interrupts are typically needed only in specialized situations. Interrupts always execute *atomically* with respect to noninterrupt code in the sense that their internal states are invisible (remember that we are limiting the discussion to uniprocessors). If there are no nested interrupts, then interrupt handlers also execute atomically with respect to other interrupts.

There are a few major semantic distinctions between interrupts and threads. First, while threads can *block* waiting for some condition to become true, interrupts cannot. Blocking necessitates multiple stacks. Interrupts run to completion except when they nest, and nested interrupts always run in LIFO fashion. The second main distinction is that the thread-scheduling discipline is implemented in software, whereas interrupts are scheduled by the hardware interrupt controller.

Many platforms support a *nonmaskable interrupt* (NMI), which cannot be disabled. This interrupt is generally used for truly exceptional conditions such as hardware-level faults from which recovery is impossible. Therefore, the fact that an NMI preempts all computations in a system is irrelevant: those computations are about to be destroyed anyway.

16.2.2 Semantics

Unfortunately, there are genuine differences in the semantics of interrupts across hardware platforms and embedded compilers. This section covers the more important of these.

On most platforms, instructions execute atomically with respect to interrupts, but there are exceptions such as the x86 string-copy instructions. Nonatomic instructions are either restarted after the interrupt finishes or else their execution picks up from where it left off. Typically, nonatomic instructions are slow CISC-like instructions that involve many memory operations. However, slow instructions are not always interruptible. For example, the ARM architecture can store and load multiple registers using a single noninterruptible instruction. The GNU C compiler uses these instructions to reduce code size. Other compilers avoid the slow instructions to keep interrupt latencies low. The correct trade-off between latency and code size is, of course, application dependent.

Processor architectures vary in the amount of state that is saved when an interrupt fires. For example, the 68HC11, a CISC architecture, saves all registers automatically, whereas the AVR, a RISC, saves only the program counter. Most versions of the ARM architecture maintain register banks for interrupt mode and for “fast interrupt” mode to reduce the overhead of handling interrupts.

Minor details of handling interrupts can be smoothed over by the compiler. Almost all embedded C compilers support interrupt handlers that look like normal functions, but with a nonportable pragma indicating that the code generator should create an interrupt *prologue* and *epilogue* for the function rather than following the normal calling convention. The prologue and epilogue save and restore any necessary state that is not saved in hardware, hiding these details from the programmer. On all but the smallest platforms, the top-level interrupt handler may make function calls using the standard calling conventions.

*Note that a reentrant interrupt is different from a reentrant function. A reentrant function is one that operates properly when called concurrently from the main context and an interrupt, or from multiple interrupts.

Embedded systems supporting multiple threads have multiple stacks, one per thread. The simplest and highest-performance option for mixing interrupts and threads is for each interrupt to use the current thread stack. However, this wastes RAM since every thread stack must be large enough to contain the worst-case interrupt stack requirement. The alternative is for the interrupt to push a minimal amount of state onto the current stack before switching the stack pointer to a dedicated system stack where all subsequent data is stored. On ARM hardware, each register bank has its own stack pointer, meaning that stack switching is performed automatically and transparently.

16.3 Problems in Interrupt-Driven Software

This section addresses several difficult problems commonly encountered during the development of interrupt-driven embedded software: avoiding the possibility of stack overflow, dealing with interrupt overload, and meeting real-time deadlines. Although interrupt overload is technically part of the overall problem of meeting real-time deadlines, these are both difficult issues and we deal with them separately.

16.3.1 Stack Overflow

As an embedded system executes, its call stacks grow and shrink. If a stack is ever able to grow larger than the region of memory allocated to it, RAM becomes corrupted, invariably leading to system malfunction or crash. There is a tension between overprovisioning the stack, which wastes memory that could be used for other purposes and underprovisioning the stack, which risks creating a system that is prone to stack overflow.

Most developers are familiar with the *testing-based* approach to sizing stacks, where empirical data from simulated or actual runs of the system are used to guide allocation of stack memory. Testing-based stack allocation is more of an art than a science. Or, as Jack Ganssle [7, p. 90] puts it, “With experience, one learns the standard, scientific way to compute the proper size for a stack: Pick a size at random and hope.”

The stack memory allocation approach that is complementary to testing is *analysis-based*. In the simplest case, analysis is done by manually counting push, pop, and call instructions along different paths through the compiled system and adding up their effects. This approach does not scale, and so analysis must be automated. Sizing stacks through analysis can be more reliable than using testing, but analysis requires more effort unless very good tools are available.

An embedded system is *stack safe* if it is impossible for a stack to overflow into memory used for other purposes. This notion of safety is analogous to the *type safety* guaranteed by languages like Java, which ensure that, for example, an integer can never be interpreted as a pointer. In theory, the ideal system would be just barely stack safe: if any less memory were allocated to any stack, it would be possible for an overflow to happen. In practice, a safety margin may be desirable as a hedge against errors in stack depth analysis.

16.3.1.1 Analysis versus Testing

This section discusses the two approaches to sizing the stack in more detail. To implement the testing-based approach, one just runs the system and sees how big the stack gets. Ideally, the system is tested under heavy, diverse loads to exercise as much code as possible. Measuring the maximum extent of the stack is not hard: simulators can record this directly, and in a real system one would initialize stack memory to known values and see how many of these get overwritten. The testing-based approach treats the system, to a large extent, as a black box: it does not matter how or why stack memory is consumed, we only want to know how big each stack can get.

The second way to find the maximum size of the stack, the analysis-based approach, looks at the flow of control through an embedded system with the goal of finding the path that pushes the maximal amount of data onto the stack. This path may be complex, involving the main function plus several interrupt handlers.

Notice that the testing- and analysis-based approaches are both trying to do exactly the same thing: find the path through the program that produces worst-case stack behavior. The fundamental problem with testing is that for any complex system it is a mathematical certainty that testing will miss some paths

through the code. Consider, for example, a system containing five interrupt handlers, each of which fires an average of 50 times per second and spends 200 μ s at its worst-case stack depth. Assuming that interrupts arrive independently, the interrupts will exhibit their worst-case stacking only about every 300 years. In contrast, if 100,000 of these systems are deployed, we can expect to see one failure per day.

The analysis-based approach to bounding stack size, however, has the problem that it often overestimates the maximum stack size. Consider the extreme example of a reentrant interrupt handler, which makes it difficult to compute any bound at all. In other words, an analyzer may be forced to conclude that the worst-case stack depth is infinite, even though the system's developers know that this interrupt only arrives twice per second and cannot possibly preempt itself. Similarly, the analysis may be forced to assume that a timer interrupt may fire during a call to a certain function, when in fact this function is setting up the timer and it is known that the timer cannot fire until well after the function returns.

The true worst-case stack size for any complex system is unlikely to be computable, but it can be bracketed by combining the testing and analysis approaches. The three values are related like this:

$$\text{worst depth seen in testing} \leq \text{true worst depth} \leq \text{analytic worst depth}$$

To some extent, the gaps between these numbers can be narrowed through hard work. For example, if testing is missing paths through the system, then maybe more clever tests can be devised, or maybe the tests need to be run for longer time. In contrast, if the analysis misses some crucial feature of the system such as a causal relationship between interrupt handlers, then possibly the analysis can be extended to take this relationship into account.

The advantages of the analysis-based approach are that analysis can be much faster than testing (a few seconds or less), and that analysis can produce a guaranteed upper bound on stack depth resulting in stack-safe systems. Also, by finding the worst-case path through a program, analysis identifies functions that are good candidates for inlining or optimization to reduce stack depth. The main advantage of testing, however, is that of course all embedded systems are tested, and so there is no reason not to include stack size tests. If the results returned by analysis and testing are close together, then both testing and analysis can be considered to be successful. In contrast, if there is a substantial gap between the two results then there are several unpleasant possibilities. First, testing may be missing important paths through the code. Second, the analysis getting confused in some way, causing it to return an overly pessimistic result. If the gap cannot be closed then picking a size for the stack becomes an economic trade-off. If the stack is just a little larger than the worst size seen during testing, RAM consumption is minimized, but there is significant risk of stack overflow after deployment. In contrast, if the stack is set to its analyzed worst-case size, the possibility of overflow is eliminated, but less RAM will be available for other parts of the system.

16.3.1.2 Stack Depth Analysis

The *control flow graph* (CFG) for an embedded system is at the core of any analysis-based approach to determining worst-case stack depth. The CFG is simply a representation of the possible movement of the program counter through a system's code. For example, an arithmetic or logical operation always passes control to its successor and a branch may pass control either to its successor or to its target. The CFG can be extracted from either the high-level language (HLL) code or the executable code for a system. For purposes of bounding stack depth, a disassembled executable is preferable: the HLL code does not contain enough information to effectively bound stack depth.

For some systems, such as those written in the style of a cyclic executive, constructing the CFG is straightforward. For other systems, particularly those containing recursion, indirect calls, and a real-time operating system (RTOS), either a much more sophisticated analysis or a human in the loop is required. Indirect calls and jumps come from a variety of sources, such as event loops, callbacks, switch statements, device driver interfaces, exceptions, object dispatch tables, and thread schedulers.

The CFG alone is not enough to determine the maximum stack size: the effect that each instruction has on the stack depth is also important. Assume for a moment that a system only manipulates the stack through `push` and `pop` instructions. At this point, an analyzer can compute the stack effect of different

paths through the CFG: its job is to find the path that pushes most data onto the stack. This is easily accomplished with standard graph algorithms. However, if the system contains code that loads new values directly into the stack pointer, for example, to push an array onto the stack or to implement `alloca`, this simple approach is again insufficient. A stronger approach that can identify (or at least bound) the values loaded into the stack pointer is required.

A simple cyclic executive can be described by a CFG. In the presence of interrupts or threads, a system contains multiple CFGs: one per entry point. Stack bounds for each of these graphs can be computed using the same algorithm. A complication that now arises is that we need to know how to put the individual results together to form a stack depth bound for the whole system. If all interrupts run without enabling interrupts, then this is easy:

$$\text{worst case depth} = \text{depth of main} + \text{maximum depth of any interrupt}$$

Since many systems enable interrupts while running interrupt handlers, the following equation is more broadly applicable:

$$\text{worst case depth} = \text{depth of main} + \text{total depth of all interrupts}$$

There are two problems with using the second equation. First, it provides an answer that is potentially too low if some interrupt handlers are reentrant. Second, it provides an answer that is too high if, for example, only one interrupt enables interrupts while running. A better approach is to determine the exact interrupt preemption graph—which interrupt handlers can preempt, which others and when—and to use this as a basis for computing a bound.

From the point of view of stack depth analysis, reentrant interrupt handlers are a big problem. Stack depth cannot be bounded without knowing the maximum number of outstanding instances of the reentrant interrupt handler. Finding this number requires solving a global timing analysis problem that, as far as we know, has never been addressed either in industry or in academia. In general, a better solution is to avoid building systems that permit reentrant interrupts.

16.3.1.3 Tools for Stack Depth Analysis

A few compilers emit stack depth information along with object code. For example, GNAT, the GNU Ada Translator, includes stack bounding support [4]. AbsInt sells a standalone stack depth analysis tool called StackAnalyzer [1]. The only tools that we know of which analyze stack depth in the presence of interrupts are ours [16] and Brylow et al.'s [5]. In related work, Barua et al. [3] investigate an approach that, upon detecting stack overflow, spills the stack data into an unused region of RAM, if one is available.

16.3.2 Interrupt Overload

Many interrupt-driven embedded systems are vulnerable to *interrupt overload*: the condition where external interrupts are signaled frequently enough that other activities running on a processor are starved. An interrupt overload incident occurred during the first moon landing, which was nearly aborted when a flood of radar data overloaded a CPU on the Lunar Landing Module, resulting in guidance computer resets [14, pp. 345–355]. The problem on Apollo 11 appears to have been caused by spurious signals coming from a disconnected device. Neglecting to bound maximum interrupt arrival rates creates an important gap in the chain of assumptions leading to a high-assurance embedded system.

Interrupt overload is not necessarily caused by high interrupt loads but rather by unexpectedly high interrupt loads. For example, a fast processor running software that performs minimal work in interrupt mode can easily handle hundreds of thousands of interrupts per second. In contrast, a slow processor running lengthy interrupt code can be overwhelmed by merely hundreds of interrupts per second.

Computing a reliable maximum request rate for an interrupt source in an embedded system is difficult, often requiring reasoning about complex physical systems. For example, consider an optical shaft encoder used to measure wheel speed on a robot. The maximum interrupt rate of the encoder depends on the maximum speed of the robot and the design of the encoder wheel. However, what happens if the robot

TABLE 16.1 Potential Sources of Excessive Interrupts for Embedded Processors. The Top Part of the Table Reflects the Results of Experiments and the Bottom Part Presents Numbers That We Computed or Found in the Literature

Source	Max. Interrupt Freq. (Hz)
Knife switch bounce	333
Loose wire	500
Toggle switch bounce	1,000
Rocker switch bounce	1,300
Serial port @115 kbps	11,500
10 Mbps Ethernet	14,880
CAN bus	15,000
I2C bus	50,000
USB	90,000
100 Mbps Ethernet	148,800
Gigabit Ethernet	1,488,000

exceeds its maximum design speed, for example, while going downhill? What if the encoder wheel gets dirty, causing it to deliver pulses too often?

The data in Table 16.1 show some measured and computed worst-case interrupt rates. Switches that we tested can generate surprisingly high-frequency events (“bounces”), exceeding 1 kHz, during the transition between open and closed. This could easily cause problems for a system designed to handle only tens of switch transitions per second. The traditional way to debounce a switch is to implement a low-pass filter either in hardware or software. Although debouncing techniques are well known to embedded systems designers, it is not enough just to debounce all switches: new and unforeseen “switches” can appear at runtime as a result of loose contacts or damaged wires. Both of these problems are more likely in embedded systems that operate in difficult environmental conditions with heat and vibration, and without routine maintenance. These conditions are, of course, very common—for example, in automobiles.

Network interface controllers (NICs) represent another potential source for interrupt overload. For example, consider an embedded CPU that exchanges data with other processors over 10 Mbps Ethernet using a specialized protocol that specifies 1000-byte packets. If the NIC generates an interrupt on every packet arrival, the maximum interrupt rate is 1.25 kHz. However, if a malfunctioning or malicious node sends minimum-sized (72-byte) packets, the interrupt rate increases to nearly 15 kHz [10], potentially starving important processing.

16.3.2.1 Preventing Interrupt Overload

The upper bound on the amount of time spent in a given interrupt handler is easy to compute: it is just the maximum execution time of the handler multiplied by the worst-case arrival rate. For example, if a network interface can deliver up to 10,000 packets per second and the interrupt handler runs for 15 μ s, then at most 15% of the processor’s time will be spent handling network interrupts.

There are a few basic strategies for making interrupt overload less likely or impossible:

- Keep interrupt handlers short. This is almost always a good idea anyway.
- Bound the arrival rates of interrupts. In many cases, the worst-case arrival rate of an interrupt can be bounded by studying the interrupting device. Clearly, when assumptions are made (e.g., about the maximum rate of change of some physical quantity or the minimum size packet that will cross a network), these assumptions must be carefully justified.
- Reduce the worst-case arrival rate of interrupts. For example, a smart NIC need not generate an interrupt for each arriving packet and a serial port need not generate an interrupt for each arriving byte. The general pattern is that smart devices can greatly reduce the interrupt load on a processor.
- Poll for events rather than using interrupts. The problem with polling is that it adds processor overhead even when there are no events to process. Some existing hardware devices, such as NICs,

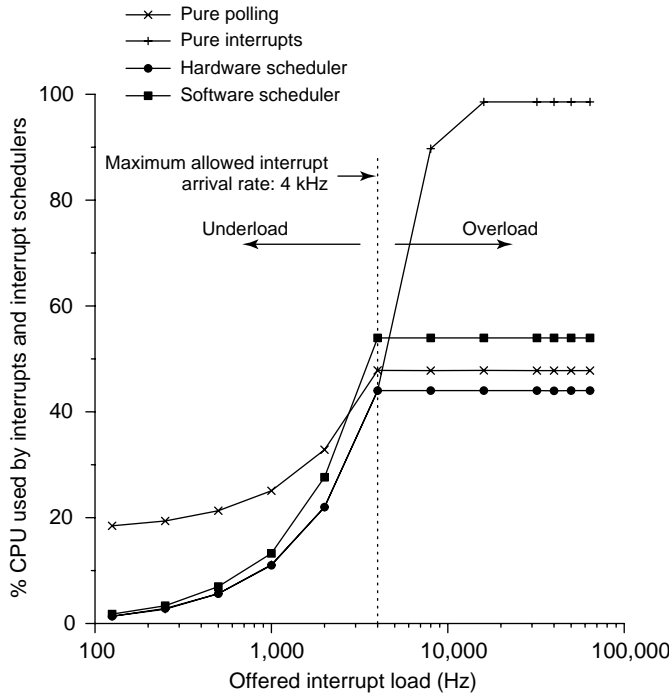


FIGURE 16.1 Interrupt schedulers at work.

already adaptively switch between interrupts and polling depending on system load, but generally they do not do so in a timely fashion: there is a period of overload before the mode switch occurs. This would be harmful in the context of a real-time system.

- Use an *interrupt scheduler*. A technique that we developed in previous work [15]. An interrupt scheduler is a small piece of hardware or software that limits the maximum arrival rate of an interrupt source. The software implementation works by switching between interrupt behavior when arrival rates are low and polling-like behavior when arrival rates are high. Figure 16.1 illustrates the effect of an interrupt scheduler attached to an Atmel AVR processor running at 4 MHz. In this scenario, the interrupt handler performs 250 cycles (62.5 μ s) of work and the developers would like to handle at most 4000 interrupts per second. Pure polling performs well during overload but incurs close to 20% CPU overhead even when no interrupts are arriving. In contrast, the pure interrupt-driven solution performs well when few interrupts arrive but saturates the CPU during overload. The interrupt schedulers support good performance during both underload and overload. The software scheduler, which runs on an unmodified CPU, incurs some overhead, whereas the hardware scheduler essentially offers perfect performance characteristics.

16.3.3 Real-Time Analysis

Section 16.3.2 addressed the problem of bounding the arrival rate of interrupts to prevent them from starving noninterrupt work. In reality, preventing interrupt overload is just one subproblem of the overall problem of *real-time schedulability analysis*: ensuring that the time constraints of all computations can be met. The literature on real-time scheduling is vast. In this section, we only deal with issues specific to interrupt-driven software and provide references to the wider literature.

Every embedded system has a discipline for scheduling interrupts: for deciding which interrupt handler gets to execute, and when. Some hardware platforms, such as the x86 with PIC, provide *prioritized, preemptive scheduling* by default. A prioritized interrupt scheduler permits a high-priority interrupt to

preempt a lower-priority interrupt. In contrast, if a low-priority interrupt becomes pending while a higher-priority interrupt is executing, the low-priority interrupt cannot execute until it is the highest-priority pending interrupt. Prioritized interrupts can be analyzed using *rate-monotonic analysis* (RMA). In some cases, interrupt priorities are fixed at hardware design time, limiting the flexibility of a system to accommodate, for example, a new latency-sensitive peripheral. Embedded platforms with sophisticated interrupt controllers permit priorities to be configured dynamically.

To apply RMA to an interrupt-driven system, it is necessary to know the *worst-case execution time* (WCET) C and the *minimum interarrival time* T for each of the N interrupt handlers in a system. The minimum interarrival time of an interrupt is the inverse of its maximum arrival rate. Then, the overall CPU utilization U of interrupts is computed as

$$U = \sum_{i=1 \dots N} \frac{C_i}{T_i}$$

As long as U is less than 0.69, it is guaranteed that every interrupt will complete before the end of its period. In contrast, if interrupts have deadlines shorter than their interarrival times, or if interrupts are disabled for nonnegligible amounts of time, then the situation is more complex. However, solutions exist to many different variations of the scheduling problem. Good references on prioritized, preemptive scheduling include Audsley et al. [2] and Klein et al. [11]. Also, products for analyzing fixed-priority systems, such as RapidRMA [20] and the RTA-OSEK Planner [6], are available.

An important benefit of prioritized, preemptive interrupt scheduling is that it composes naturally with the predominant scheduling discipline provided by RTOSs: prioritized, preemptive thread scheduling. In this case, an overall schedulability analysis is performed by considering interrupts to have higher priority than threads.

Some platforms, such as Atmel's AVR, provide prioritized scheduling among pending interrupts but not among concurrently executing interrupts. Thus, by default, the AVR does not provide prioritized preemptive interrupt scheduling: the distinction is subtle but important. For example, if a low-priority AVR interrupt becomes pending while a high-priority interrupt is running with interrupts enabled, then the low-priority interrupt preempts the high-priority interrupt and begins to execute. To implement prioritized preemptive scheduling on the AVR, it is necessary for each interrupt handler to clear the individual enable bits for all lower-priority interrupts before enabling interrupts. Then, these bits must be reset to their previous values before the interrupt handler finishes executing. This bit-twiddling is cumbersome and inefficient. In contrast, the AVR naturally supports *prioritized, nonpreemptive interrupt handling* if interrupt handlers avoid enabling interrupts.

The real-time behavior of a system supporting prioritized, nonpreemptive interrupt handling can be analyzed using nonpreemptive static priority analysis for which George et al. [9] have worked out the scheduling equations. To analyze nonpreemptive interrupts composed with preemptive threads, an analysis supporting mixed preemption modes must be used. Saksena and Wang's [17,21] analysis for preemption threshold scheduling is suitable. In this case, the priority of each interrupt and thread is set as in the fully preemptive case, while the preemption threshold of each interrupt handler is set to the maximum priority, and the preemption threshold of each thread is set to the priority of the thread. The overall effect is that interrupts may preempt threads but not interrupts, while threads can preempt each other as expected.

16.4 Guidelines for Interrupt-Driven Embedded Software

This section synthesizes the information from this chapter into a set of design guidelines. These guidelines are in the spirit of MISRA C [12]: a subset of the C language that is popular among embedded software developers. For embedded systems that are not safety critical, many of these rules can be relaxed. For example, a full WCET and schedulability analysis is probably unnecessary for a system where the only consequence of missed real-time deadlines is degraded performance (e.g., dropped packets and missed

data samples). In contrast, it is seldom a good idea to relax the rules for stack correctness or concurrency correctness, since stack overflows and race conditions generally lead to undefined behavior.

For each of these rules, developers creating embedded software should be able to justify—to themselves, to other team members, to managers, or to external design reviewers—either that the rule has been followed, or else that it does not apply. Obviously, these rules should be considered to be over and above any in-house software design guidelines that already exist. Chapters 4 and 5, and Appendix A, of Ganssle’s [8] book are also good sources for design guidance for interrupt-driven software.

A difference between these guidelines and MISRA C is that tools for statically checking many of the interrupt guidelines either do not exist or else are not in common use. A reasonable agenda for applied research in embedded software would be to create software tools that can soundly and completely check for conformance to these rules.

16.4.1 Part 1: Scheduling

This set of guidelines invites developers to quantify the scheduling discipline for interrupts and to figure out if it is the right one.

1. The scheduling discipline for interrupts must be specified. This is straightforward if scheduling is either preemptive or nonpreemptive priority-based scheduling. However, if the scheduling discipline mixes preemptive and nonpreemptive behavior, or is otherwise customized, then it must be precisely defined. Furthermore, response time equations for the scheduling discipline must either be found in the literature or derived.
2. The scheduling discipline must never permit two different interrupt handlers to each preempt the other. This is never beneficial: it can only increase the worst-case interrupt latency and worst-case stack depth.
3. When one interrupt is permitted to preempt another, the reason for permitting the preemption should be precisely identified. For example, “the timer interrupt is permitted to preempt the Ethernet interrupt because the timer has a real-time deadline of 50 μ s while the Ethernet interrupt may run for as long as 250 μ s.” System designers should strive to eliminate any preemption relation for which no strong reason exists. Useless preemption increases overhead, increases stack memory usage, and increases the likelihood of race conditions.

16.4.2 Part 2: Callgraph

Once the callgraphs for a system have been identified, many nontrivial program properties can be computed.

1. The callgraphs for the system must be identified. There is one callgraph per interrupt, one for the main (noninterrupt) context, and one for each thread (if any). Most callgraph construction can be performed automatically; human intervention may be required to deal with function pointers, computed gotos, long jumps, and exceptions.
2. Every cycle in the callgraph indicates the presence of a direct or indirect recursive loop. The worst-case depth of each recursive loop must be bounded manually. Recursive loops should be scrutinized and avoided if possible. It is particularly important to identify (or show the absence of) unintentional recursive loops. Unintentional recursion commonly occurs in systems that use callbacks.

16.4.3 Part 3: Time Correctness

1. The maximum arrival rate of each interrupt source must be determined. This could be a simple rate or else it could be bursty. The reason that the interrupt rate cannot exceed the maximum must be specified. This reason can either be external (i.e., there is some physical limit on the interrupt rate) or internal (i.e., an interrupt scheduler has been implemented).
2. The deadline for each interrupt must be determined. This is the maximum amount of time that may elapse between the interrupt’s firing condition becoming true and the interrupt’s handler running.

Furthermore, the cost of missing the deadline must be quantified. Some interrupts can be missed with very little cost, whereas others may be mission critical. In general, only part of an interrupt handler (e.g., the part that reads a byte out of a FIFO) must complete by the deadline. The final instruction of the time-sensitive part of the interrupt handler should be identified.

3. The WCET of each interrupt must be determined. This can be computed using a WCET tool, by measurement if the interrupt code is straight line or nearly so, or even by manual inspection. Gross estimates of WCET may be valid if the consequences of missing real-time deadlines are not catastrophic.
4. The WCET of the nontime-sensitive part of each interrupt handler should be considered. If these entrain a substantial amount of code, then the nontime-sensitive code should be run in a delayed context such as a deferred procedure call [19, pp. 107–111] or a thread to avoid unduly delaying lower-priority interrupts.
5. The longest amount of time for which interrupts are disabled must be determined, this is simply the maximum of the WCETs of all code segments that run with interrupts disabled. This time is used as the *blocking term* [18] in the schedulability equations.
6. The ability of each interrupt to meet its deadlines must be verified. This is accomplished by plugging the system's constants into the appropriate schedulability equation, such as the rate-monotonic rule shown in Section 16.3.3.
7. The impact of interrupts on any noninterrupt work with real-time deadlines must be considered; the details are beyond the scope of this chapter.

16.4.4 Part 4: Stack Correctness

1. A *stack model* for the system should be developed, identifying the effect of interrupts on each stack in the system (see Section 16.2.2) and also the way that individual interrupt stack depths combine (see Section 16.3.1.2). Any assumptions made by the stack model should be explicitly justified.
2. The stack budget for interrupts must be determined. This is just the worst-case (smallest) amount of RAM that may be available for interrupts to use under any circumstances.
3. The worst-case stack memory usage of each interrupt must be determined. This could be done using a tool, by manual inspection of the object code, or else by measurement if the interrupt code is straight line or nearly so.
4. The overall worst-case stack depth for interrupts must be computed by plugging the system's constants into the stack model.

16.4.5 Part 5: Concurrency Correctness

1. Reachability analysis must be performed using the callgraph to determine, which data structures are reachable from which interrupt handlers, and from the program's main context. A data structure is any variable or collection of variables whose state is governed by a single set of invariants. A data structure is *shared* if it may be accessed by more than one interrupt handler, if it may be accessed by at least one interrupt and by the noninterrupt context, or if it is accessed by any reentrant interrupt. Any data structures not meeting these criteria are *unshared*. Note that the reachability analysis is complicated by pointers, whose worst-case points-to sets must be taken into account.
2. All automatic (stack-allocated) data structures must be unshared.
3. A data structure is *protected* if its invariants are broken only in an *atomic* context. In an interrupt-driven system, atomicity is achieved by disabling one or more interrupts. All shared variables should be protected.
4. For every unprotected shared variable, a detailed description must be provided explaining why the unprotected access does not compromise system correctness.
5. For every reentrant interrupt handler, a detailed explanation must be provided that explains why the reentrancy is correct and what benefit is derived from it.
6. Every function that is reachable from an interrupt handler must be reentrant.

16.5 Conclusions

This chapter has summarized some problems associated with creating correct interrupt-driven software, and it has proposed a set of design rules that can help developers avoid common (and uncommon) traps and pitfalls. It is important for developers to gain a strong understanding of interrupts and their associated software development issues. First, these issues occur across a wide range of embedded software systems. Second, many of the same issues recur in the kernels of general-purpose operating systems. Third, it is empirically known that interrupts are hard to get right. Finally, embedded systems are often used in safety-critical applications where the consequences of a bug can be severe.

References

1. AbsInt. StackAnalyzer, 2004. <http://www.absint.com/stackanalyzer>.
2. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5): 284–292, Sept. 1993.
3. S. Biswas, M. Simpson, and R. Barua. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proc. of the ACM Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Washington, DC, Sept. 2004.
4. E. Botcazou, C. Comar, and O. Hainque. Compile-time stack requirements analysis with GCC. In *Proc. of the 2005 GCC Developers Summit*, Ottawa, Canada, June 2005.
5. D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pp. 47–56, Toronto, Canada, May 2001.
6. ETAS Inc. OSEK-RTA Planner, 2006. <http://en.etasgroup.com/products/rta>.
7. J. Ganssle. *The Art of Designing Embedded Systems*. Newnes, 1999.
8. J. Ganssle. Cheap changes. *Embedded.com*, July 2004. <http://www.embedded.com/showArticle.jhtml?articleID=22103322>.
9. L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report 2966, INRIA, Rocquencourt, France, Sept. 1996.
10. S. Karlin and L. Peterson. Maximum packet rates for full-duplex Ethernet. Technical Report TR-645-02, Princeton University, Feb. 2002.
11. M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
12. MISRA. MISRA-C:2004—Guidelines for the use of the C language in critical systems, 2004. <http://www.misra.org.uk>.
13. Moteiv. Telos rev. B datasheet, 2005. <http://www.moteiv.com>.
14. C. Murray and C. B. Cox. *Apollo: The Race to the Moon*. Simon and Schuster, 1989.
15. J. Regehr and U. Duongsaa. Preventing interrupt overload. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.
16. J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pp. 306–322, Philadelphia, PA, Oct. 2003.
17. M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp. (RTSS)*, Orlando, FL, Nov. 2000.
18. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, Sept. 1990.
19. D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.
20. TriPacific Inc. RapidRMA, 2006. <http://www.tripac.com>.
21. Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of the 6th Intl. Workshop on Real-Time Computing Systems and Applications*, Hong Kong, Dec. 1999.

17

QoS Support and an Analytic Study for USB 1.x/2.0 Devices

Chih-Yuan Huang

National Taiwan University

Shi-Wu Lo

National Chung-Cheng University

Tei-Wei Kuo

National Taiwan University

Ai-Chun Pang

National Taiwan University

17.1	Introduction	17-1
	Overview of USB • A USB Subsystem	
17.2	QoS Guarantees for USB Subsystems	17-6
	Introduction • Overview • QoS Guarantees for Periodic Requests • Probabilistic QoS Analysis for Sporadic Transfers	
17.3	Summary	17-18

17.1 Introduction

With the recent advances in the operating system design and hardware technology, resource management strategies in computing systems begin to draw more attention from system engineers. Exploring all possible strategies is complicated by the modularization design, which usually divides an operating system into several components. The concept of modularization separates and isolates the management of different types of system resources. The resource management issues are no longer dominated by CPU scheduling policies. Even if a process obtains a good service from the CPU scheduler, it does not guarantee that the process could always access disks as it wishes or use any specific Universal Serial Bus (USB) devices with proper services. The definition of “quality of service” (QoS) could also be different when different resources are considered: for an I/O interface, QoS might be in terms of bandwidth reserved, while QoS for memory management might be in terms of the number of bytes allocated in memory. Different QoS definitions and enforcement mechanisms for different components of an operating system are needed for servicing applications with a proper QoS.

The modularization of computer systems has moved the duty of resource allocation gradually away from the kernel. While various I/O devices are manufactured and connected to a few common interfaces, such as SCSI, USB, and IEEE-1394, the I/O subsystems have now become one of the major players in providing QoS guarantees for applications. The traditional concepts and implementations in resource allocation, such as fairness or throughput, could no longer satisfy the needs of many modern applications. The delivering of QoS guarantees to applications is the responsibility of not only the so-called kernel but also of all the subsystems involved in the servicing of the applications. The execution of a video player serves as a good example in how to satisfy the QoS requirements of an application. The scheduling of CPU usage alone could not resolve the QoS issues. Instead, resource allocation problems on disks, I/O bus, or even networks could no longer be negligible. A simple I/O subsystem is shown in Figure 17.1.

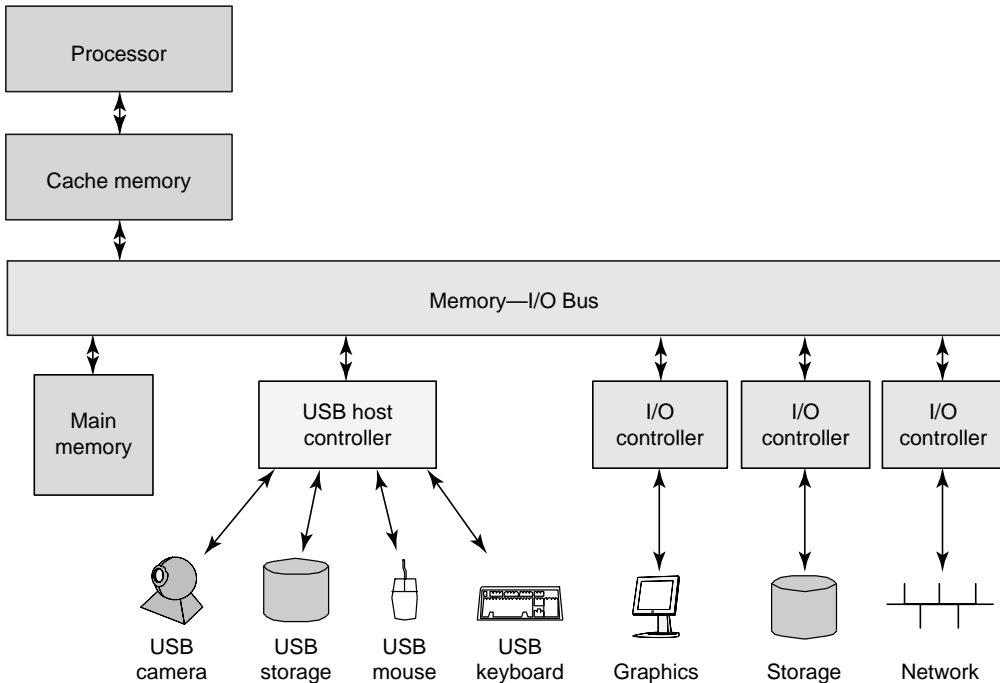


FIGURE 17.1 A simple I/O subsystem.

How to provide QoS guarantees for task executions has been an active research and implementation topic in the past decades. Real-time scheduling algorithms [9,10,12] were often adopted by researchers in enforcing resource reservation. Several good system implementations were done over RTLinux [21], a real-time extension of Linux, such as those for Ethernet-based real-time communication [16] and a lightweight TCP/IP stack for embedded systems [15]. A real-time emulation program was proposed to build soft real-time scheduling on the top of UNIX [1], and the Linux source code was modified by adding a new scheduling class called SCHQH_QOS to let applications specify the amount of CPU time per period [2]. At the Carnegie Mellon University, the concept of time reservation was proposed and implemented in the real-time Mach and Linux to support various multimedia and time-critical applications [6,11]. In addition to the QoS reservation for CPU time, researchers have also started exploring the resource allocation problem over various buses. In particular, the controller area network (CAN), which is a serial communication protocol for distributed real-time systems, was studied by Hong and Kim [3] in proposing a bandwidth reservation algorithm. In Ref. 13, Natale discussed the applicability of the earliest deadline first (EDF) algorithm for the scheduling of CAN messages, where QHF always allocated resources to the ready job with the closest deadline. A higher bus utilization was delivered. Nolte et al. [14] presented a new share-driven server-based method for scheduling messages sent over by the controller area network (CAN). The server-based scheduling is based on the earliest deadline first (EDF) algorithm, and it allows a higher utilization of the network than that with the CAN's native fixed-priority scheduling approach. Kettler et al. [7] developed formal scheduling models for several types of commonly used buses, e.g., PCI and MCA, in PCs and workstations. Although USB is a very popular bus, little work has been done for USB request scheduling.

17.1.1 Overview of USB

USB is widely used for peripherals in modern desktop systems. USB provides a flexible and user-friendly architecture for connecting a diverse range of peripherals to systems. USB emerged as a result of the difficulties associated with the cost, configuration, and attachment of peripheral devices in the personal

computer environment. That is, USB creates a method of attaching and accessing peripheral devices that reduces the overall cost, simplifies the attachment and configuration from the end-user perspective, and solves several technical issues associated with the old-style peripherals. This reduces the necessary connections and also uses fewer system resources (e.g., I/O channels, memory addresses, interrupt channels, and DMA channels).

USB is a tree-like topology (see Figure 17.5). The branches of the tree are extended using USB hubs. Up to 127 devices can be attached in this hierarchy. The tree begins at the **root**, which comprises a host controller. A host controller implements the serial packet transfer hardware (transmit/receive, error detection, frame management, DMA) that host software can use to communicate with USB devices. The details of packet transfers are omitted in this chapter and can be found in the USB specification [18–20]. USB is designed to support many types of devices, such as human interface devices (HIDs) (e.g., a keyboard and a mouse), block devices (e.g., disks), communication transceivers, stereo speakers, and video cameras. USB 1.x provides good bandwidth (up to 12 Mbit/s) with USB 2.0 (480 Mbit/s) providing even better throughput for data intensive applications. The USB specification version 1.x defined only two device speeds, such as low speed at 1.5 Mbit/s and full speed at 12 Mbit/s. The high speed at 480 Mbit/s has been added in USB specification 2.0. Low-speed devices are the cheapest to manufacture and are adequate for supporting low data rate devices, such as mice, keyboards, and a wealth of other equipment designed to interact with people. The introduction of high-speed data rate enables high-bandwidth devices such as web cameras, printers, and mass storage devices. The data transfer modes on USB could be classified into four categories: *isochronous transfer*, *interrupt transfer*, *control transfer*, and *bulk transfer*. Isochronous and interrupt transfers are periodic data transfers, while control and bulk transfers introduce aperiodic bus activities. Different types of devices require different bandwidths and ways to interact with USB host controllers, where a host controller manages the bandwidth of a USB bus. For example, a HID demands a periodic but light bus workload. A storage device requires a best-effort service from the corresponding host controller. However, an operating system must manage a number of devices through USB simultaneously, and devices would compete for a limited bus bandwidth. How to properly manage USB bandwidth is of paramount importance if reasonable QoS requirements are, indeed, considered.

In addition, we shall not only address scheduling issues related to data transfers of USB 1.x/2.0 devices but also the best-effort and guaranteed services for USB requests. This chapter will first provide an integrated real-time driver architecture over existing USB implementations for compatibility considerations. It then presents a real-time scheduling algorithm that processes requests in the time frame for USB 1.x/2.0. A period modification approach and the corresponding schedulability test are presented to provide guaranteed QoS services to isochronous and interrupt requests for USB 1.x/2.0 devices that are periodic. It then provides methodology in request insertions for the data structure dedicated for the USB 1.x/2.0 host controllers. Performance improvement strategies on the binary encoding of request frequencies are also proposed to boost the bandwidth utilization. For best-effort requests, such as that for bulk transfers, we revise the existing methodology and provide a probabilistic performance analysis.

17.1.2 A USB Subsystem

This section provides an overview to a USB subsystem on Linux. A USB subsystem on Linux is a layered structure, as shown in Figure 17.2. Since USB supports many types of devices, such as character and block devices, it is better to provide an abstraction of USB device drivers so that the kernel could have a transparent view over devices. For example, the kernel does not need to distinguish the ways in which an ATAPI CD-ROM device is connected to the system (e.g., through IDE or USB). A common ATAPI driver is adopted to invoke a proper USB driver to access the CD-ROM. The USB core, as shown in Figure 17.2, provides a set of consistent and abstracted APIs to the upper-level device drivers (so-called *client drivers*, i.e., one called by the ATAPI driver in the above example). The USB core would call the corresponding USB HC driver of a USB host controller to access its devices. Inside the USB core, the USB control protocol is implemented so that client drivers do not need to manage how data packets are sent to or received from the devices.

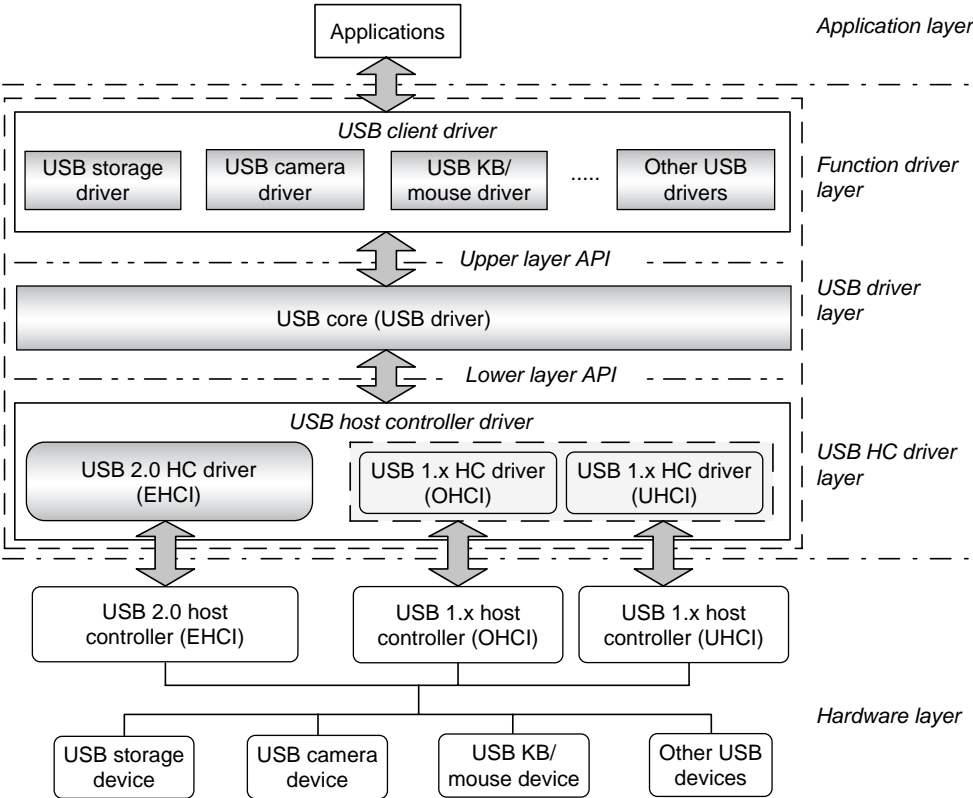


FIGURE 17.2 The device driver hierarchy of the USB subsystem.

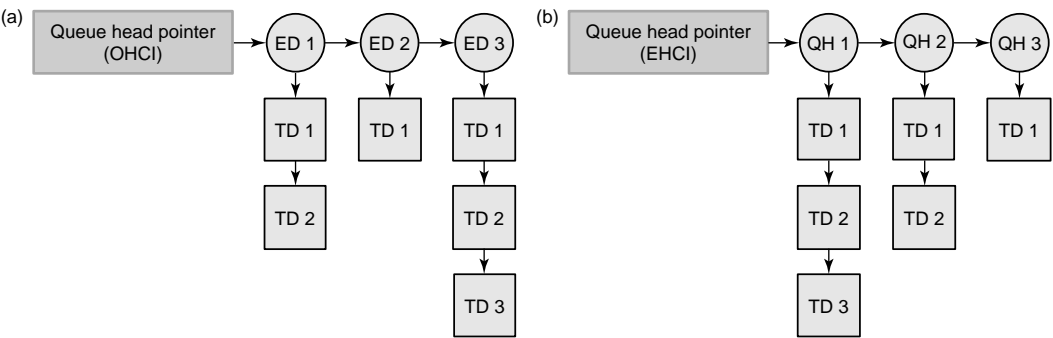


FIGURE 17.3 (a) The queue list structure of the OHCI USB schedule. (b) The queue list structure of the EHCI USB schedule.

The USB core translates requests issued by the USB device drivers into host-controller-awared data structures called *USB schedule*. The USB schedule is memory resident and consists of a number of queues of transfer descriptors (TDs), as shown in Figure 17.3a for USB 1.x and Figure 17.3b for USB 2.0, where one or more TDs are created for each application request by the corresponding USB device driver. The host controller interprets the TDs to access the designated devices. The queues are hooked on the proper device registers of a USB host controller, as shown in Figure 17.4a for USB 1.x and Figure 17.4b for USB 2.0,

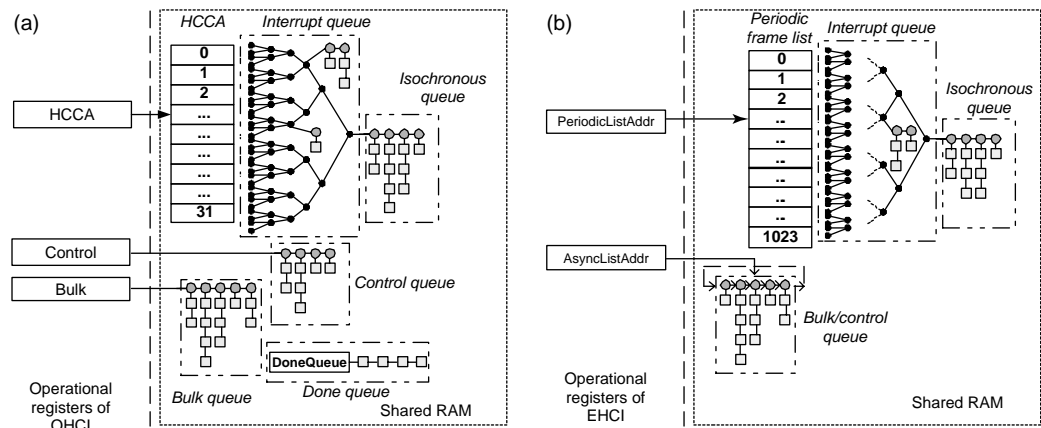


FIGURE 17.4 (a) The USB 1.x communication channel. (b) The USB 2.0 communication channel.

where the basic rule in hooking queues on the schedule is on the types of transfers. The actual layout of the USB schedule is host controller-dependent. There are three major types of host controllers: the Open Host Controller Interface (OHCI) [18] and the Universal Host Controller Interface (UHCI) [4] belong to the USB 1.x specifications, and the Enhanced Host Controller Interface (EHCI) [5] belongs to the USB 2.0 specification.

As mentioned earlier, data transfer modes of USB could be classified into four categories: isochronous transfer, interrupt transfer, control transfer, and bulk transfer. In OHCI specification (a USB 1.x host controller specification), each of the transfer modes needs a queue list to link the data transfer packets, i.e., TDs and endpoint descriptors (EDs), to the corresponding devices, as shown in Figure 17.3a, where an ED is at the head of the TD list for each device to record the transfer information regarding the device, e.g., the device number for the host controller. For bulk and control data transfers, the host controller maintains a pointer to the head of the TD list for each type, as shown in Figure 17.4a. For interrupt data transfers, a USB subsystem maintains 32 interrupt entries, which forms a tree structure. Each (internal or leaf) node could be hooked up with a list of TDs for a device of the interrupt type. The list of TDs for all isochronous devices is hooked up at the root of the tree structure, as shown in Figure 17.4a. Each interrupt entry will be visited every 32 ms. When an interrupt entry is visited, all devices with a TD list on the path between the interrupt entry and the root will be serviced within 1 ms. In other words, the isochronous queue will be serviced every 1 ms, provided that time is left after servicing other devices on the path (ahead of the root).

In EHCI specification (a USB 2.0 host controller specification), each of the transfer modes needs a queue list to link the data transfer packets, i.e., queue-element-transfer-descriptors (referred to as TDs, too) and queue-head descriptors (QHs), to the corresponding devices, as shown in Figure 17.3b, where a QH is at the head of the TD list for each process on a specific device to record the transfer information regarding the device, e.g., the device number for the host controller. QHs are very similar to the EDs in OHCI. For bulk and control data transfers, the host controller maintains a pointer to the head of the TD list for each type, as shown in Figure 17.4b. For interrupt data transfers, a USB subsystem maintains 1024 interrupt entries, which form a tree structure in EHCI. Each (internal or leaf) node could be hooked up with a list of TDs for a device of the interrupt type. The list of TDs for all isochronous devices is hooked up at the root of the polling tree. Each periodic-frame-list entry, as shown in Figure 17.4b, will be visited every 1024 μ -frame, which is 8 times faster than the millisecond frame of USB 1.x. When a periodic-frame-list entry is visited, all devices with a TD list on the path between the periodic-frame-list entry and the root will be serviced within 125 μ s. In other words, the isochronous queue will be serviced every μ -frame, provided that some time is left after servicing other devices on the path (ahead of the root).

For the USB client drivers, the control and data transfer are accomplished by interacting with the USB core through a data structure called the USB request block (URB). Requests are filled up in URBs and

handled by the USB core asynchronously: A configured callback function will be called on the completion of each URB. With the needs of the QoS support, information regarding timing constraints should be included in URBs as parameters. In this chapter, we shall provide QoS management for the USB core, including the management of the polling tree for interrupt entries and the methodology for USB bandwidth allocation.

17.2 QoS Guarantees for USB Subsystems

This section provides several methods to achieve QoS guarantee support for USB 1.x and 2.0 subsystems. From Sections 17.1.1 and 17.1.2, we knew that USB 1.x and 2.0 adopt similar mechanisms to transfer data between host controllers and USB devices, so we will focus our discussions on USB 2.0 and EHCI in this section. The methods provided in this section about QoS support for USB 2.0 could also apply to USB 1.x subsystem.

17.2.1 Introduction

A USB host controller manages the bandwidth of a USB bus. The configurations of USB 1.x and 2.0 devices/hubs/controllers vary, as shown in Figure 17.5. USB 2.0 provides better throughput for data-intensive applications and is also backward compatible with USB 1.x. When a USB 2.0 host controller needs to talk to a USB 1.x device, the USB 2.0 host controller goes through either a companion controller (i.e., a USB 1.x host) or a transaction translator in the USB 2.0 hub (i.e., the left child of the USB 2.0 host controller in Figure 17.5). The USB 2.0 host talks to the translator of the USB 2.0 hub at a high speed. The translator talks to the USB 1.x hub that connects to both the USB 1.x device and the USB 2.0 hub in a compatible way. There is one translator and several ports per hub, where each port is connected to either a device or a hub. In this section, we are interested in the QoS management of USB 2.0 bus bandwidth that is shared among USB 2.0 and 1.x devices. That is, we shall explore the QoS management issues for requests of all devices or hubs in the left subtree of the USB 2.0 host in Figure 17.5. Note that devices or hubs in the right subtree of the USB 2.0 host controller would be managed by a USB 1.x controller technically in most platforms.

Different from USB 1.x, USB 2.0 with a 480 MHz oscillator could operate at the so-called *high speed*, at 480 Mbit/s. Requests of USB 1.x and 2.0 devices are saved in the data structures for controllers, as shown in Figure 17.4b. Each periodic-frame-list entry is visited every 1024 μ -frame (i.e., 125 μ s time interval).

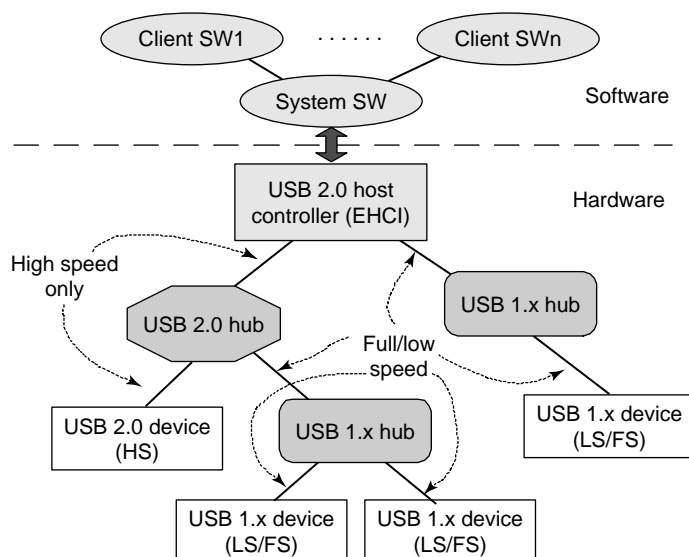


FIGURE 17.5 An example USB 2.0 topology.

Within each μ -frame, USB bandwidth is shared among requests, as shown in Figure 17.6b. Here SOF stands for the starting of the frame. Bandwidth marked as “Periodic” is for isochronous and interrupt transfers, and bandwidth marked as “NonPeriodic” is for control and bulk transfers. Periodic transfers could occupy up to 80% of the total bus bandwidth, and control transfers have higher priority than bulk transfers do. Bulk transfers are serviced in a best-effort fashion to utilize the remaining USB bandwidth. The allocation of the bus bandwidth depends on the implementation policy. Since the time frame for USB 1.x is in 1 ms (referred to as an m -frame), one m -frame is equal to 8 μ -frames, as shown in Figure 17.6a. Each request of a USB 1.x device is partitioned into eight subrequests (the partitioning is referred to as *split transaction* for the rest of this section) in eight consecutive μ -frames.

Although the USB specifications provide a bandwidth management mechanism (in which an isochronous request could specify the maximum number of bytes to be transferred for a device type in each μ -frame), little support is provided for other kinds of requests. Furthermore, even for isochronous requests, there is no way to specify the minimum number of bytes to be transferred for a device in each μ -frame. The bandwidth allocation for periodic and sporadic transfers is also done in a very coarse-grained fashion (such as a 80% upper bandwidth bound for periodic transfers). In other words, the USB specifications do not provide either an absolute or probability-based guarantee for the minimum bandwidth usage for different device requests. This observation underlies the motivation of this chapter. That is, how to propose a proper resource management and scheduling mechanism to guarantee the QoS requirements of different devices and to comply with USB specifications.

17.2.2 Overview

The purpose of this section is to propose a real-time USB device driver architecture and a set of scheduling methods to provide an integrated QoS service for periodic and sporadic requests. As discussed in Section 17.1.2, USB data transfer modes are classified into four categories: isochronous transfer, interrupt transfer, control transfer, and bulk transfer. Requests of different transfer modes are arranged in different data structures, as shown in Figure 17.4b. We propose to have a layer of real-time request scheduling being built inside the USB core, as shown in Figure 17.7, where data structures in Figure 17.4b are shown in a more simplified way. The responsibility of the real-time scheduler is to do admission control and to insert QHs and TDs of admitted requests in proper locations of their corresponding data structures to meet their QoS requirements.

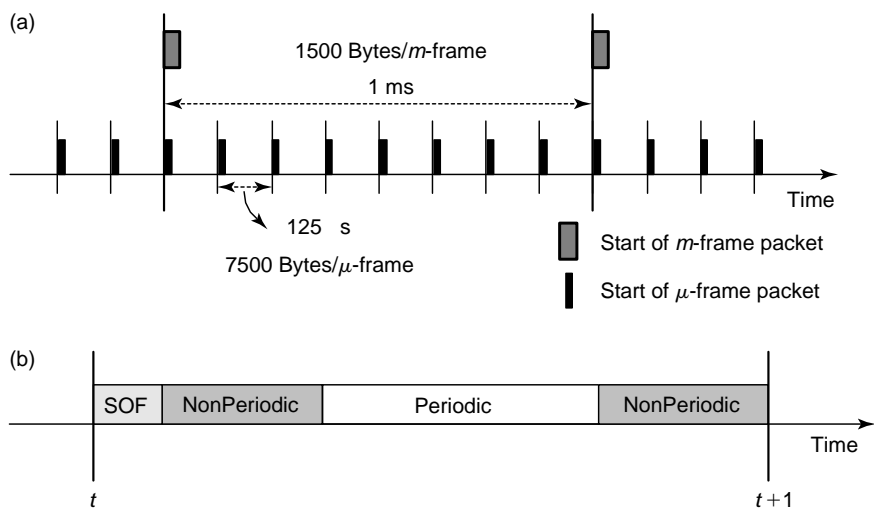


FIGURE 17.6 (a) The USB m -frame/ μ -frame time frame. (b) The bandwidth reservation of the USB m -frame/ μ -frame time frame.

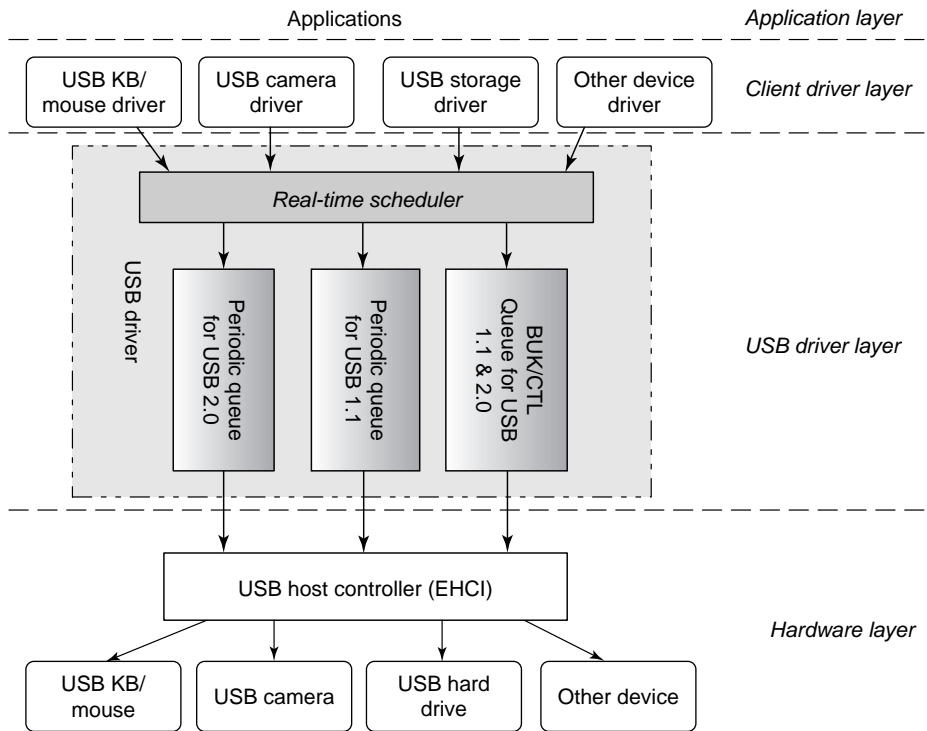


FIGURE 17.7 A real-time USB driver architecture.

To guarantee the QoS requirements of different devices and to better manage the USB bandwidth, two essential technical issues must be addressed: (1) an admission control policy and QoS guarantees for periodic and sporadic requests and (2) the scheduling of QHs and TDs with QoS considerations. Under the satisfaction of the QoS requirements for devices, we shall also maximize the utilization of the USB bandwidth. Note that we aim at providing a hard QoS guarantee for periodic requests and a soft QoS guarantee for sporadic requests in terms of their average response time and waiting time.

The rationale behind this chapter is to install a virtually fixed polling tree structure for USB bandwidth allocation with low runtime overheads. We propose a period modification policy to construct a polling tree for periodic requests based on their corresponding QoS requirements (in Section 17.2.3.1). The tree structure is revised only when a new device request is admitted, or when an existing device is closed. We present a split-transaction-based technique to handle the compatibility problem in QoS management for USB 1.1 and 2.0 devices, where the time interval of a data transfer frame for USB 1.x is 8 times of that for USB 2.0. An efficient schedulability test is presented for online admission control. The technical issues in the insertion of QHs and TDs in the polling tree are then further explored in Section 17.2.3.2. We consider the different frame intervals for USB 1.x and USB 2.0 in the insertion process. Implementation remarks on the approximation of polling periods are addressed in Section 17.2.3.3. When sporadic requests are considered, we propose an evaluation framework to derive the average waiting time for requests under the existing USB scheduling policy.

17.2.3 QoS Guarantees for Periodic Requests

17.2.3.1 A Period Modification Policy

The purpose of this section is to propose scheduling algorithms for the QoS guarantees of periodic requests, i.e., those for isochronous and interrupt transfers. This section first proposes a period modification policy

to insert QHs at proper nodes of the polling tree structure, as shown in Figure 17.4b. The idea is to insert all USB 2.0 QHs (and their TDs) and all USB 1.x QHs (and their TDs) at the root and nodes corresponding to the polling rate 2^3 , respectively. We shall then propose a schedulability test for admission control of USB requests for the QoS guarantees of isochronous and interrupt transfers. In Section 17.2.3.2, we shall propose a method to move QHs (and their TDs) down to nodes at lower levels of the polling tree for better USB bandwidth utilization.

For the purpose of USB bandwidth reservation, we include additional information regarding deadlines, service frequencies, and USB bandwidth requirements inside URB, where URB is a data structure used for the interchanging of control and data information between the USB core and device drivers. Since the USB core is also named as the USB driver (USB D) on Linux, USB D and the USB core are used interchangeably for the rest of this section. An abstracted real-time scheduler layer is proposed in the driver hierarchy, as shown in the Figure 17.7. The responsibility of the real-time scheduler is to communicate with the device driver layer (e.g., USB camera driver in Figure 17.7) and insert requests into the polling tree of USB D based on their QoS requirements. The real-time scheduler should also be responsible for the admission control of new QoS requests.

The location of the residing node for a QH (and its TDs) in the polling tree determines the service frequency of its corresponding requests, where a request often corresponds to several TDs. The polling tree, as shown in Figure 17.8, is a complete binary tree of 11 levels. The root is at the level 0, and the leaf nodes are at the level 10. There are 1024 leaf nodes in the polling tree. Let each leaf node have a corresponding unique number $X = b_n, b_{n-1}, \dots, b_0$ in a binary representation format (from right to left, for example). According to EHCI, leaf nodes should be visited in a round-robin fashion. When a leaf node is visited, all QHs (and their TDs) on the nodes in the path from the leaf node all the way to the root will be serviced with a μ -frame, i.e., 125 μ s. If they could not be handled within one μ -frame, the host controller will abort the process (i.e., the service of this path) and then jump to the next leaf node. Any aborted path is considered as an *overflowed* path. As a result, any QHs (and their TDs) at the node of the i th level will be serviced every $2^i \mu$ -frames. To have an even service over requests, we propose to visit leaf nodes in the order of their $S(X) = b_0, b_1, \dots, b_{n-1}, b_n$, where $X = b_n, b_{n-1}, \dots, b_0$ is its leaf number.

Consider a workload with n periodic requests $\{(b_1, p_1), (b_2, p_2), (b_3, p_3), \dots, (b_n, p_n)\}$ for devices, where b_i denotes the number of bytes to be transferred for every $p_i \mu$ -frames. The *period modification policy* is defined as follows: (1) For each USB 2.0 periodic request (b_i, p_i) , a corresponding QH is created at the root of the polling tree with a new period as one μ -frame (because it is now at the root) and a new number on the

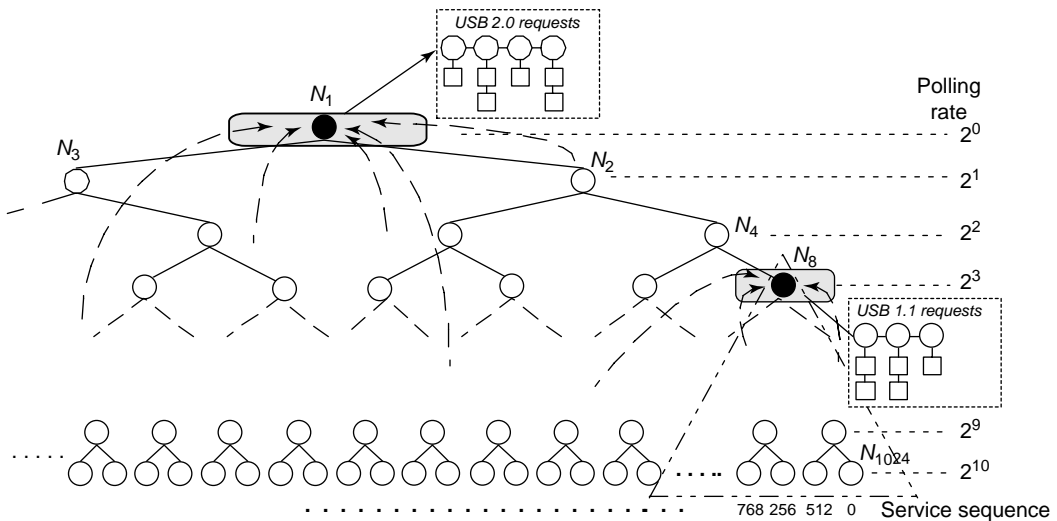


FIGURE 17.8 A polling tree for the USB scheduler of a USB 2.0 host controller.

bytes to be transferred in a period as $\lceil \frac{b_i}{p_i} \rceil$. (2) For each USB 1.1 periodic request (b_i, p_i) , a corresponding QH is created at the rightmost node of the level 3, i.e., N_8 in Figure 17.8. The request has a new period as 8 μ -frame (because it is now at the level 3) and a new number on the bytes to be transferred in a period as $\lceil \frac{b_i \times 8}{p_i} \rceil$. The TDs for the QH are created according to EHCI and the number of bytes to be transferred. Note that after the period modification policy is done, the USB 2.0 host controller will service QHs (and their TDs) at N_8 automatically and evenly in eight consecutive μ -frames, as shown in Figure 17.6a. Those requests at N_8 are for USB 1.x devices. Because of the behavior of a USB 2.0 host controller and the proposed $S(X)$ service order, the insertion of all USB 1.x requests at N_8 is equivalent to the even insertion of them over all nodes at the level 3. Such an approach is referred to as the *split-transaction-based approach* for the rest of this section. Since each USB 2.0 or 1.x request is assigned a smaller period in the period modification policy, the policy would introduce extra overheads because more packet headers and the ceiling calculation for the number on the bytes would be transferred in a new period.

For example, a request that corresponds to 32 bytes per 4 μ -frames could be modified as 8 bytes per μ -frame. The period modification policy inserts all USB 2.0 requests (i.e., their QHs and TDs) at the root, and the root node is serviced every μ -frame. Furthermore, the policy inserts all USB 1.1 requests (i.e., their QHs and TDs) at N_8 but has the requests evenly split over eight consecutive μ -frames. Because 7500 bytes could be transferred within one μ -frame, the schedulability test for a given set of n periodic requests $B = \{(b_1, p_1), (b_2, p_2), (b_3, p_3), \dots, (b_n, p_n)\}$ could be derived as follows:

$$\sum_{\text{USB2.0}(b_i, p_i) \in B} \left(\left\lceil \frac{b_i}{p_i} \right\rceil + O \right) + \sum_{\text{USB1.1}(b_j, p_j) \in B} \left(\left\lceil 8 \times \frac{b_j}{p_j} \right\rceil / 8 + O \right) \leq 7500 \quad (17.1)$$

In Formula 17.1, O denotes the protocol overheads of a USB payload. The overheads include packet preambles, packet headers, and some other necessary fields. To be more specific, the overheads are 9 bytes for isochronous transfers and 13 bytes for interrupt transfers. For example, consider an isochronous-transfer request with 50 bytes per 16 μ -frames. With the period modification policy, $(\lceil \frac{50}{16} \rceil + 9)$ bytes would be transferred within each μ -frame. In the next section, we shall propose an algorithm to further reduce the overheads.

17.2.3.2 A Workload Reinsertion Algorithm: Bandwidth Utilization Optimization

In the previous section, USB 2.0 and 1.x requests are inserted as QHs and TDs at the root and N_8 of the polling tree, respectively. A period modification policy with a split-transaction-based approach is presented. In this section, we shall start with the results of the period modification policy and then reinsert QHs (and their TDs) back to proper nodes in the polling tree. The objective is to minimize the protocol overheads and additional bandwidth requirements (i.e., the difference between $(\lceil \frac{b_i}{p_i} \rceil)$ and b_i in Formula 17.1).

Given a collection of admitted requests $T = \{(b_1, p_1), (b_2, p_2), \dots, (b_n, p_n)\}$ in terms of their QHs and TDs, we shall try to push QHs (and their TDs) at the root or N_8 down to the nodes of larger heights as much as possible. The further we push a QH (and its TDs) away from the root or N_8 , the lower the protocol overheads would be. There are two basic guidelines for this *workload reinsertion algorithm* (which will be presented in pseudocode later). The first guideline is that the destination node of an original QH (and its TDs) should not have a polling period longer than the period p_i of the corresponding request. For example, let a video camera demand a 4096-byte transfer per 8 μ -frames. If the data on the camera is not transferred to the host controller within an 8 μ -frames window, the data in the memory buffer of the camera might be overwritten by new data. The second guideline is that the workload reinsertion algorithm should meet the μ -frame requirement for each path visiting of a tree structure. Since only 7500 bytes could be transferred within every μ -frame, the total number of bytes for the services of all QHs (and their TDs) on a path should be no more than 7500.

We shall illustrate the idea behind the workload reinsertion algorithm and its relationship with the period modification policy by using the following example: to simplify the presentation, the protocol overheads are not shown in this example. Consider the insertions of QHs (and their TDs) for the following three USB 2.0

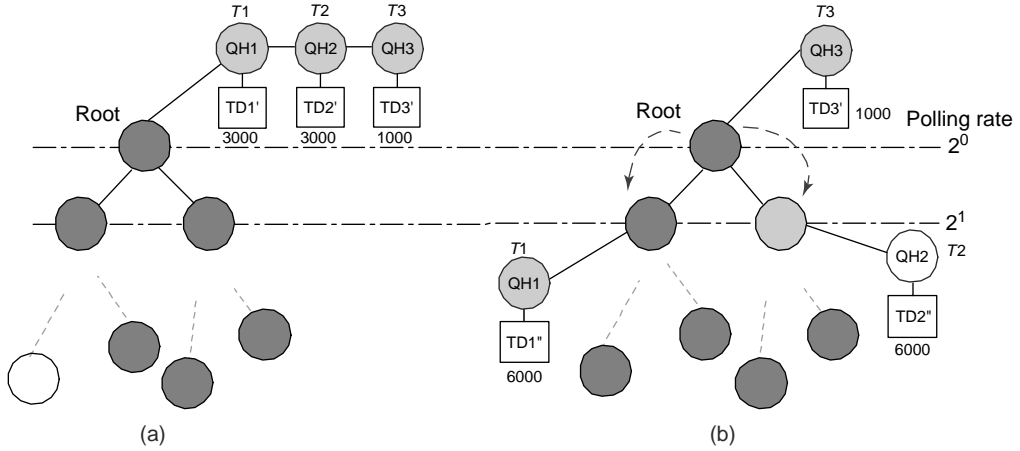


FIGURE 17.9 Period modification policy and workload reinsertion. (a) Results of the period modification policy. (b) Results of the workload reinsertion algorithm.

requests: $T = \{T_1 = (6000 \text{ bytes}, 2 \mu\text{-frames}), T_2 = (6000 \text{ bytes}, 2 \mu\text{-frames}), \text{ and } T_3 = (2000 \text{ bytes}, 2 \mu\text{-frames})\}$. With the period modification policy presented in Section 17.2.3.1, the workload is first revised, as shown in Figure 17.9a: $T' = \{T'_1 = (3000 \text{ bytes}, 1 \mu\text{-frame}), T'_2 = (3000 \text{ bytes}, 1 \mu\text{-frame}), \text{ and } T'_3 = (1000 \text{ bytes}, 1 \mu\text{-frame})\}$. The workload would pass the admission control because $3000 + 3000 + 1000 \leq 7500$ (see Formula 17.1). The workload reinsertion algorithm then tries to revise the QHs and the TDs of T'_1 and T'_2 to move them down to the nodes of the second level. When we want to push T'_3 down to any one of the nodes of the second level, we would notice that the new payload size will be 2000 bytes because nodes at the second level have a $2\text{-}\mu\text{-frame}$ polling period. As a result, it is not feasible to push T'_3 to any node of the second level because of the violation of the second guideline. As astute readers might notice, the workload reinsertion optimization problem is a combinational problem, which is apparently not an easy problem.

For the rest of this section, we shall present a greedy algorithm for workload reinsertion, referred to as Algorithm *WorkloadReInsertion*. The main idea for this greedy algorithm is as follows: All QHs are first classified as USB 1.x or 2.0 requests and then sorted in a decreasing order of their original periods, where each QH corresponds to a request. USB 1.x requests are handled first for workload reinsertion because USB 1.x requests that are involved with split transactions (due to the behavior of the host controller) have more restrictions in reinsertion. The reinsertion of USB 2.0 requests starts when the reinsertion of USB 1.x requests is done. Algorithm *WorkloadReInsertion* runs in a number of iterations for each type of requests. Within each iteration, the algorithm tries to push the QH with the largest original period that has not been considered so far down to a node of a larger height. Whenever a QH is considered to move down by one level, there are two choices: the right child node n_R or the left child node n_L . Let $S(n)$ denote the sum of the numbers of the bytes (plus the protocol overheads) for all of the QHs (and their TDs) hooked up at the node n and all of the nodes in the subtree with n as the root. That is, $S(n) = S(n_L) + S(n_R) + l_n$, where l_n denotes the sum of the numbers of the bytes (plus the protocol overheads) for all of the QHs (and their TDs) hooked up at the node n . The choice on the right or left child node should consider the requirements of USB 2.0 in servicing each path in a $\mu\text{-frame}$ and the split transaction requirement of USB 1.x requests. The heuristics of the algorithm is to prefer the child node with a smaller value of the function $S(n)$.

Consider two lists of pending requests R_1 and R_2 for Algorithm *WorkloadReInsertion*, where R_1 and R_2 are lists of USB 1.x and 2.0 requests, respectively. According to the period modification policy, QHs and their TDs of requests in R_1 and R_2 are created and inserted into $Q[1]$ and $Q[2]$, respectively (Steps 5 and 6). Note that we change the per-period workload of each request in R_1 to $\lceil (b_i/p_i) \rceil$ and the per-period workload of each request in R_2 to $\lceil 8(b_i/p_i) \rceil$ in these two steps, where all of QHs and TDs of requests in R_2 are at the root node, and those in R_1 are at N_8 . Steps 7–9 do the admission control based on Formula 17.1.

Algorithm 1 The Workload Reinsertion Algorithm.

```

1: WorkloadReInsertion( $R_1, R_2$ )
2:  /*  $R_1$ : a list of USB 1.x requests;
3:    $R_2$ : a list of USB 2.0 requests; */
4:  {
5:   Q[1] = the QHs and their TDs of requests in  $R_1$  according to the period modification policy
6:   Q[2] = the QHs and their TDs of requests in  $R_2$  according to the period modification policy
7:   if Q[1] and Q[2] together fail Formula 17.1 then
8:     return FAIL
9:   end if
10:  Sort all requests, i.e., QHs, in Q[1] and Q[2] in the decreasing order of their original periods;
11:  for  $k = 1; k \leq 2; k++$  do
12:     $Q = Q[k]$ 
13:    while Q is not empty do
14:      Let  $e_i = (b_i, p_i)$  be the first QH in Q
15:       $n \leftarrow \text{root}$ 
16:      for  $j = 1; j \leq 10; j++$  do
17:        if  $p_i < 2^j$  then
18:          break
19:        end if
20:         $p \leftarrow n$ ; /* previous node */
21:        if  $S(n_L) < S(n_R)$  then
22:           $n \leftarrow n_L$ ; /*  $n_L$ : left child of  $n$ ;  $n_R$ : right child of  $n$  */
23:        else
24:           $n \leftarrow n_R$ 
25:        end if
26:        Move  $e_i$  to node  $n$ 
27:        Adjust  $e_i$ 's QH and TDs
28:        if the revising could result in any overflowed path then
29:          /* Some paths cannot meet the  $\mu$ -frame servicing constraint */
30:          Move  $e_i$  to node  $p$ 
31:          break
32:        end if
33:         $S(p) \leftarrow S(p_L) + S(p_R) + l_p$  /* Update  $S(n)$  value of node  $n$  and previous node  $p$ ; */
34:         $S(n) \leftarrow S(n_L) + S(n_R) + l_n$ 
35:      end for /* End For  $j$  */
36:      Remove  $e_i$  from Q
37:    end while
38:  end for
39:  }

```

The algorithm then sorts all of the requests in each queue in the decreasing order of their original periods (Step 19). In Steps 11–32, the algorithm runs in iterations and tries to push the QH with the largest original period down to a node of a larger height (in each iteration). In the iteration (Steps 16–30), the algorithm always chooses the child node with a smaller $S(n)$ value to move. When a request (i.e., e_i) is moved down by one level, its per-period workload size is modified (Step 22) because the polling period is doubled. The algorithm has to check up whether the moving operation will result in any violation of the μ -frame constraint (Steps 23–27), i.e., the appearance of any overflowed path. The $S(n)$ values of nodes are modified accordingly if any request is moved to other nodes.

17.2.3.3 A Binary Encoding Algorithm for Period Modification: A Multiple-QH Approach

In the previous sections, a period modification policy and a workload reinsertion algorithm are proposed for the construction of the polling tree. Although the workload reinsertion algorithm tries to move QHs and their TDs to proper nodes in the polling tree, one request still has only one corresponding QH and resides at a node with a polling period being $2^i \mu$ -frames. The purpose of this section is to further improve the USB bandwidth utilization by offering more than one QH for each request such that the period in executing a request can be better approximated.

Given a polling period x , $\frac{1}{x}$ could be approximated by a sequence of numbers $(y_0, y_1, \dots, y_{10})$ such that $\frac{1}{x} \leq y = (\sum y_i \times \frac{1}{2^i})$, where y_i is 0 or 1. Given a polling period x (where $x \in [1, 1024]$), we shall find the smallest value for y that satisfies the following Formula:

$$f(x) = \left\{ y \mid \frac{1}{x} \leq \left(y = \sum_{i=0}^{10} \left(y_i \times \frac{1}{2^i} \right) \right) \text{ and } y_i = [0, 1] \right\} \quad (17.2)$$

Note that there can be only one unique sequence of numbers $(y_0, y_1, \dots, y_{10})$ for the smallest value for y that satisfies Formula 17.2. For each $y_i = 1$ for $i \in [0, 10]$, the corresponding TDs of a device should be inserted to a node that corresponds to $2^i \mu$ -frames polling period.

For example, suppose that a request needs to transfer X bytes per Y ms period. After the corresponding QH and TD are inserted at a node in the polling tree (in Section 17.2.3.2), we can apply the binary-based representation method presented here to further split the polling period into multiple polling periods. For the original period Y , it can be transformed into one unique sequence of numbers $(y_0, y_1, \dots, y_{10})$. Although we split a QH (and its TDs) into multiple QHs, there is no buffer overwriting problem at the device because the *de facto* polling period is still smaller than the original period. Suppose that a device requires to transfer 16 bytes per $96\text{-}\mu$ -frame polling period (i.e., $X = 16$ and $Y = 96$). The $96\text{-}\mu$ -frame period could be approximated by two polling periods: 128 and 256μ -frames. The period modification policy might assign the corresponding QH and TDs to a node with a $64\text{-}\mu$ -frames polling period. For further improvement on the protocol overheads, the QH can be split into two QHs on nodes with polling periods equal to 128 and 256μ -frames, respectively. Each of the QHs could have 16 bytes to transfer for its polling period. Note that the total data transfer rate is $16/(128 \times 125 \mu s) + 16/(256 \times 125 \mu s) (=1500) > 16/(96 \times 125 \mu s) (=1333.33)$.

A binary encoding algorithm for period modification could be done in a greedy way: the algorithm should first sort all of the requests in the polling tree in a nondecreasing order of their saving before and after a binary encoding of their periods. The algorithm then tries to insert QHs (and their TDs) of a request in the first-fit fashion by searching nodes in the subtree of the node assigned by the workload reinsertion algorithm. The insertion should not result in any overflowed path. If the insertion results in any overflowed path, then the binary encoding attempt of the request is aborted. The algorithm continues in an attempt for the binary encoding of the next request's period. To limit the complexity of the binary encoding algorithm for runtime usage, we shall restrict the number of y_i in encoding. For example, when the number of y_i is only two, then this binary encoding algorithm could be very efficient (however, at the cost of the wasting of potential USB bandwidth). The decision on the number of y_i for encoding is basically a trade-off between the runtime efficiency and the approximation precision/bandwidth utilization.

17.2.3.4 QoS Guarantees for USB 1.x Subsystem

The previous sections discussed many methods to support QoS guarantees for periodic requests. In Section 17.2.3.1, a *Period Modification Policy* inserts all requests of USB 2.0 and USB 1.x at the root and nodes corresponding to the polling rate 2^3 , respectively. Then it proposed a schedulability test for admission control of USB requests for the QoS guarantees of isochronous and interrupt transfers. In Section 17.2.3.2, a *Workload Reinsertion Algorithm* is used to minimize the protocol overheads and additional bandwidth requirements for better USB bandwidth utilization. Section 17.2.3.3 further improves the USB bandwidth

utilization by offering more than one QH for each request such that the period in executing a request can be better approximated. All of the methods can apply to the USB 1.x subsystem.

For the purpose of USB 1.x bandwidth reservation, URB is a data structure used for the interchanging of control and data information between the USB core and device drivers. In Figure 17.2, it shows the USB driver architecture with USB 1.x/2.0. The USB core is named as the USB driver (USBD). The real-time scheduler is in USBD such that device drivers would call (or send messages to) the new layer, instead of following the original procedure. The real-time scheduler determines how TDs of a request are inserted into the ED of the device to meet the QoS requirements of devices. Note that the real-time scheduler might reject requests for the QoS requirements of devices.

USBD collects the upper-level driver's requests (in terms of URB, as illustrated in the previous sections) and allocates USB bandwidth for each request. Different drivers might need different polling periods to transfer request messages between the peripherals and the corresponding device drivers. The Linux USB OHCI driver restricts periodic service requests (for interrupt and isochronous devices) to a set of fixed periods (e.g., 1, 2, 4, ..., and 32 ms). Note that the tree structure of the TD lists in Figure 17.4a determines the frequencies of the traversing of nodes on paths to the root. If a device has a corresponding node on a path to the root, then the device is serviced in the 1 ms corresponding to the path. Because there are 32 leaf nodes, as shown in Figure 17.4a, the tree has 63 internal/leaf nodes. In other words, the structure provides the host controller driver 63 different nodes to hook up EDs.

A polling tree of the 63 nodes provides an abstraction in the handling of isochronous and interrupt transfers. There are five levels in the polling tree and 32 paths from leaf nodes to the root node, as shown in Figure 17.10. The visiting order to each leaf node is marked as the service sequence under the tree in Figure 17.10, where a visiting occurs for every 1 ms time frame. An OHCI host controller travels through each path from a leaf node to the root node of the corresponding polling tree. A node could have a list of EDs, and an ED could have a list of TDs, as shown in Figure 17.3a. An OHCI host controller processes pending TDs on its way from a leaf node to the root node. The handling of pending TDs on each path should not be more than 1 ms; otherwise, the host controller will abort the path and then continue its work to the next path according to the service sequence. Any aborted path is considered as an *overflowed* path. As a result, any nodes at the i th level (counted from the root node) are serviced at every 2^i ms.

Consider a workload with n periodic requests $\{(b_1, p_1), (b_2, p_2), (b_3, p_3), \dots, (b_n, p_n)\}$ for devices, where b_i denotes the number of bytes to be transferred for every p_i ms. Before we introduce the admission control policy, we first introduce a period modification policy to move and modify the ED of a request at a node to the corresponding ED at the root node: when the ED of a request is moved from a node to the root node, extra protocol overheads will occur, and the ED should be revised to fit the polling periods of the former node and the root node. However, besides the protocol overheads, when the number b_i of bytes per transfer cannot divide the original period p_i , then further overheads occur, i.e., an additional bandwidth requirement of $((\lceil \frac{b_i}{p_i} \rceil) p_i - b_i)$ bytes per p_i time units.

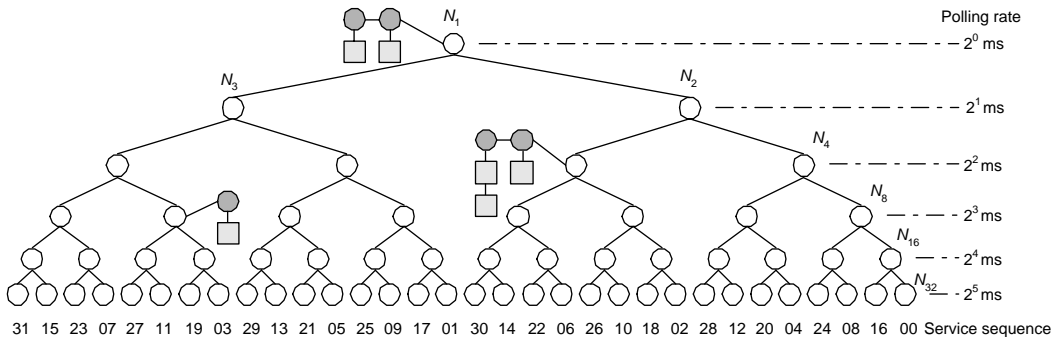


FIGURE 17.10 A polling tree for the USB scheduler of a USB 1.x host controller.

The admission control policy is as follows: because the period modification policy moves all EDs to the root node, and the root node is serviced for every 1 ms, all requests are schedulable if the services of all EDs could be done within 1 ms. Because only 1500 bytes could be transferred within every 1 ms under USB 1.1, the total number of bytes for the services of all EDs should be no more than 1500. That is,

$$\sum_{i=1}^n \left(\left\lceil \frac{b_i}{p_i} \right\rceil + O \right) \leq 1500 \quad (17.3)$$

In Formula 17.3, O denotes the protocol overheads of a USB payload. Specifically, the overheads are 9 bytes for isochronous transfers and 13 bytes for interrupt transfers. The term $\lceil \frac{b_i}{p_i} \rceil$ denotes the payload data size after the period modification. For example, consider an ED of an isochronous transfer, which represents 50-byte transfer per 16 ms. After the period modification policy, $(\lceil \frac{50}{16} \rceil + 9)$ bytes must be transferred for corresponding the ED within 1 ms.

In addition, the binary encoding algorithm in Section 17.2.3.3 could be applied for USB 1.x subsystem. Because the USB 1.x host controller could support a polling tree with five levels, the Formula 17.2 could be easily rewritten as follows for USB 1.x:

$$f(x) = \left\{ y \mid \frac{1}{x} \leq \left(y = \sum_{i=0}^5 \left(y_i \times \frac{1}{2^i} \right) \right) \text{ and } y_i = [0, 1] \right\} \quad (17.4)$$

17.2.4 Probabilistic QoS Analysis for Sporadic Transfers

This section aims at proposing an evaluation method for the probabilistic QoS analysis of bulk transfers. Given the arrival distributions of bulk transfers, we could derive the average waiting time of a request and thus provide a probabilistic QoS analysis for bulk transfers.

Although sporadic requests include bulk and control transfers, we should focus our QoS guarantee discussions on bulk transfers because control transfers are usually used only in the initialization phase for USB devices. Because USB 1.x and USB 2.0 use similar mechanism to deal with the sporadic requests, we should use USB 2.0 as the example to explain probabilistic QoS analysis for sporadic transfers.

In USB 2.0, the QHs (and their TDs) of sporadic requests that correspond to bulk transfers are linked in the AsyncList-Queue-List, as shown in Figure 17.4b. They are serviced in a round-robin fashion with a predetermined order, where lists serviced first are considered having higher priorities: A *transaction attempt* by the host controller is defined as a service of a TD. The host controller first executes a transaction attempt on the TD at the head of the highest-priority list pointed by the corresponding QH (referred to as a *QH list*). After the transaction attempt is done, the corresponding TD is removed from the QH list, and the host controller moves to the QH list with the second highest priority. The host controller executes a transaction attempt over the TD at the head of each QH list one by one and then goes back to the highest-priority QH list again when it reaches the lowest-priority nonempty QH list. Note that TDs are inserted in QH lists when their corresponding requests arrive.

We propose to revise the above scheduling methodology in terms of a size limitation on TDs. We shall first derive the mean waiting time and mean response time for TDs without any size limitation: that is, TDs could have a wide range of fluctuations with any restriction. The behavior of sporadic services for bulk transfers can be modeled as a limited polling service system, where polling is a way of serving requests in a cyclic order with generally nonzero switch-over times [17]. Figure 17.11 provides an illustration of a polling system for EHCI. In the figure, there are N QH lists serviced by EHCI, where each QH list is a list of TDs with the same upper bound on the workload servicing time and has the same arrival distribution. The service-switching time from one queue to another is defined as a *reply interval* (also named as *switch-over time* or *walking time* in the literature).

Let each QH list with a virtually infinite storage capacity have an independent request arrival stream at a rate λ . $B^*(s)$, b , and $b^{(2)}$, respectively, denote the LST (Laplace–Stieltjes Transform) of the distribution function, mean, and second moment for the service time of EHCI. That is, $b = -B^{*'}(0)$ and

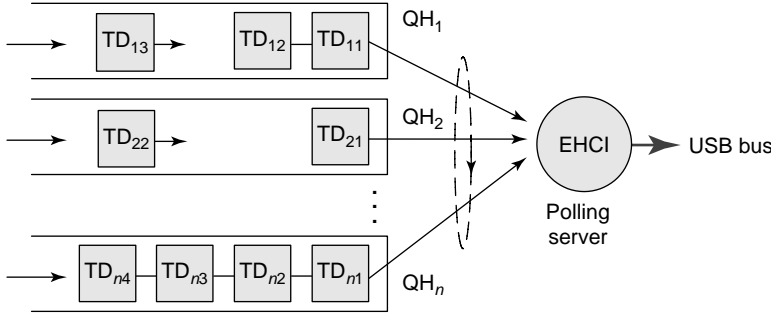


FIGURE 17.11 A polling system for sporadic transfers.

$b^{(2)} = B^{*''}(0)$. Within a finite reply interval, EHCI moves from a transaction attempt of a QH list to the next nonempty QH list. Let $R^*(s)$, r , and δ^2 , respectively, denote the LST of the distribution function, mean, and variance for the reply interval. That is, $r = -R^*(0)$ and $\delta^2 = R^{*''}(0) - r^2$. Let $\tau_i(m)$ be the polling instant for the i th QH list in the m th polling cycle, and $L_i(t)$ the number of TDs in the i th QH list at time t . Define the joint generation function $F_i(z_1, z_2, \dots, z_N) = E[\prod_{j=1}^N z_j^{L_j(\tau_i(m))}]$ for $[L_1(t), L_2(t), \dots, L_N(t)]$ at $t = \tau_i(m)$, i.e., at the instant when EHCI becomes available to service bulk transfers in the i th QH list. The marginal generation function for $L_i(t)$ at time $\tau_i(m)$ is defined as $F_i(z) = E[\prod_{j=1}^N z_j^{L_j(\tau_i(m))}] = F_i(1, \dots, 1, z, 1, \dots, 1)$, where z is the i th argument in $F_i(1, \dots, 1, z, 1, \dots, 1)$. We have $F_{i+1}(z_1, z_2, \dots, z_N) = R_i[\prod_{j=1}^N P_j(z_j)] F_i(z_1, z_2, \dots, z_{i-1}, 0, [\prod_{j=1, (j \neq i)}^N P_j(z_j)], z_{i+1}, \dots, z_N)$, where $P_j(z_j)$ is the probability at the instant when the i th QH list is polled.

Let $f_i(j)$ be the mean number of requests at the i th QH list when the i th QH list is polled. We have $f_i(j) = \frac{\partial F_i(z_1, z_2, \dots, z_N)}{\partial z_j} \big|_{z_1 = z_2 = \dots = z_N = 1}$. Note that, if L_i^* denotes the number of requests at the i th QH list when it is polled, then $E[L_i^*] = f_i(i)$. We shall first derive the average length of a QH list and then provide the average waiting time and the response time for a sporadic request as follows:

Lemma 17.1

If the generation function for L_i , denoted by $Q_i(z)$, is given by the time average of $z^{L_i(t)}$ over the regenerative cycle, then the mean number of requests in each QH list at the time of a request departure from that QH list is $\lambda E[W] + \lambda b = Q_i'(1)$.

Proof

Given the generation function $Q_i(z) = E[z^{L_i(t)}]$, the relationship between $F_1(0, 1, \dots, 1)$ and $Q_i(z)$ could be formulated as follows [8]: note that since each QH list is identical in its characteristics, we shall work on the proof for the first QH list directly.

$$Q_1(z) = \frac{F_1(z, 1, \dots, 1) - F_1(0, 1, \dots, 1)}{z[1 - F_1(0, 1, \dots, 1)]} B^*(\lambda - \lambda z) \quad (17.5)$$

where $F_1(0, 1, \dots, 1)$ is the probability for the first QH list being empty when it is polled. Based on Formula 17.5, we could have the following formula:

$$Q_1'(1) = \frac{\left[\frac{dF_1(z, 1, \dots, 1)}{dz} \right]_{z=1}}{1 - F_1(0, 1, \dots, 1)} - 1 + \lambda b \quad (17.6)$$

The objective is to find the mean request waiting time $E[W]$, where $E[W] = -W^{*'}(0)$, and $W^*(s)$ is the LST of the distribution function for the request waiting time. Since the requests left in the first QH list

when a specific request of the list is done are those that arrive when the request is in the system, we have the following relationship:

$$W^*(\lambda - \lambda \cdot z)B^*(\lambda - \lambda \cdot z) = Q_1(z) \quad (17.7)$$

By differentiating each side of Formula 17.7, we have the following relationship:

$$\lambda E[W] + \lambda b = Q'_1(1) \quad (17.8)$$

which is the mean number of requests in each QH list at the time of a request departure from the QH list.

Lemma 17.2

The mean waiting time for a request in each QH list is

$$E(W) = \frac{\delta^2}{2r} + \frac{N[\lambda b^{(2)} + r(1 + \lambda b) + \lambda \delta^2]}{2[1 - N\lambda(r + b)]}$$

Proof

First let us derive the relation between $F_i(z_1, z_2, \dots, z_N)$ and $F_{i+1}(z_1, z_2, \dots, z_N)$. To express $F_{i+1}(z_1, z_2, \dots, z_N)$ in terms of $F_i(z_1, z_2, \dots, z_N)$, we note that $[L_1(t), L_2(t), \dots, L_N(t)]$ at $t = \tau_{i+1}(m)$ is $[L_1(t), L_2(t), \dots, L_N(t)]$ at $t = \tau_i(m)$ plus the number of arrivals during a service time and a reply interval. The following formula could be derived:

$$F_{i+1}(z_1, z_2, \dots, z_N) = R^* \left[\sum_{j=1}^N (\lambda - \lambda \cdot z_j) \right] \cdot \left\{ B^* \left[\sum_{j=1}^N (\lambda - \lambda \cdot z_j) \right] \frac{1}{z_j} [F_i(z_1, z_2, \dots, z_N) - F_i(z_1, \dots, z_{i-1}, 0, z_{i+1}, \dots, z_N)] + F_i(z_1, \dots, z_{i-1}, 0, z_{i+1}, \dots, z_N) \right\} \\ i = 1, 2, \dots, N \quad (17.9)$$

Let $z_k = z$ and $z_1 = \dots = z_{k-1} = z_{k+1} = \dots = z_N = 1$ in Formula 17.9. We have the following formula:

$$F_{i+1}(1, \dots, 1, z, 1, \dots, 1) = \begin{cases} R^*(\lambda - \lambda \cdot z) \cdot \{B^*(\lambda - \lambda \cdot z) \frac{1}{z} [F_i(1, \dots, 1, z, 1, \dots, 1) - F_i(1, \dots, 1, 0, 1, \dots, 1)] + F_i(1, \dots, 1, 0, 1, \dots, 1)\}, & k = i \\ R^*(\lambda - \lambda \cdot z) \cdot \{B^*(\lambda - \lambda \cdot z) \frac{1}{z} [F_i(1, \dots, 1, z, 1, \dots, 1) - F_i(1, \dots, 1, 0, 1, \dots, 1)] + F_i(1, \dots, 1, 0, 1, \dots, 1)\}, & k \neq i \end{cases} \quad (17.10)$$

where z and 0 appear in the k th and i th positions of F_{i+1} and F_i , respectively. By differentiating each side of Formula 17.10 with respect to z and having $z = 1$, we have the following formula:

$$F_1(0, 1, \dots, 1) = \frac{1 - N\lambda(r + b)}{1 - N\lambda b} \quad (17.11)$$

Furthermore, by differentiating Formula 17.9, the mean number of requests at the first QH list when it is polled is shown as follows:

$$\left[\frac{dF_1(z, 1, \dots, 1)}{dz} \right]_{z=1} = \frac{N\lambda^2(r^2 + \delta^2) + 2N\lambda r}{2[1 - N\lambda(r + b)]} + \frac{N\lambda r[2N\lambda^2rb - (N + 1)\lambda r + N\lambda^2b'']}{2(1 - N\lambda b)[1 - N\lambda(r + b)]} \quad (17.12)$$

By replacing the counterparts of Formulas 17.11 and 17.12 in Formula 17.6, and based on Formula 17.8, the mean request waiting time for a request can be derived as follows.

$$E(W) = \frac{\delta^2}{2r} + \frac{N[\lambda b^{(2)} + r(1 + \lambda b) + \lambda \delta^2]}{2[1 - N\lambda(r + b)]} \quad (17.13)$$

Corollary 17.1

The mean response time $E(T)$ for a request is

$$E(T) = E[W] + b = \frac{\delta^2}{2r} + \frac{N[\lambda b^{(2)} + r(1 + \lambda b) + \lambda \delta^2]}{2[1 - N\lambda(r + b)]} + b \quad (17.14)$$

Proof

The correctness of this corollary directly follows from Lemma 17.2.

The scheduling methodology for bulk transfers could be further revised to have a fixed size limitation on TDs so that bulk transfers could be serviced in a more “fair” fashion.

17.3 Summary

As modern operating systems become more and more modularized, the duty of resource allocation is now often shared among the kernel and subsystems. Exploring the idea and implementations of resource reservation over USB bus interfaces are major topics in this chapter. This chapter can be summarized as follows.

This chapter proposes the QoS-based system design for USB 1.x/2.0 device management. It addresses scheduling issues related to data transfers of USB 1.x/2.0 devices. An integrated real-time driver architecture that complies with the USB 1.x/2.0 specifications, and a bandwidth reservation algorithm that partitions available USB bandwidth among devices that need different attention from the system are proposed. A period modification approach and the corresponding schedulability test are also presented to provide guaranteed QoS services to isochronous and interrupt requests that are periodic. We then propose a workload reinsertion algorithm and a binary encoding algorithm to reduce protocol overheads in request insertions for the data structure dedicated for the USB host controllers. In particular, the binary encoding algorithm could split a request into a number of subrequests and insert them at virtually any node level in the polling tree, and better USB bandwidth utilization could be achieved. For best-effort requests, such as that for bulk transfers, a real-time scheduling algorithm with a probabilistic performance analysis could give you the average waiting time of each device.

References

1. B. Adelberg, H. G. Molina, and B. Kao, Emulating Soft Real-Time Scheduling Using Traditional Operating Systems Schedulers, *IEEE 14th Real-Time Systems Symposium*, Dec. 1994.

2. S. Childs and D. Ingram, The Linux-SRT Integrated Multimedia Operating Systems: Bring QoS to the Desktop, *IEEE 2001 Real-Time Technology and Applications Symposium*, 2001.
3. S. H. Hong and W. H. Kim, Bandwidth Allocation in CAN Protocol, *Proceedings of the IEEE Control Theory and Applications*, Jan. 2000.
4. Intel, *Universal Host Controller Interface (UHCI) Design Guide, Revision 1.1*. Intel, 1996.
5. Intel, *Enhanced Host Controller Interface Specification for Universal Serial Bus*. Intel, 2002.
6. A. Juvva and K. Rajkumar, Real-Time Filesystems Guaranteeing Timing Constraints for Disk Accesses in RT-Mach Molano, *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 155–165, Dec. 1997.
7. K. A. Kettler, J. P. Lehoczky, and J. K. Strosnider, Modeling Bus Scheduling Policies for Real-Time Systems, *IEEE Real-Time Systems Symposium*, Dec. 1995.
8. P. J. Kuehn, Multiqueue Systems with Nonexhaustive Cyclic Service, *Bell System Technical Journal*, vol. 58, pp. 671–698, Mar. 1979.
9. T. W. Kuo and A. K. Mok, Load Adjustment in Adaptive Real-Time Systems, *IEEE 12th Real-Time Systems Symposium*, Dec. 1991.
10. C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *JACM*, 20(1), pp. 46–61, Jan. 1973.
11. C. W. Mercer and R. Rajkumar, An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management, *IEEE Real-Time Technology and Applications Symposium*, May 1995.
12. A. K. Mok, *Fundamental Design Problems for the Hard Real-Time Environment*, PhD thesis, MIT Ph.D. Dissertation, Cambridge, MA, 1983.
13. D. M. Natale, Scheduling the CAN Bus with Earliest Deadline Techniques, *The 21st IEEE Real-Time Systems Symposium*, Nov. 2000.
14. T. Nolte, M. Sjodin, and H. Hansson, Server-Based Scheduling of the CAN Bus, *Proceedings of the Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 169–176, Sept. 2003.
15. S. Perez and J. Vila, Building Distributed Embedded Systems with RTLinux-GPL, *Proceedings of the Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 161–168, Sept. 2003.
16. H. Sato and T. Yakoh, A Real-Time Communication Mechanism for RTLinux, *Conference of the IEEE on the 26th Annual Industrial Electronics Society*, vol. 4, 2000.
17. H. Takagi, *Analysis of Polling System*, MIT Press, Cambridge, MA, USA, 1986.
18. USB, *Open Host Controller Interface Specification for USB*. Compaq, and Microsoft, and National Semiconductor, 1996.
19. USB, *Universal Serial Bus Specification, Revision 1.1*. Compaq, and Intel, and Microsoft, and NEC, 1998.
20. USB, *Universal Serial Bus Specification, Revision 2.0*. Compaq, and Hewlett-Packard, and Intel, and Lucent, and Microsoft, and NEC, and Philips, Apr. 2000.
21. V. Yodaiken, The RT-Linux Manifest, *Department of Computer Science Socorro NM 87801*, 2001.

18

Reference Middleware Architecture for Real-Time and Embedded Systems: A Case for Networked Service Robots

	18.1	Introduction	18-1
	18.2	Robot Middleware Requirements	18-2
		Device Abstraction and Component Models • Dynamic Reconfigurability • Resource Frugality for Scalability • Real-Time and QoS Capabilities	
	18.3	Reference Robot Middleware Architecture	18-4
		General Robot Hardware Platform • Motivations for RSCA • Overall Structure of RSCA • RSCA Core Framework • Assessment of RSCA	
Saehwa Kim	18.4	Future Challenges of Robot Middleware	18-10
Seoul National University		Performance Optimization via New Metrics • Streaming Support for High-Volume Data • Domain-Specific Abstractions	
Seongsoo Hong	18.5	Conclusions	18-12
Seoul National University			

18.1 Introduction

Convergence between computers and communications technologies has brought about strong commercial trends that have led to widespread deployment of distributed computing systems. These trends include the advent of e-commerce, integrated corporate information infrastructure, and web-based entertainment, just to name a few. Middleware technology has been extremely successful in these areas since it has solved many difficulties that developers faced in constructing distributed computing systems.

Middleware technology is ever expanding its frontier based on its success in the information infrastructure domain. It is thus not so strange to see middleware technology being adopted even in rather extraordinary domains such as automobile and robot industries. Unfortunately, the straightforward adoption of middleware technology to real-time and embedded systems may well lead to a failure since they are subject to severe nonfunctional constraints such as real-time guarantee, resource limitation, and fault-tolerance. In practice, most current middleware products are of only limited use in real-time and embedded systems [3].

A networked service robot is representative of contemporary real-time and embedded systems. It often makes a self-contained distributed system, typically composed of a number of embedded processors, hardware devices, and communication buses. The logistics behind integrating these devices are dauntingly complex, especially if the robot is to interface with other information appliances [5]. Thus, middleware should play an essential role in the robotics domain.

To develop commercially viable middleware for a networked service robot, it is very important to explicitly elicit the unique requirements of real-time and embedded middleware and then come up with technologies that address them. These pose difficult technical challenges to middleware researchers. In this chapter, we attempt to propose a reference middleware architecture for real-time and embedded systems using robotics as a domain of discussion. To do so, we first identify the general requirements of the robot middleware and then present a specific middleware architecture called the robot software communications architecture (RSCA) that satisfies these requirements.

Many technical challenges still remain unresolved in real-time and embedded middleware research. These challenges have something to do with new performance optimization metrics, effective data streaming support, and domain-specific abstractions. For instance, real-time and embedded middleware is subject to different types of performance optimization criteria since a simple performance metric alone cannot capture a wide variety of requirements of commercial embedded systems. Instead, performance–power–cost product should be considered. Also, real-time and embedded middleware should provide right abstractions for developers. Inappropriate middleware abstraction may hurt both system performance and modeling capability.

While real-time and embedded systems industry desperately wants to adopt middleware technology, current middleware technology fail to fulfill this demand mainly because it ignores important practical concerns such as those mentioned above. We address them as future challenges before we conclude this chapter.

18.2 Robot Middleware Requirements

The emergence of an intelligent service robot has been painfully slow even though its utility has been evident for a long time. This is due in part to the variety of technologies involved in creating a cost-effective robot. However, the ever-falling prices of high-performance CPUs and the evolution of communication technologies have made the realization of robots' potential closer than ever. It is now important to address the software complexity of the robot. The aim of the robot industry is to improve the robot technology and to facilitate the spread of robots' use by making the robot more cost-effective and practical. Specifically to that end, it has been proposed that the robot's most complex calculations be handled by a high-performance remote server, which is connected via a broadband communication network to the robot. For example, a robot's vision or navigation system that needs a high-performance micro-processor unit (MPU) or digital signal processor (DSP) would be implemented on a remote server. The robot would then act as a thin client, which makes it cheaper and more lightweight.

Clearly, this type of distributed system demands a very highly sophisticated middleware to make the logistics of such a robot system manageable. There has been a great deal of research activity in robot middleware area and yet there is still no current middleware that has garnered wide industrial approval. This gives rise to specific requirements that are yet to be fulfilled.

In this section, we investigate the requirements that the middleware developers must satisfy to successfully design and deploy middleware in networked service robots. These requirements are (1) provision of device abstraction and component models, (2) dynamic reconfigurability, (3) resource frugality for scalability, and (4) real-time and quality of service (QoS) capabilities.

18.2.1 Device Abstraction and Component Models

A robot, like an automotive and avionic system, is an electromechanical system that is composed of a wide variety of hardware and software modules that have distinct idiosyncrasies. These modules work closely

with each other and have complex interaction patterns. For example, vision, navigation, and motion control modules are the most essential modules in the robot. A motion control module consists of a motor, an encoder, and a DSP. A motion controller task running on the DSP reads in position data from the encoder, runs a motion control algorithm, and then produces an actuation command to the motor. A navigation module reads in map data from the vision module, computes its route, and then sends a motion control command to the motion control module. The vision module reads in raw vision data from a robot's stereo camera sensors and then process the data to update the map.

In contrast, unlike an automotive and avionic system, a robot system makes an open platform that has to accommodate independently developed applications and execute them to enrich its features for improved user services. Such robot applications, even though they rely on services provided by the lower-level robot modules, should be made unaware of low-level devices inside the robot. As such, the robot middleware must be able to offer a sufficient level of device encapsulation to robot application developers by ensuring that the robot is device-agnostic. Specifically, the robot middleware is required to provide a device abstraction model that can effectively encapsulate and parameterize wide variety of devices used inside the robot. Currently, only few middleware products offer such a device abstraction.

As previously mentioned, a networked service robot is not only a self-contained distributed system by itself but also a part of an overall distributed system including remote servers and various home network appliances. For example, as explained before, computation-intensive modules such as vision and navigation modules could be strategically allocated to a remote server and a robot would collaborate with the remote server, a local cache, and even a residential gateway to obtain motion control commands from the remote navigation module. In such an environment, application software needs to be designed to reflect the distributed nature of a robot's operating environment (OE).

The sheer complexity of the robot software in a distributed environment requires a middleware-supported component technology as well as procedural and object-oriented communication middleware. While the communication middleware based on remote procedure calls and remote method invocations are effective in hiding heterogeneity in network protocols, operating systems, and implementation languages in distributed computing, it cannot support the plug and play of software modules. Recently, advanced software component models and their associated middleware products facilitate the easy integration of independently developed software modules and their life-cycle management. Thus, robot application software should be constructed according to a component-based software model and the middleware of the robot should support this model along with the reconfigurable distributed computing.

The middleware-supported component technology offers another benefit in the networked service robot. It serves as seamless glue between the device abstraction layer and any upper-level software layers. The device abstraction model of the robot yields logical devices that play the role of device drivers for real hardware devices. These logical devices can be transparently abstracted into components using a component wrapper. Therefore, they can be smoothly incorporated into component-based robot software. Surely, there are several differences between logical device components and pure software components. First, logical device components are preinstalled while the robot is manufactured. Second, logical devices that abstract processing units such as DSP, field-programmable gate array (FPGA), and general purpose processor (GPP) may load and execute other software components. Third, logical device components directly interface with their underlying hardware devices just like device drivers.

18.2.2 Dynamic Reconfigurability

A networked service robot should be able to adapt to various needs of users and varying operating conditions throughout its lifetime. A recent industry study shows that the robot needs a new set of features every three weeks since users are quickly accustomed to existing applications of the robot [4]. Also, the networked service robot is susceptible to network connectivity problems since part of its intelligence is implemented through the Internet. If its remote navigation server is temporarily unreachable, the robot should be able to reconfigure its software system to utilize its local navigation module with lesser precision

and get the system back later when the server is available again. Thus, dynamic reconfigurability becomes a very important requirement of robot middleware.

Robot middleware can support dynamic reconfigurability at three levels: system-level, application level, and component level. System-level reconfigurability is the ability to dynamically configure preinstalled applications and currently running applications without any interruption to the robot operation. With the system-level dynamic reconfigurability, the robot middleware can install/uninstall, instantiate/destroy, and start/stop diverse robot applications while the robot is operating. Application-level reconfigurability is the ability to dynamically configure the properties and structures of an application. Such a reconfiguration can be used to increase the system reliability by replicating components and dynamically replacing faulty components. Finally, component-level configurability denotes the ability to configure the properties of individual components and dynamically select one of many alternative implementations of a component.

18.2.3 Resource Frugality for Scalability

While the current middleware technology offers significant benefits to developers, real-time and embedded systems still cannot afford to enjoy all of these benefits if the thick layers of middleware incur too much runtime overhead and require too much hardware resource. From time to time, a new middleware standard comes out into the world after it solves most of the requests from the involved parties. This seems unavoidable since it is the role of middleware to embrace heterogeneity in distributed computing.

However, it is quite often the case that such middleware is ignored by real-time and embedded systems industry. In automotive industry, for instance, 16-bit microcontrollers with 64 KB memory are still easily found in practice. Thus, to warrant the success of middleware in real-time and embedded systems industry, it is inevitable to design middleware specifically customized to a given application domain. One noticeable example is AUTOSAR (automotive open system architecture) from the automotive industry [2]. AUTOSAR is the industry-led middleware standard for automotive electronic control units (ECU). It has already been widely adopted by many industry leaders. One of the reasons for its success lies in a very compact and efficient runtime system that is obtained through purely static binding of components. Unlike many other middleware architectures, AUTOSAR does not support dynamic binding of components.

18.2.4 Real-Time and QoS Capabilities

A networked service robot is heavily involved in real-time signal processing such as real-time vision and voice processing. Furthermore, most of the services provided by a networked service robot incorporate critical missions with strict deadlines. It is difficult to meet the real-time constraints of a robot application solely by application-level techniques without support from the underlying middleware and operating system. Particularly, the robot middleware has to provide support for real-time guarantees and QoS so that robot application developers can exploit these services.

The most essential real-time and QoS capabilities that the robot middleware should provide include abstractions for static and dynamic priority distributed scheduling, prioritized communications, admission control, resource reservation, and resource enforcement.

18.3 Reference Robot Middleware Architecture

We have identified the requirements of the robot middleware to use them in deriving a reference middleware architecture for a networked service robot. In this section, we present one specific example of such middleware called the RSCA. We argue that the RSCA demonstrates how typical robot middleware is organized. To help readers understand it better, we first show the typical internal hardware structure of a networked service robot on which it is running, focusing on its heterogeneity and distributed nature. We then show the overall structure of the RSCA and explain in detail its component management layer that is called a core framework (CF). After that, we show how the RSCA meets the requirements identified in the previous section.

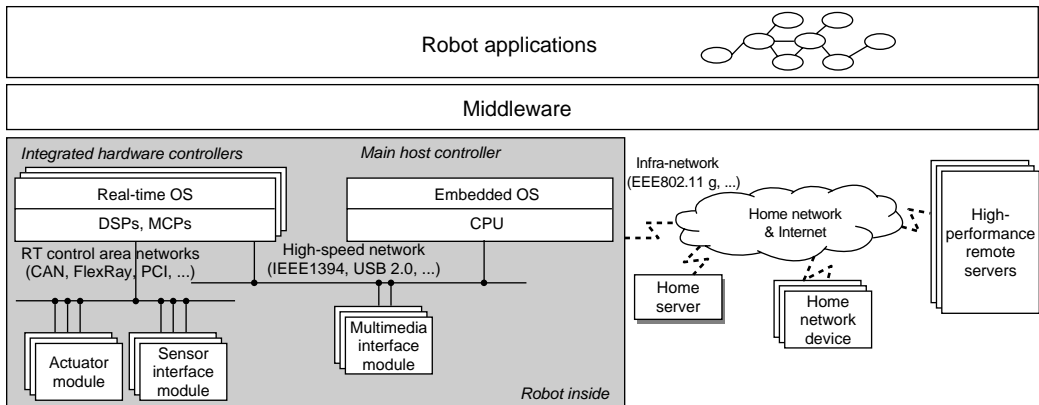


FIGURE 18.1 Reference structure of hardware and software of a networked service robot.

18.3.1 General Robot Hardware Platform

Figure 18.1 depicts the internal structure of hardware and software of the networked service robot, mainly concentrating on the hardware aspect. Two of the most essential properties of the networked service robot are (1) that it should be able to utilize a high-performance remote server provided by a service provider and (2) that it should be able to interface with various smart home appliances and sensor networks that are connected to a larger home network. Clearly, this type of robot is inherently a part of an overall distributed system including remote servers and various home network appliances.

Apparently from Figure 18.1, the networked service robot itself is a self-contained distributed system as well; it is composed of a number of embedded processors, hardware devices, and communication buses. More specifically, as shown in Figure 18.1, main hardware components in the robot are main host controller (MHC) and one or more integrated hardware controllers (IHC). An IHC provides access to sensors and actuators for other components such as other IHCs and an MHC. The MHC acts as an interface to the robot from the outside world; it provides a graphical user interface for interactions with the robot users and it routes messages from an inner component to the remote server or the home network appliances and vice versa.

For communication among the IHCs and MHC, a high-bandwidth medium such as Giga-bit Ethernet, USB 2.0, or IEEE1394 is used. It allows a huge amount of data streams such as MPEG4 video frames to be exchanged inside the robot. Also, a controller area network such as CAN or FlexRay is used for communication among the IHCs, sensors, and actuators. Note that it is important to provide timing guarantees for this type of communication.

18.3.2 Motivations for RSCA

We believe that the hardware structure shown in Figure 18.1 will be typical of future robot platforms. Simultaneously, it will pose a tremendous amount of software complexity to robot software. To overcome this challenge, robot application developers should maximize interoperability, reusability, and reconfigurability of robot software. Also, platform-, model-, and component-based methods should be incorporated into robot software development. To that end, we have developed the RSCA by adopting a widely accepted middleware technology from the software radio (SDR) domain. It is called software communications architecture (SCA) [7]. We have extended it for use in a networked service robot. Beyond simply creating the RSCA, the desired goal is to create a standard that could serve the robotics community at large.

18.3.3 Overall Structure of RSCA

The RSCA is specified in terms of a set of common interfaces for robot applications. These interfaces are grouped into two classes: (1) standard OE interfaces and (2) standard application component interfaces.

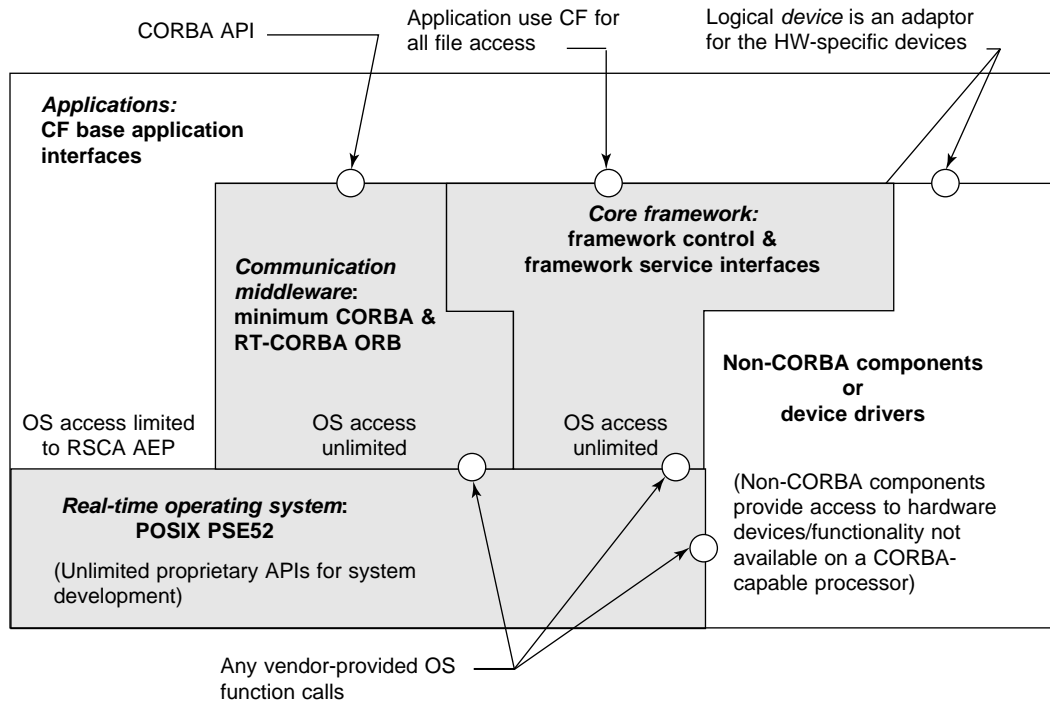


FIGURE 18.2 Overview of the operating environment of RSCA.

The former defines APIs that developers use to dynamically deploy and control applications and to exploit services from underlying platforms. The latter defines interfaces that an application component should implement to exploit the component-based software model supported by the underlying platforms.

As shown in Figure 18.2, the RSCA's OE consists of a real-time operating system (RTOS), a communication middleware, and a deployment middleware called core framework (CF). Since the RSCA exploits COTS software for RTOS and communication middleware layers, most of the RSCA specification is devoted to the CF. More specifically, it defines the RTOS to be compliant to the PSE52 class of the IEEE POSIX.13 Real-Time Controller System profile [6], and the communication middleware to be compliant to minimum CORBA [13] and RT-CORBA v.2.0 [9,14]. The CF is defined in terms of a set of standard interfaces called CF interfaces, and a set of XML descriptors called domain profiles, as will be explained subsequently in Section 18.3.4.

The RTOS provides a basic abstraction layer that makes robot applications both portable and reusable on diverse hardware platforms. Specifically, a POSIX compliant RTOS in the RSCA defines standard interfaces for multitasking, file system, clock, timer, scheduling, task synchronization, message passing, and I/O.

The communication middleware is an essential layer that makes it possible to construct distributed and component-based software. Specifically, the RT-CORBA compliant middleware provides (1) a standard way of message communication, (2) a standard way of using various services, and (3) real-time capabilities. First, the (minimum) CORBA ORB in the RSCA provides a standard way of message communication between components in a manner that is transparent to heterogeneities existing in hardware, operating systems, network media, communication protocols, and programming languages. Second, the RSCA communication middleware provides a standard way of using various services. Among others, naming, logging, and event services are key services that the RSCA specifies as mandatory services. Finally, the RT-CORBA in the RSCA provides real-time capabilities including static and dynamic priority scheduling disciplines and prioritized communications in addition to the features provided by CORBA.

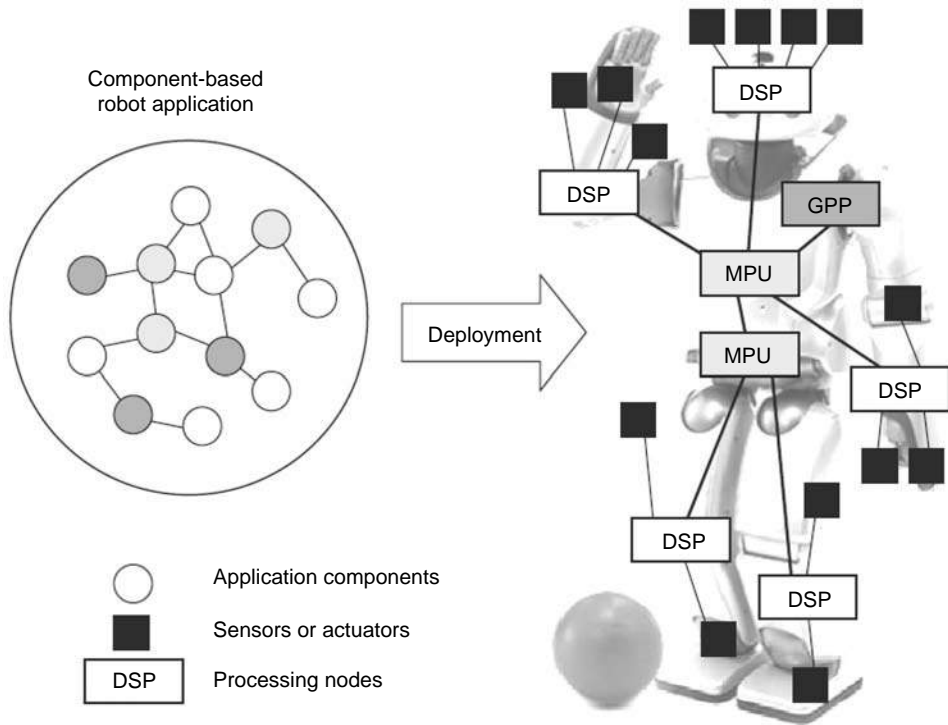


FIGURE 18.3 Deployment of component-based robot applications.

The deployment middleware layer provides a dynamic deployment mechanism by which robot applications can be loaded, reconfigured, and run. A robot application consists of application components that are connected to and cooperate with each other as illustrated in Figure 18.3. Consequently, the deployment entails a series of tasks that include determining a particular processing node to load each component, connecting the loaded components, enabling them to communicate with each other, and starting or stopping the whole robot software. Subsequently in the next subsection, we present the structure of the RSCA's deployment middleware in detail.

18.3.4 RSCA Core Framework

Before getting into the details of the RSCA CF, we begin with a brief explanation about structural elements that the RSCA CF uses to model a robot system and the relationship between these elements. In the RSCA, a robot system is modeled as a domain that distinguishes each robot system uniquely. In a domain, there exist multiple processing nodes and multiple applications. The nodes and applications respectively serve as units of hardware and software reconfigurability. Hardware reconfigurability is achieved by attaching or detaching a node to or from the domain. A node may have multiple logical devices, which act as device drivers for real hardware devices such as field programmable gate arrays (FPGAs), DSPs, general-purpose processors, or other proprietary devices. In contrast, software reconfigurability is achieved by creating an instance of an application in a domain or removing the instance from the domain. An application consists of components, each of which is called a resource. As depicted in Figure 18.4, a resource in turn exposes ports that are used for communication to or from other resources [8]. For communication between two components, a port of one component should be connected to a port of the other where the former port is called a *uses* port and the latter port is called a *provides* port. For ease of communication between the components and the logical devices, the logical devices are modeled as a specialized form of a resource.

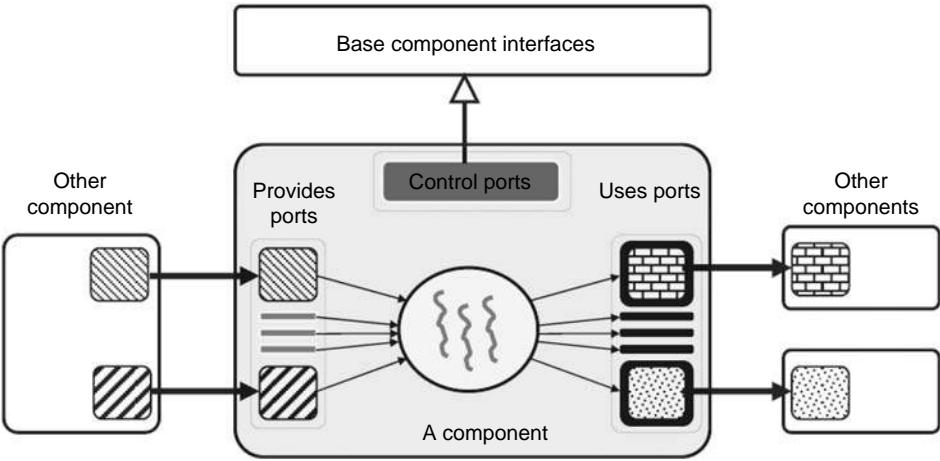


FIGURE 18.4 RSCA component model.

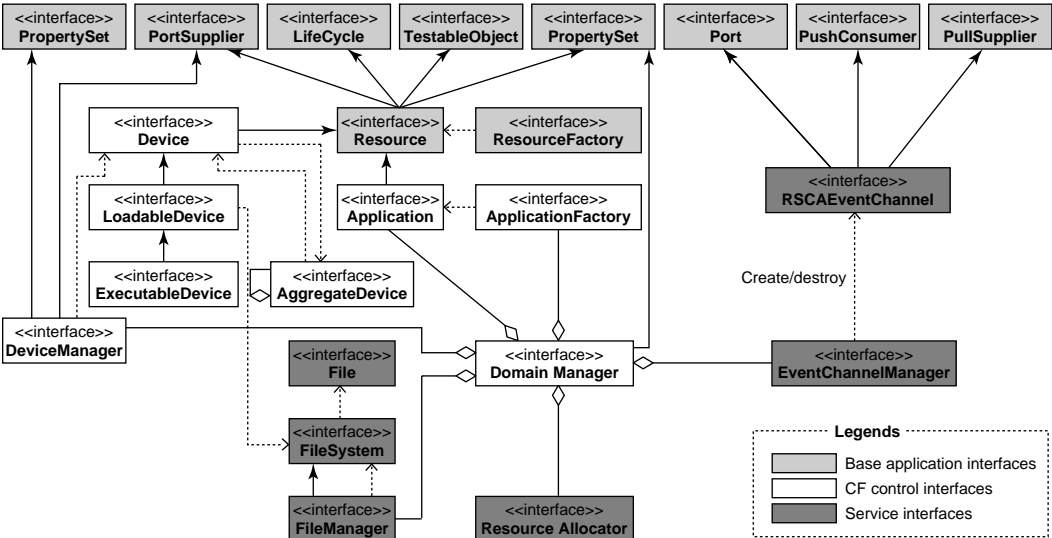


FIGURE 18.5 Relationships among CF interfaces.

Configurations of each of the nodes and applications are described in a set of XML files called domain profiles. We now explain the CF interfaces and the domain profiles in detail.

18.3.4.1 Core Framework Interfaces

As shown in Figure 18.5, CF interfaces consist of three groups of APIs: base application interfaces, CF control interfaces, and service interfaces. Each of the interface group is defined as application components, domain management, and services, respectively. The deployment middleware is therefore the implementation of the domain management and service part of the RSCA CF interfaces.

Specifically (1) base application interfaces are interfaces that the deployment middleware uses to control each of the components comprising an application. Thus, every application component should implement these interfaces as depicted in Figure 18.5. These interfaces include the functionalities of starting and stopping the resource, configuring the resource, and connecting a port of the resource to a port of another

resource. (2) The CF control interfaces are interfaces provided to control the robot system. Controlling the robot system includes activities such as installing/uninstalling a robot application, starting/stopping it, registering/deregistering a logical device, tearing-up/tearing-down a node, etc. (3) Service interfaces are common interfaces that are used by both deployment middleware and applications. Currently, three services are provided: distributed file system, distributed event, and QoS.

Among these interfaces, *ResourceAllocator* and *EventChannelManager* are interfaces defined for QoS and distributed event services, respectively. The *ResourceAllocator* together with domain profiles allow applications to achieve desired QoS guarantees by simply specifying their requirements in the domain profiles [10]. To guarantee the desired QoS described in the domain profiles, *ResourceAllocator* allocates a certain amount of resources based on current resource availability and these allocated resources are guaranteed to be enforced throughout the lifetime of an application relying on the COTS layer of the OE. In contrast, the *EventChannelManager* service together with domain profiles allow applications to make connections via CORBA event channels by simply specifying their connections in domain profiles. The *RSCAEventChannel* interface represents an event channel created by the *EventChannelManager*.

18.3.4.2 Domain Profiles

Domain profiles are a set of XML descriptors describing configurations and properties of hardware and software in a domain. They consist of seven types of XML descriptors as shown in Figure 18.6. (1) The *device configuration descriptor* (DCD) describes hardware configurations. (2) The *software assembly descriptor* (SAD) describes software configurations and the connections among components. (3) These descriptors consist of one or more *software package descriptors* (SPDs), each of which describes a software component (*Resource*) or a hardware device (*Device*). (4) The *Properties Descriptor File* (PRF) describes optional reconfigurable properties, initial values, and executable parameters that are referenced by other domain profiles. (5) The *DomainManager configuration descriptor* (DMD) describes the *DomainManager* component and services used. (6) The *software component descriptor* (SCD) describes interfaces that a component provides or uses. Finally, (7) the *device package descriptor* (DPD) describes the hardware device and identifies the class of the device.

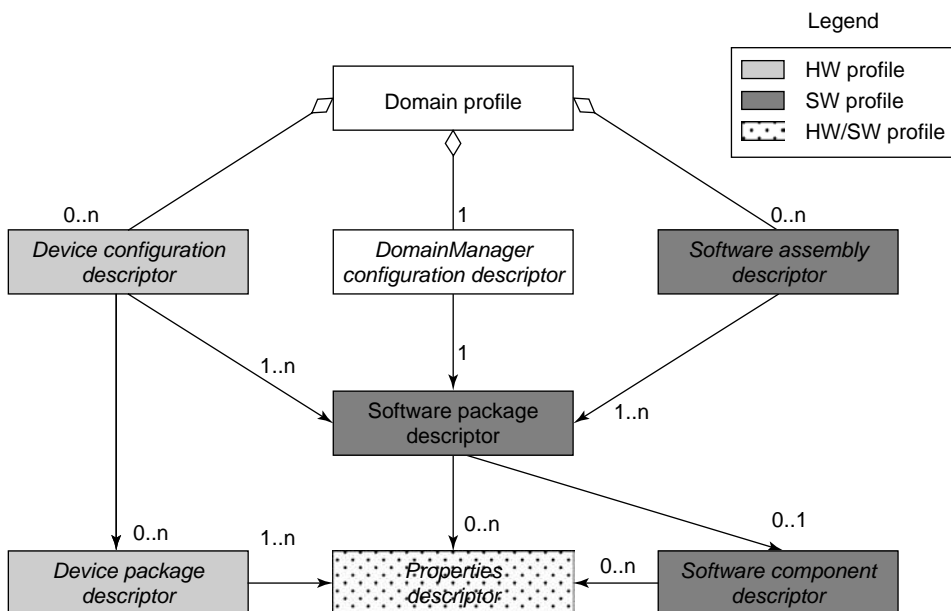


FIGURE 18.6 Relationships among domain profiles.

18.3.5 Assessment of RSCA

We now show how the RSCA meets the requirements identified in Section 18.2. First, the RSCA provides device abstraction via the *Device* interface. The *Device* interface provides interfaces to allocate and deallocate a certain amount of resources such as memory, CPU, and network bandwidth. The *Device* interface also supports synchronization of accesses to a resource by providing resource usage and management status. An application developer should, of course, specify how resources are allocated and synchronized based on efficiency of the resource usage. In contrast, the *Resource* interface provides interfaces for each component. The *Device* interface is also a specialized interface of the *Resource* interface, which enables logical devices to be transparently abstracted into components. The *Resource* interface provides interfaces for connecting components and managing component life cycle by inheriting the *PortSupplier* and *LifeCycle* interfaces, respectively. Along with these, XML descriptors that describe RSCA components and their assembly make possible the dynamic plug and play of RSCA components.

Second, the RSCA fully supports the dynamic reconfigurability of the robot system, an application, and a component. For the system-level reconfiguration, the RSCA provides a way to dynamically change the installed and running applications. For the application-level reconfiguration, the RSCA provides a way to describe an application in various possible configurations (structures and parameters), each for different application requirements and constraints. The RSCA should choose the most appropriate assembly among all possible assemblies according to the current resource availability at the deployment time. The *ApplicationFactory* is the component in the RSCA CF responsible for searching for the most suitable processing units (devices) that can satisfy the system dependency and resource requirements of a given component. However, for the time being, the current RSCA does not support runtime reconfiguration such as replicating components and replacing faulty components. Finally, for reconfiguration at the individual component level, the RSCA provides a way to specify and dynamically configure reconfigurable parameters of components. Among the RSCA CF interfaces, *PropertySet* provides interfaces that allow applications to query and configure the reconfigurable parameters at runtime.

Third, the RSCA supports resource frugality by employing a compact and efficient communication middleware and deployment middleware. The communication middleware of the RSCA is minimum CORBA that supports the pure static interface invocation of objects. It prohibits objects with unknown interfaces from communicating with each other. Also, the CF of the RSCA (the deployment middleware) does not support any configurable built-in library for component life-cycle management. Such a library is usually called a container. A container is supported by most traditional deployment middleware such as CORBA component model (CCM) [12], Enterprise JavaBeans [15], and Distributed Component Object Model (DCOM) [11]. The adoption of a container into these middleware products has been very successful in the information infrastructure domain. However, such a full-featured configurable built-in library unavoidably causes a large execution overhead and thus is not appropriate for networked service robots. Instead, the RSCA CF allows developers to manage component life cycle through the *ResourceFactory* interface, which is implemented by developers.

Finally, the RSCA supports application-level QoS guarantees. The RTOS and the communication middleware support real-time guarantees for individual components. Application developers can achieve their desired QoS by simply specifying their requirements in domain profiles. To guarantee the QoS described in domain profiles, the *ResourceAllocator* provides mechanisms for resource management, admission control, resource allocation, and resource enforcement.

18.4 Future Challenges of Robot Middleware

To develop commercially viable middleware for real-time and embedded systems, researchers must address technical challenges that arise owing to future trends in embedded systems design. Particularly, two trends are important for the discussion of future real-time and embedded middleware. First, an embedded system will become a heterogeneous distributed system as it requires extremely high performance in

both computing and communication. For instance, even a small SDR handset consists of a number of heterogeneous processors connected with diverse communication buses. Moreover, the advent of the MPSoC (multiprocessor system on a chip) technology makes chip supercomputers available for embedded systems. Secondly, applications that used to run only on high-performance mainframe computers will be deployed on embedded systems for enhanced user interfaces. However, such performance hogging applications should be rewritten to adapt to the unique OE of embedded systems.

Under these circumstances, middleware should play a crucial role for future real-time and embedded systems as it has done in other domains. In this section, we show three of the future technical challenges of real-time and embedded middleware, they are (1) performance optimization via new metrics, (2) streaming support for high-volume data, and (3) provision of domain-specific abstractions.

18.4.1 Performance Optimization via New Metrics

With the proliferation of inexpensive high-performance embedded microprocessors, future real-time and embedded systems are expected to be supercomputers that provide massive computational performance. They will make embedded devices much easier to use through human-centric interfaces that integrate real-time vision processing, speech recognition, and multimedia transmission. Clearly, middleware will be an enabling technology for realizing these functionalities [1].

However, to achieve embedded supercomputing, we need to change the way a system is designed and optimized and reflect this change into the future middleware. For instance, let us consider the case of the Internet-implemented robot intelligence. Historically, the most critical hurdle that robot industry has been facing in commercializing robots is the unacceptable cost of a robot compared to the level of intelligence that it can deliver. The Internet-implemented robot intelligence is a result of cost-conscious system redesign and optimization since it amortizes the cost of high-performance computing over a number of client robots. This leads to affordable intelligence to each robot. As we have seen earlier, the component-based middleware is a key to such an innovation.

Embedded supercomputers are mostly mobile systems whose massive computational performance is derived from the power in a battery. Thus, reduction in the power consumption of embedded systems is another critical design aspect. Consequently, embedded system developers should judge their systems via a new metric considering performance per unit power consumption and eventually via their performance–power–cost product [1]. Research into real-time and embedded middleware should deliver software architectures and mechanisms that considers this new metric.

18.4.2 Streaming Support for High-Volume Data

A majority of important applications running on real-time and embedded systems involve streaming of extremely large volume of data. For instance, a networked service robot hosts applications for real-time vision, speech recognition, and even multimedia presentation. However, current middleware products for real-time and embedded systems only partially address problems related to the data streaming. They provide neither a streaming data model as a first-class entity nor a data synchronization model for stream dataflows. Clearly, the future real-time and embedded middleware should support these mechanisms.

18.4.3 Domain-Specific Abstractions

A straightforward adoption of existing middleware products to real-time and embedded systems leads to a failure for the lack of domain-specific abstractions. Many robot developers find it very hard to use them in developing robots since their programming abstractions are different from those of existing robot software frameworks that have been used in robot industry for decades. Robot software frameworks specialize in modeling sensor and actuator devices and executable devices such as DSPs and MPUs. They also provide abstraction models that capture robot behavior architectures. To expedite the successful adoption of robot middleware, it is thus desirable to integrate existing robot software frameworks into robot middleware.

With this, robot developers can not only design robot applications using conventional robot software frameworks but also enjoy component-based software deployment at runtime.

18.5 Conclusions

While middleware technology has been successfully utilized in the enterprise computing domain, its adoption into commercial real-time and embedded systems is slow due to their extra requirements related to resource limitation, reliability, and cost. In this chapter, we presented a reference middleware architecture that specifically addresses the requirements of real-time and embedded systems. In doing so, we took robotics as a domain of discussion since a networked service robot, a representative real-time and embedded system, draws a great deal of interest. The middleware is called the RSCA.

The RSCA provides a standard OE for robot applications along with a framework that expedites the development of such applications. The OE is composed of an RTOS, a communication middleware, and a deployment middleware, which collectively forms a hierarchical structure that meets the four requirements we have identified for the networked service robot. The RSCA meets these requirements by providing the following capabilities. First, the RSCA provides a device abstraction model as well as a component model. This enables the plug and play of software modules allowing robot applications to be developed separately from the robot hardware in a device-agnostic manner. Second, the RSCA supports the dynamic reconfigurability of a robot system, an application, and a component. This enables robots to be adapted to various needs of users and varying operating conditions. Third, the RSCA supports resource frugality by employing the pure static interface invocation of objects dismissing a full-featured configurable built-in library for the component life-cycle management. This allows the RSCA to be used for robots even under stringent resource constraints. Finally, the RSCA supports real-time and QoS capabilities. As such, the RSCA solves many of the important problems arising when creating an application performing complex tasks for networked service robots.

The RSCA only partially addresses the technical challenges of real-time and embedded middleware since new important requirements arise with new trends in future real-time and embedded computing. These challenges have something to do with new performance optimization metrics, effective data streaming support, and domain-specific abstractions. Middleware research should deliver software architectures and mechanisms to cope with them.

Acknowledgment

This work was supported in part by Korea Ministry of Information and Communication (MIC) and Institute of Information Technology Assessment (IITA) through IT Leading R&D Support Project.

References

1. T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabati, and W. Wolf, Mobile supercomputers, *Communications of the ACM*, vol. 37, no. 5, pp. 81–83, May 2004.
2. AUTOSAR, AUTomotive Open System ARchitecture, <http://www.autosar.org>.
3. W. Emmerich, Software engineering and middleware: a roadmap, in *The Future of Software Engineering*, A. Finkelstein ed., pp. 76–90, ACM Press, New York, 2000.
4. Y. Fujita, Research and development of personal robot in NEC, NEC response to the Robotics Request for Information, <http://www.omg.org/cgi-bin/doc?robotics/06-02-02>.
5. S. Hong, J. Lee, H. Eom, and G. Jeon, The robot software communications architecture (RSCA): embedded middleware for networked service robots, in *Proceedings of the IEEE International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 2006.

6. Institute for Electrical and Electronic Engineers (IEEE), Information technology—standardized application environment profile—POSIX realtime application support (AEP), IEEE Std 1003.13, February 2000.
7. Joint Tactical Radio Systems, Software Communications Architecture Specification V.3.0, August 2004.
8. S. Kim, J. Masse, and S. Hong, Dynamic deployment of software defined radio components for mobile wireless internet applications, in *Proceedings of International Human.Society@Internet Conference (HSI)*, June 2003.
9. F. Kuhns, D. C. Schmidt, and D. L. Levine, The performance of a real-time I/O subsystem for QoS-enabled ORB middleware, in *Proceedings of the International Symposium on Distributed Objects and Applications*, pp. 120–129, September 1999.
10. J. Lee, S. Kim, J. Park, and S. Hong, Q-SCA: incorporating QoS support into software communications architecture for SDR waveform processing, *Journal of Real-Time Systems*, vol. 34, no. 1, pp. 19–35, September 2006.
11. Microsoft Corporation, The distributed component object model (DCOM), <http://www.microsoft.com/com/tech/DCOM.asp>.
12. Object Management Group, CORBA component model version 3.0, June 2002.
13. Object Management Group, The common object request broker architecture: core specification revision 3.0, December 2002.
14. Object Management Group, Real-time CORBA specification revision 1.1, August 2002.
15. Sun Microsystems Inc., Enterprise JavaBeans technology, <http://java.sun.com/products/ejb/>.

IV

Real-Time Communications/ Sensor Networks

19

Online QoS Adaptation with the Flexible Time-Triggered (FTT) Communication Paradigm

Luis Almeida

Universidade de Aveiro

Paulo Pedreiras

Universidade de Aveiro

Joaquim Ferreira

Instituto Politécnico de Castelo Branco

Mário Calha

Universidade de Lisboa

José Alberto Fonseca

Universidade de Aveiro

Ricardo Marau

Universidade de Aveiro

Valter Silva

Universidade de Aveiro

Ernesto Martins

Universidade de Aveiro

19.1	Introduction	19-1
19.2	Toward Operational Flexibility	19-2
	Rationale for Improving Operational Flexibility in Distributed Embedded Systems • Where We Stand with Existing Communication Protocols • Requirements for Operational Flexibility	
19.3	The Flexible Time-Triggered Paradigm	19-5
	System Architecture • The Dual-Phase Elementary Cycle • The System Requirements Database • Main Temporal Parameters within the EC	
19.4	The Synchronous Messaging System	19-8
	Synchronous Scheduling Model • Schedulability Analysis of Synchronous Traffic • Adding Dynamic QoS Management • QoS Management Example Based on the Elastic Task Model	
19.5	The Asynchronous Messaging System	19-14
	Asynchronous Traffic Model • Schedulability of Asynchronous Traffic	
19.6	Case Study: A Mobile Robot Control System	19-17
	Establishing the Communication Requirements • Using Online Adaptation • Experimental Results	
19.7	Conclusions	19-19

19.1 Introduction

Owing to continued developments in the integration of processing and communications technology along the last decades, distributed architectures have progressively become pervasive in many real-time application domains, ranging from avionics to automotive systems, industrial machinery, feedback control, robotics, computer vision, and multimedia [16]. These architectures present several advantages: they favor

dependability through easy replication of nodes and definition of error-containment regions; composability since the system can be built by integrating nodes that constitute independent subsystems; scalability through easy addition of new nodes to increase the system capacity; and maintainability owing to the modularity of the architecture and easy node replacement [10].

In these systems, there has also been a trend toward higher flexibility to support dynamic configuration changes such as those arising from evolving requirements, environmental changes, and online quality of service (QoS) management [10,13,26,39,43]. These features are useful to increase the efficiency in the utilization of system resources [13] and this is relevant given the close relationship between resource utilization and delivered QoS in a general sense. In several applications, assigning higher CPU and network bandwidth to tasks and messages, respectively, increases the QoS delivered to the application. This is true, for example, in control applications [12], at least within certain ranges [28], and in multimedia applications [22]. Therefore, managing the resources assigned to tasks and messages, e.g., by controlling tasks execution or messages transmission times and rates, allows a dynamic control of the delivered QoS. Efficiency gains can be achieved in two situations: either by maximizing the utilization of system resources to achieve a best possible QoS for different load scenarios or by adjusting the resource utilization according to the application instantaneous QoS requirements, i.e., using only the resources required at each instant. Both situations referred above require an adequate support from the computational and communications infrastructure so that relevant parameters of tasks and messages can be dynamically adjusted [10]. However, performing this adjustment promptly in a distributed system is a challenging task because of the network-induced delays, the need to achieve a consensus among the nodes involved in the adjustment, and the need to enforce the adjustment synchronously in all nodes.

Therefore, the authors proposed previously an architecture in which a global traffic scheduler and system synchronizer are placed in a central locus that also includes a database with all communication requirements and other system characteristics [2,6,34]. The fact that all communication and synchronization related data are centralized in one node allows its prompt and efficient management. This node is called master and the scheduling of the system activities (tasks and messages) is periodically broadcast to the other nodes in a master/slave fashion using specific control messages. This became known as the flexible time-triggered (FTT) paradigm since it still follows the time-triggered model [21], synchronized by the master, but allows complex online system updates with low latency, which is well suited to support dynamic QoS management with arbitrary policies and with timeliness guarantees [2,34]. To the best of our knowledge, the FTT paradigm has been the first attempt to introduce a high level of operational flexibility in the time-triggered communication model to support emerging applications that adapt to current environment operating conditions or that adjust the QoS provided at each instant to maximize both the resource efficiency and an aggregate QoS metric. This paradigm has been applied to several networking technologies, leading to the protocols FTT-CAN [6], FTT-Ethernet [35], and more recently FTT-SE [27], which are based on CAN, Ethernet, and microsegmented switched Ethernet, respectively. Practical applications of these protocols are reported in the literature concerning the control of mobile autonomous robots with FTT-CAN [42] and video surveillance with FTT-Ethernet [36].

In this chapter, we present the FTT paradigm in Section 19.3, after discussing the motivation for its development in Section 19.2. Then, in Sections 19.4 and 19.5 we analyze the two communication subsystems that are part of the FTT model, namely the synchronous messaging system (SMS) and the asynchronous messaging system (AMS), focusing on the respective real-time analysis, which is the basis for the guarantees of timely behavior. We conclude the chapter with the analysis of a case study related to a mobile robot highlighting the benefits of using such synchronous and flexible approach.

19.2 Toward Operational Flexibility

In general, the term flexibility means the ability to change form. Since the term can apply to many different areas, we need to define exactly to which form we are referring. For example, concerning embedded communication systems, we can refer to aspects such as physical media, topology, bit encoding, live

insertion support, mode changes support and rapid download of new application software, and dynamic communication requirements support. The more options the system supports the higher is its degree of flexibility. Among these aspects, flexibility with respect to the communication requirements brings up the potential for higher efficiency in the use of system resources, which is, today, a particularly important issue given the growing pressure to reduce costs particularly in some sectors such as the automotive domain, or even the consumer electronics domain [10]. It also brings the ability to adapt to different operational environment conditions, adjusting the resources used to the QoS that can be provided at each instant [15]. We refer to this flexibility with respect to the communication requirements as operational flexibility, meaning that the system supports the online addition, removal, and modification of message streams and, in a broader scope, of tasks, too, i.e., it supports system online reconfiguration.

19.2.1 Rationale for Improving Operational Flexibility in Distributed Embedded Systems

As referred above, the main motivation for improving the operational flexibility of distributed embedded systems (DESSs) is to support their use in dynamic operational scenarios:

- Systems with variable number of users, either humans or not (e.g., traffic control and radar) so that more users can be served, even if with lower QoS [23,29].
- Systems that operate in changing physical environments (e.g., robots and cars) so that the sampling/execution/transmission rates and control parameters are adapted to minimize resource utilization while maintaining an adequate level of QoS [19,38].
- Systems that can self-reconfigure dynamically to cope with hazardous events or evolving functionality or requirements to improve availability or dependability in general (e.g., military systems and telecommunication systems) [18,30,39,40].

Similar motivation is pointed out by Prasad et al. [37] and Lu et al. [26], stating that using static schedules in a system, thus without operational flexibility, leads to inefficient resource usage and to nongraceful degradation and that it is more efficient to leave some operational decisions in terms of resource usage to runtime. Thus, operational flexibility is also the key to support the so-called adaptive embedded systems as defined in the ARTIST Roadmap [10], which will be the basis for forthcoming efficient designs from consumer electronics to automotive systems, telecommunications, home automation, industrial equipment, etc.

From this reasoning, it becomes clear the importance of efficient resource management, naturally extending to the network, a fundamental resource in a DES. It is thus our purpose to provide the operational flexibility, at the network level, required to support efficient network bandwidth management while, at the same time, providing timeliness guarantees and safe behavior.

19.2.2 Where We Stand with Existing Communication Protocols

Supporting a high level of operational flexibility, just by itself, is not a challenge. In fact, using any common network protocol that does not constrain the load generated by each node, e.g., Ethernet or CAN, would allow changing the network operational parameters online. However, if unconstrained changes occur at runtime it is not possible to provide any guarantees with respect to timeliness and safety.

For this reason, safety-critical systems typically use static TDMA-based protocols (e.g., TTP/C, FlexRay, and SAFEbus), which maximize the use of *a priori* knowledge projected into the future such as the activation/transmission instants. With this *modus operandi*, the traffic timeliness can be assessed during the design phase and errors, revealed as omissions, can be promptly detected at the nodes interfaces. These protocols have, naturally, low operational flexibility but this can be improved, for example, as in TTP/C, by allowing shifting between a set of predefined modes online. However, the number of modes is normally small due to implementation constraints, e.g., memory, not being sufficient to provide efficient QoS adaptation. In fact, the number of modes is typically much lower than the total number of possible state combinations among network streams, or subsystems [40].

Other protocols, normally targeted to interconnect high-bandwidth computer peripherals, such as USB2.0 and Firewire, support online bandwidth negotiation and reservation with isochronous channels. However, once channels are assigned they are not further adapted unless they are destroyed and created again, which takes further time. This is still not adequate to efficient dynamic QoS management in which, for example, a new stream that would cause an overload can still be accepted by removing bandwidth assigned to other streams whose subsystems can tolerate a lower QoS. This kind of support is generally not available in current protocols and it is a challenging task to provide the required level of operational flexibility with timeliness and safety as necessary for many DESs.

19.2.3 Requirements for Operational Flexibility

Achieving high network operational flexibility with timeliness and safety requires:

1. Bounded communication delays.
2. Online changes to the communication requirements with low and bounded latency, which requires dynamic traffic scheduling.
3. Online admission control (based on appropriate schedulability analysis) complemented with bandwidth management with low and bounded latency.
4. Sufficient resources for a predefined safe operating mode.
5. Change attributes that define, which online changes are permitted for each stream.

The first requirement calls for an appropriate network access protocol that is deterministic and analyzable. The second and third requirements correspond to a dynamic planning-based traffic scheduling paradigm. The fourth and fifth requirements are ways of constraining flexibility to a set of permitted changes that always result in safe operational scenarios. All the latter four requirements call for prompt access to global information concerning the managed message stream, e.g., transmission time, nominal, minimum and maximum rates, deadline, and priority. This constraint leads to a centralized architecture in which all the relevant information can be accessed locally, because using a distributed architecture the management actions would require a consensus protocol, causing a substantial communication overhead with the consequent delays.

Therefore, the system architecture should include a communication requirements database (CRDB) residing in a specific node where the traffic scheduler and bandwidth manager, including admission controller, also execute. This, naturally, points to a master–slave architecture in which the CRDB, traffic scheduler, and bandwidth manager reside in the master and control the pace of communications in the system by sending appropriate triggers regularly (Figure 19.1). This is a time-triggered model in which the notion of time is projected from the master onto the rest of the system, supporting global offsets among message streams as well as prompt detection of omissions (absence of reaction to the master triggers) and easy enforcement of fail-silence, for example, using bus-guardians (transmission instants, or transmission windows, are well known). Moreover, the properties of the underlying network protocol can be overridden by the centralized traffic scheduler, further increasing the flexibility of the scheduling and making it less dependent on the specific network technology.

One alternative approach would be dropping the traffic scheduler and let the nodes decide when to transmit, in an event-triggered fashion. The QoS management would still be carried out in a single node, holding the CRDB, and any operational change in the message set would be preceded by a consensus phase involving all nodes. With respect to the previous time-triggered architecture, this one has no support for global offsets but there are no extra triggers consuming bandwidth, either. However, there is a strong dependency on the native real-time capabilities of the underlay specific network technology, which cannot be easily overridden in this case.

Finally, dependable operation in the two cases requires replication of both network and master, or node holding the CRDB, as well as the enforcement of a fail-silent behavior in the nodes or, alternatively, the deployment of adequate mechanisms to tolerate timing failures. This topic is, however, outside the scope of this chapter. The reader is referred to Ref. 17 for an overview of the fault-tolerance mechanisms developed for a master–slave architecture of this kind, which is described in the following sections.

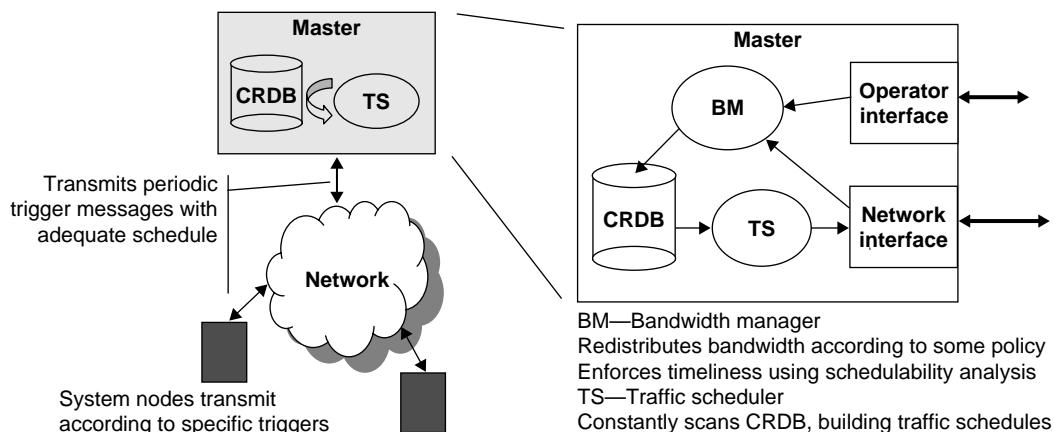


FIGURE 19.1 Master-slave paradigm for high operational flexibility with timeliness guarantees.

19.3 The Flexible Time-Triggered Paradigm

The FTT communication paradigm emerged from the architectural requirements referred above to provide a high level of operational flexibility under timeliness and safety guarantees, together with the option for a time-triggered model. When deploying the FTT paradigm, a network technology must be chosen (e.g., CAN and Ethernet), leading to a specific protocol, (e.g., FTT-CAN, FTT-Ethernet, and FTT-SE), respectively. These protocols, despite a few differences with respect to specific technology-related mechanisms, bear several important properties that are technology independent, emerging from the FTT paradigm. This section will mostly address the common FTT features, pointing to the specificities of each protocol when relevant. The reader is referred to the literature on the specific protocols for more detailed information.

19.3.1 System Architecture

The FTT system architecture follows closely the one presented in Figure 19.1, but with the CRDB replaced by an SRDB (system requirements database) because it contains more system information than just communication requirements (see further on). It is an asymmetric architecture of the master-slave type, with both communication requirements and message scheduling localized in one single node, the master. This allows updating both communication requirements and message scheduling on-the-fly and facilitates implementing online admission control to support dynamic changes in the communication requirements with guaranteed timeliness and low latency, as required for dynamic QoS management.

The FTT protocols also support a combination of time and event-triggered traffic called synchronous and asynchronous, respectively, with the first type being scheduled by the master and the latter triggered autonomously by the nodes, but still confined to specific windows defined by the master, so both traffic classes are temporally isolated. The way the asynchronous traffic is supported depends on the specific network protocol used, but the temporal isolation property is enforced in any implementation. Two subsystems provide application services for each type of communication, namely the SMS and the AMS. The SMS offers services based on the producer-consumer model [20] while the AMS offers send and receive basic services, only. These subsystems are further analyzed later on.

19.3.2 The Dual-Phase Elementary Cycle

A key concept in the FTT protocols is the elementary cycle (EC), which is a fixed duration time slot used to allocate traffic on the bus. This concept is also used in other protocols such as ETHERNET Powerlink and WorldFIP. The bus time is organized as an infinite succession of ECs. Within each EC there are two

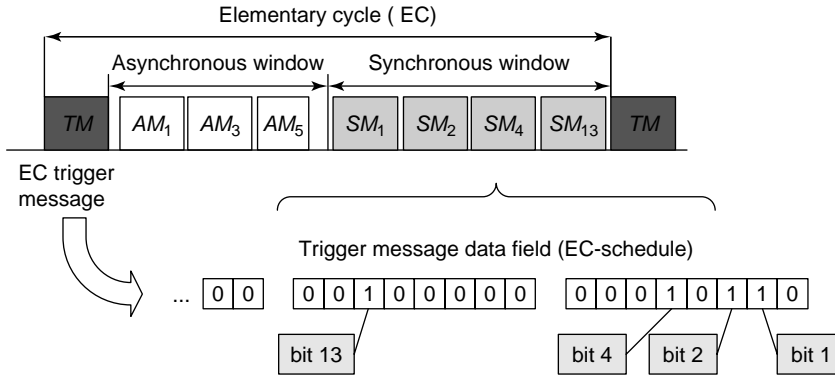


FIGURE 19.2 The elementary cycle and EC-schedule encoding in FTT-CAN.

windows, each one dedicated to a different type of traffic: synchronous and asynchronous. The order of these windows is protocol dependent. For example, in FTT-CAN the asynchronous window precedes the synchronous one (Figure 19.2) while in FTT-Ethernet and FTT-SE it is the opposite, the reason being related with implementation aspects. In any case, the traffic scheduling is carried out considering strict temporal isolation between the two windows meaning that a transmission is only started if it finishes within the respective window.

The master node starts each EC by broadcasting a trigger message (TM). This control message synchronizes the network and conveys in its data field the identification of the synchronous messages that must be transmitted by the slave nodes within the respective EC (the EC-schedule). All the nodes on the network decode the TM and scan a local table to identify whether they are the producers of any of the scheduled messages. If so, they transmit those messages in the synchronous window of that EC. In FTT-CAN, the collisions within the window are sorted out by the native arbitration mechanism and in FTT-SE there are no collisions since messages are queued in the output switch ports. In FTT-Ethernet, the master indicates, together with the EC-schedule, when each message must be transmitted within the window to avoid the nondeterministic collision resolution mechanism of shared Ethernet. The traffic scheduler never schedules more messages than those that fit in the respective synchronous window, thus enforcing strict temporal isolation between consecutive ECs.

This transmission control scheme with the master sending only one control message per cycle is called master/multislave and saves overhead with respect to common master-slave protocols since the same master message triggers several slave messages and the turnaround time of all slaves is overlapped.

The asynchronous traffic in FTT-CAN is efficiently handled using the native arbitration of CAN. The protocol is designed in a way that the sequence of asynchronous windows behaves like a CAN bus operating at a lower bandwidth. All nodes can issue transmit requests freely within the asynchronous windows. At the end of this window nonserved requests are withdrawn and queued, being resubmitted in the following asynchronous windows. FTT-Ethernet uses a polling mechanism that, despite less efficient, ensures collision-free access to the medium and thus is temporally deterministic. The aperiodic traffic can still be real-time or non-real-time, with the former having higher priority over the latter. In the FTT-CAN protocol, this property is enforced by allocating appropriate CAN IDs. In FTT-Ethernet, the master uses specific polling periods according to the latency requirements of each real-time asynchronous message.

19.3.3 The System Requirements Database

The SRDB contains three components: synchronous requirements, asynchronous requirements, and system configuration and status. The synchronous requirements component is formed by the synchronous requirements table (SRT), which includes the description of the current synchronous message streams.

$$\text{SRT} \equiv \{SM_i(C_i, Ph_i, P_i, D_i, Pr_i, Xf_i), i = 1, \dots, N_S\} \quad (19.1)$$

For each message stream, C is the respective maximum transmission time (including all overheads), Ph the relative phasing (i.e., the initial offset), P the period, D the deadline, and Pr a fixed priority defined by the application. Ph , P , and D are expressed as integer multiples of the EC duration. N_S is the number of synchronous messages. Xf is a pointer to a custom structure that can be defined to support specific parameters of a given QoS management policy, e.g., admissible period values, elastic coefficient, value to the application (see Ref. 34 for a more complete description of using this structure).

The asynchronous requirements component is formed by the reunion of two tables, the asynchronous requirements table (ART) and the non-real-time requirements table (NRT). The ART contains the description of time-constrained event-triggered message streams, e.g., alarms or urgent reconfiguration requests.

$$\text{ART} \equiv \{AM_i(C_i, \text{mit}_i, D_i, Pr_i), i = 1, \dots, N_A\} \quad (19.2)$$

This table is similar to the SRT except for the use of the mit_i , minimum interarrival time, instead of the period, and the absence of an initial phase parameter, since there is no phase control between different asynchronous messages.

The NRT contains the information required to guarantee that non-real-time message transmissions fit within the asynchronous window, as required to enforce temporal isolation (Equation 19.3). The master only needs to keep a track of the length of the longest non-real-time message that is transmitted by each node.

$$\text{NRT} \equiv \{NM_i(\text{SID}_i, \text{MAX}_C, Pr_i), i = 1, \dots, N_N\} \quad (19.3)$$

SID is the node identifier; MAX_C the transmission time of the longest non-real-time message transmitted by that node, including all overheads; and Pr the node non-real-time priority, used to allow an asymmetrical distribution of the bus bandwidth among nodes. N_N is the number of stations producing non-real-time messages. The last component of the SRDB is the system configuration and status record (SCSR), which contains all system configuration data plus current traffic figures. This information is made available at the application layer so that it can be used either for profiling purposes or at runtime to make the system adaptive, raise alarms, etc. It may occur that some of the SRDB components are not necessary in some simpler systems and in which case they are not implemented.

19.3.4 Main Temporal Parameters within the EC

One fundamental temporal parameter in the FTT protocols is the EC duration, which we consider to be E time units. This parameter establishes the temporal resolution of the system since all periods, deadlines, and relative phases of the synchronous messages are integer multiples of this interval (Equation 19.4). In other words, it sets the time scale of the synchronous traffic scheduler.

$$\forall_{i=1, \dots, N_S} \quad Ph_i = k * E, P_i = l * E, D_i = m * E \quad \text{with } k, l, m \text{ being positive integers} \quad (19.4)$$

Then, within the EC we can identify three intervals, the first of which is the time required to transmit the TM, which we consider constant and equal to LTM time units. It corresponds to an overhead that must be taken into account when scheduling. The following two intervals within the EC are the asynchronous and synchronous windows. Figure 19.3 shows the case of FTT-CAN in which the asynchronous window precedes the synchronous one. The duration of this latter window in the n th EC, $\text{lsw}(n)$, is set according to the synchronous traffic scheduled for it. This value is encoded in the respective TM, together with the EC-schedule, in any FTT protocol. The value of $\text{lsw}(n)$ then determines the duration of the asynchronous window, $\text{law}(n)$, and its start or its end, depending on the specific order of the windows. Basically, $\text{law}(n)$ equals the remaining time between the TM and the synchronous window. The protocol allows establishing a maximum duration for the synchronous windows (LSW) and correspondingly a maximum bandwidth for that type of traffic. Consequently, a minimum bandwidth can be guaranteed for the asynchronous traffic, too.

$$\forall_{n=1, \dots, \infty} \quad 0 \leq \text{lsw}(n) \leq \text{LSW} \text{ and } E - \text{LTM} - \text{LSW} \leq (\text{law}(n) = E - \text{LTM} - \text{lsw}(n)) \quad (19.5)$$

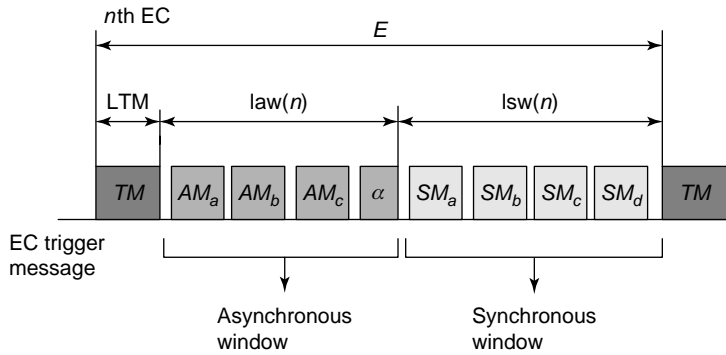


FIGURE 19.3 Main temporal parameters within the EC (FTT-CAN).

Finally, the strict temporal isolation between both types of traffic is enforced by preventing the start of any transmission that would not finish within the respective window. In the synchronous phase, this is enforced by the traffic scheduler so that all the messages specified in the EC-schedule always fit within the maximum width, LSW. In the asynchronous window, this isolation is enforced by an adequate runtime mechanism that is technology dependent. In FTT-CAN, a timer is set to expire slightly before the end of the window and a possibly pending request that cannot be served up to completion within that interval is removed from the network controller transmission buffer. In FTT-Ethernet, the master polls asynchronous transmission requests that can be served within that window, only. In both cases, this mechanism may lead to the insertion of a short amount of idle-time at the end of the asynchronous window (α in Figure 19.3).

It is also important to notice that there could have been idle-time inserted by the synchronous scheduler in a given EC to guarantee that the synchronous traffic does not extend beyond the end of the synchronous window. However, that amount of time is not considered in the value of $lsw(n)$, just the effective transmissions, thus being implicitly reclaimed for the asynchronous traffic. This fact will also have an impact on the response-time of this type of traffic as shown further on.

19.4 The Synchronous Messaging System

In the FTT architecture, the time-triggered traffic is centrally scheduled by the master and conveyed within the synchronous phase of the ECs. Access to synchronous communication is carried out via the SMS, which offers services to the application based on the producer-consumer model [20]. The SMS handles the synchronous traffic with autonomous control, i.e., the transmission and reception of messages is carried out exclusively by the network interface without any intervention from the application software. The message data is passed to and from the network by means of shared buffers. This means that the network interface, in what concerns the SMS, behaves as a temporal firewall between the application and the network, since it isolates the temporal behavior of both parts, increasing the system robustness [21]. There are two main SMS services: *SMSproduce*, i.e., writing a message in the appropriate buffer in the network interface; and *SMSconsume*, i.e., reading from a message buffer in the network interface. For both services, there is an option to synchronize with the network traffic, which allows controlling the cyclic execution of application software within remote nodes simply by adjusting the periodicity of the respective messages. Moreover, it is possible to create virtual messages, i.e., with $C = 0$, whose purpose is only to synchronize application software, e.g., to trigger certain tasks in the nodes. These are generically called *triggers* and they are encoded in the EC-schedule as normal synchronous messages but cause no transmissions, thus using no bandwidth [14]. These mechanisms allow synchronizing the application software with the network schedule thus supporting a global system management policy named network-centric [3].

Additionally, the SMS also provides the services required to manage the SRT (Equation 19.3) such as *SRTadd*, *SRTremove*, and *SRTchange*. These services automatically invoke an online admission control to

assure a continued timely communication. However, for particular applications where such a feature is not required, e.g., when changes in the SRT at runtime are not required, then the online admission control can be disabled, saving unnecessary overhead.

19.4.1 Synchronous Scheduling Model

The synchronous scheduling is carried out in the master node, online, on an EC basis. The scheduling policy is decoupled from the native MAC protocol of the underlying network due to the master–slave operation. Thus, arbitrary scheduling policies may be deployed, which is another aspect of the flexibility offered by the FTT protocols. For example, earliest deadline first (EDF) scheduling can be easily deployed in CAN using FTT-CAN, overriding the fixed priorities of the native MAC that are associated to the message identifiers. Other techniques to implement EDF over CAN using dynamic manipulation of message identifiers [31,45] present several drawbacks such as the need for explicit clock synchronization among nodes, relatively inefficient and heavy computations in all nodes, a reduced number of bits to encode the dynamic priority, and the need to cyclically dequeue messages to update their identifiers [33].

The main features of the scheduling model used for the synchronous traffic have already been referred in Section 19.3.4. Basically, the traffic scheduler prevents transmissions of messages to cross the boundary of the synchronous window of a given EC it is delayed to the following ones. However, the use of inserted idle-time has a negative impact on the synchronous traffic schedulability, since it corresponds to a reduction in the effective window length (see further on). Moreover, all messages have periods, deadlines, and offsets that are integer multiples of the EC duration (Equation 19.4) and their release, i.e., when they become ready for transmission, is always synchronous with the start of an EC. The transmission time of each message is also substantially shorter than the maximum length of the synchronous window ($\forall_{i=1, \dots, N_s} C_i \ll \text{LSW}$), which means that several messages fit in a window. Finally, unless stated otherwise, we will assume that all synchronous messages are independent of each other.

19.4.2 Schedulability Analysis of Synchronous Traffic

The scheduling model used in the synchronous phase of the FTT protocols was generalized and the scheduled object was abstracted away resulting in the so-called blocking-free nonpreemptive scheduling model [4], which is found in several systems of both tasks and messages. Thus, the specific case of the FTT synchronous scheduling is just an instance of that model, which considers that the whole cycle, with duration E , is available to execute tasks, while in FTT protocols the synchronous traffic is restricted to the synchronous window within each EC, with maximum length LSW.

To transform the FTT model into the one used in Ref. 4, so that the analysis therein presented can be used, it suffices to inflate all execution times by a factor equal to E/LSW . This is equivalent to expanding the synchronous window up to the whole EC (Figure 19.4) and carries no consequence in terms of schedulability since messages scheduled for a given synchronous window will remain within the same cycle. Applying this transformation to the original set of messages in the SRT (Equation 19.1) results in a new virtual set that can be expressed as SRT^0 (Equation 19.6) in which all the remaining parameters but the execution times are kept unchanged (remember that the schedulability of SRT^0 is equivalent to the one of SRT).

Then, according to Ref. 4, we do a further transformation in this message set, obtaining the set SRT' (Equation 19.7), which results from applying a further inflation to the execution times, to cover any possibly existing inserted idle-time. Notice that X^0 is the maximum inserted idle-time in the schedule of SRT^0 .

$$\text{SRT}^0 \equiv \{SM_i^0(C_i^0, Ph_i, P_i, D_i, Pr_i), C_i^0 = E/\text{LSW} * C_i, i = 1, \dots, N_s\} \quad (19.6)$$

$$\text{SRT}' \equiv \{SM_i'(C_i', Ph_i, P_i, D_i, Pr_i), C_i' = C_i^0 * E/(E - X^0), i = 1, \dots, N_s\}, \quad X^0 = \max_n (X_n^0) \quad (19.7)$$

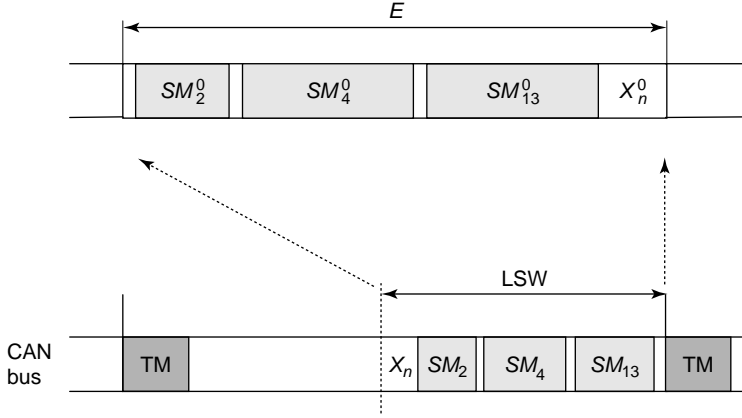


FIGURE 19.4 Expanding the synchronous window to allow using the blocking-free nonpreemptive model.

The results in Ref. 4 are now directly applicable to SRT^0 , particularly Theorem 19.1 that states:

Theorem 19.1 (Equation 19.3 in Ref. 4)

If the message set in SRT' is guaranteed to be schedulable by at least one sort of analysis for fixed priority preemptive scheduling, then SRT^0 is also schedulable.

The proof of this theorem relies on the verification that the finishing times of all tasks in the schedule of SRT' , with preemption and no inserted idle-time, are always later than the corresponding finishing times in the schedule of SRT^0 , without preemption but using inserted idle-time according to the referred model.

Expanding the execution times C'_i in Equation 19.7 with the transformation in Equation 19.6 and noting that $X_0 = E/LSW * X$, where X is the maximum inserted idle-time in the schedule of SRT , i.e., $X = \max_n(X_n)$, yields the final transformation (Equation 19.8) that has to be carried out over the original message transmission times, i.e., those in the SRT , so that any existing analysis for fixed priorities preemptive scheduling can be used.

$$C'_i = C_i * E / (LSW - X) \quad (19.8)$$

However, any schedulability assessment obtained via this theorem is just sufficient, only, because of the pessimism introduced when using an upper bound for X . Except for a few particular situations, the exact value $X = \max_n(X_n)$ cannot be easily determined but an upper bound is easy to obtain such as the transmission time of the longest message among those that can cause inserted idle-time. This is obtained with $\max_{j=k, \dots, N_s} (C_j)$ in Expression 19.9, considering the message set sorted by decreasing priorities.

$$\begin{aligned} \max_{j=k, \dots, N_s} (C_j) &\geq X = \max_n (X_n) \\ k : \sum_{i=1}^{k-1} C_i &\leq LSW \wedge \sum_{i=1}^k C_i > LSW \end{aligned} \quad (19.9)$$

An important corollary of the theorem referred above is that Liu and Layland's utilization bound for rate monotonic (RM) [25] can be used with just a small adaptation as part of a simple online admission control for changes in the SRT incurring in very low runtime overhead. This is stated in Condition C1.

$$U = \sum_{i=1}^{N_s} \left(\frac{C_i}{P_i} \right) < N_s \left(2^{\frac{1}{N_s}} - 1 \right) \left(\frac{LSW - X}{E} \right) \Rightarrow \text{SRT is schedulable with RM under any phasing} \quad (C1)$$

The same transformation can be applied to the hyperbolic bound [9] or even to the response time analysis for fixed priorities preemptive scheduling [8].

A similar line of reasoning can be followed to adapt the Liu and Layland's utilization bound for EDF. In this case, the maximum inserted idle-time (X) plus the remaining amount of time in the EC outside the synchronous window ($E - \text{LSW}$) can be considered as the worst-case transmission time of a virtual message ($C_v = E - \text{LSW} + X$) that is added to the original set and transmitted every EC ($P_v = E$). This virtual message will be the highest priority one in every EC and will fill in the part of the EC that cannot be used by the synchronous messages. Assume, now, that the resulting extended set, i.e., $\text{SRT} \cup (C_v, P_v)$, can be scheduled preemptively. In this situation, the Liu and Layland's bound can be used.

$$U_v = \frac{E - \text{LSW} + X}{E} + \sum_{i=1}^{N_s} \left(\frac{C_i}{P_i} \right) \leq 1 \quad (19.10)$$

However, owing to the extra load imposed by the virtual message, all other messages will finish transmission either in the same EC or later in this schedule than in the original one with the traffic confined to the synchronous window and with inserted idle-time. Thus, if the extended set is schedulable the SRT will also be. This results in the sufficient schedulability Condition C2.

$$U = \sum_{i=1}^{N_s} \left(\frac{C_i}{P_i} \right) \leq \frac{\text{LSW} - X}{E} \Rightarrow \text{SRT is schedulable with EDF} \quad (\text{C2})$$

under any phasing

Finally, it is interesting to notice that the analysis presented above, either for fixed priorities or EDF, also relates to the hierarchical scheduling framework with periodic servers, since it allows to determine the schedulability of a task/message set executing within a predefined periodic partition with a given length and period (LSW and E , respectively, in this case).

19.4.3 Adding Dynamic QoS Management

In its basic functionality level, the FTT paradigm requires change requests to be handled by an online admission control. The purpose of this mechanism is to assess, before commitment, if the requests can be accommodated by the system. From this point of view, the master node can be seen as a QoS server in the sense that, when a message is admitted or changed, the master node verifies if its associated requirements (memory, network bandwidth, message deadline and jitter, etc.) can be fulfilled, and in this case it also reserves these resources in a way that they will be strictly available in the future, assuring that all the accepted messages will receive the requested QoS.

However, as discussed in Section 19.2, several emerging real-time applications can benefit from a high degree of operational flexibility. This is reflected as a set of flexible QoS requirements that must be specified by the application. Then, the specific QoS level provided to the application changes at runtime according to environment or configuration changes. To handle this level of flexibility efficiently, timely, and safely, the communication protocol should not only guarantee that the minimum requirements will be fulfilled in all anticipated conditions, but also grant the highest QoS possible to all the activities or streams, at all instants, distributing adequately the available resources among them.

The FTT paradigm can provide support to this kind of dynamic QoS management in distributed systems aggregating a QoS manager to the online admission control component (Figure 19.5). With this architecture, the online admission control still decides about the acceptance of change requests based on the minimum requirements of the existing message streams. This will eventually generate some spare resources, e.g., spare bandwidth, that will be distributed by the QoS manager according to a predefined policy.

The FTT paradigm is based on a modular design, with well-defined interfaces between the system components. In this case, the QoS manager, triggered by environmental or configuration changes, executes

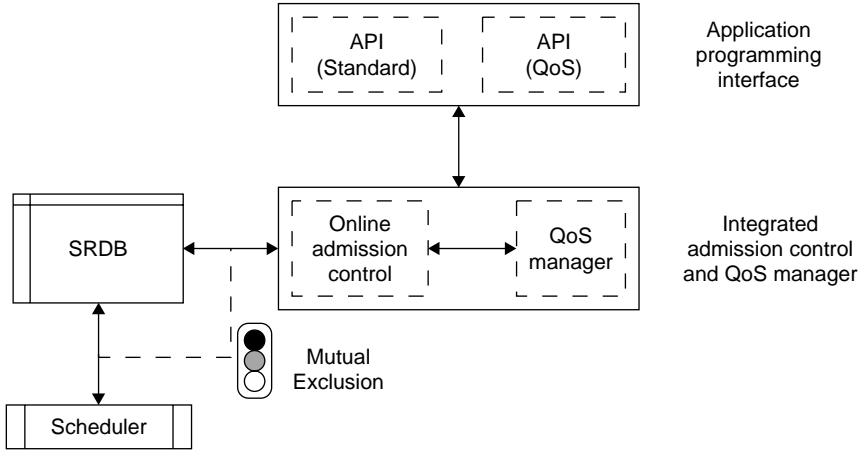


FIGURE 19.5 Adding QoS management to FTT.

its internal policy and deduces the corresponding message properties, such as periods and deadlines, which are then written onto the SRT. The traffic scheduler, in its turn, reads the SRT and enforces the specific scheduling policy. These two activities are decoupled except for a concurrent access control to prevent the scheduler from reading the SRT while being updated, to avoid inconsistencies. Thus, the traffic scheduler operates independently of the QoS management policy and can also operate in the absence of the QoS manager. Within the SRT, as described in Section 19.3.3, the QoS parameters, such as admissible period range, relative importance, and criticalness, are stored in the $*Xf_i$ field.

With respect to the application interface, the integrated online admission control and QoS manager implement the standard SRDB management functions for adding, deleting, and changing message streams plus the functions to provide user-level QoS management, such as updating the QoS parameters or requesting specific QoS levels.

19.4.4 QoS Management Example Based on the Elastic Task Model

The Elastic Task Model, proposed in Ref. 1, treats the utilization of tasks as an elastic parameter whose value can be modified by changing the period within a specified range. Each task τ_i is characterized by five parameters (Equation 19.11): a worst-case computation time C_i , a nominal period T_{i0} , a minimum period T_{i_min} , a maximum period T_{i_max} , and an elastic coefficient E_i .

$$\tau_i(C_i, T_{i0}, T_{i_max}, T_{i_min}, E_i) \quad (19.11)$$

Upon a change request with respect to one task, e.g., adding a new task or changing the period of an existing one, the utilization of all other tasks is adapted to accommodate the request and still maximize the global utilization of the task set. The elastic coefficients allow distributing the spare or needed bandwidth (utilization) asymmetrically among the tasks so that those with higher coefficient adapt more. The variation in the utilization of each task is proportional to its elastic coefficient but bounded by the maximum and minimum period values. Admission of new tasks, as well as requests for changes of existing ones, are always subject to an elastic guarantee, i.e., they are accepted only if there exists a feasible schedule in which all the other periods are within their validity range.

Initially, a schedulability test is executed over the minimum QoS requirements (T_{i_max}) to assess whether there are resources available to satisfy the minimum QoS level. If such test passes, the next step is to check if all tasks can be granted the highest possible QoS (T_{i_min}). If this test fails, the elastic algorithm is used to adapt the tasks periods to a value T_i such that $\sum (C_i/T_i) = U_d$, where U_d is a desired global utilization factor. The elastic algorithm consists in computing by how much the task set must be compressed with

respect to the nominal global utilization U_0 , i.e., determining $(U_0 - U_d)$ and then determining how much each task must contribute to this value according to its elastic coefficient.

$$\forall i, \quad U_i = U_{i_nom} - (U_0 - U_d) \frac{E_i}{E_v} \quad \text{and} \quad E_v = \sum_{i=1}^n E_i \quad (19.12)$$

However, owing to the period bounds, i.e., $T_{i_min} \leq T_i \leq T_{i_max}$, the problem of finding the set of T_i becomes nonlinear and requires an iterative solution, with a new iteration each time a task reaches its period bound.

When applying the elastic task model to the FTT protocols, the message characterization is extended with a minimum, nominal, and maximum periods as well as elastic coefficient, i.e., P_{min_i} , P_{nom_i} , P_{max_i} , and E_i , respectively.* Then, a few adaptations are needed to cope with nonpreemption and inserted idle-time, as well as with the coarse temporal resolution available to specify the message periods, deadlines, and offsets, which are integer multiples of the EC duration E . Concerning the nonpreemption and inserted idle-time, an adequate schedulability test must be used, such as Condition C2 referred in Section 19.4.2. Then, the periods resulting from applying Equation 19.12 are not necessarily multiples of the EC duration and thus must be rounded to their higher nearest integer multiples as in the following equation:

$$\forall_{i=1, \dots, N_s} \quad U'_i = \sum_{i=1}^{N_s} \frac{C_i}{P'_i} \quad \text{with} \quad P'_i = \left\lceil \frac{P_i}{E} \right\rceil * E = \left\lceil \frac{C_i}{U_i * E} \right\rceil * E \geq P_i \quad (19.13)$$

After rounding the periods, the final utilization may fall well below the desired total utilization, U_d . Therefore, to improve efficiency and maximize the use of the desired total utilization, the elastic task model was extended with an additional optimization step, performed after the initial compression algorithm and discretization in which the spare utilization factor caused by the discretization of the periods is better distributed among the messages. This process is useful for all situations in which the periods of the tasks are discrete, which is common in many systems, e.g., in those based on tick scheduling.

The optimization step, which tries to follow the elastic paradigm in the distribution of the spare utilization, is shown in Figure 19.6 and allows calculating a succession of effective utilization values, $U'_{eff}(n)$, that come closer to the desired value U_d . The process starts with $U'_{eff}(0)$ that corresponds to the total utilization obtained with the periods computed from Equation 19.12 and rounded according to Equation 19.13. Then, the process continues with the computation, for all messages, of their individual utilization, U_i^+ , that would result from reducing the period by one EC, if possible. Then, the difference ΔU_i is calculated for all messages, too, representing the potential of each message to increase the global utilization.

$$\Delta U_i = U_i^+ - U'_i \quad \text{with} \quad U_i^+ = \min \left(\frac{C_i}{P'_i - E}, \frac{C_i}{P_{min_i}} \right) \quad (19.14)$$

The process follows building a vector that contains in position i the value of ΔU_i and the global utilization U_i^+ calculated according to Equation 19.15, which corresponds to the total utilization that

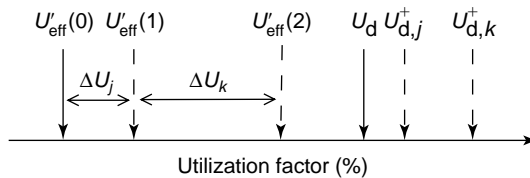


FIGURE 19.6 Improving the effective utilization factor in systems with discrete periods.

*Notice that E_i will be used to refer to an elastic coefficient while E , with no index, still refers to the EC duration.

would result if the period of message i was reduced and its utilization weighted according to the respective elastic coefficient.

$$\forall i=1, \dots, N_S \quad U_{d,i}^+ = U_d + \Delta U_i * \frac{E_v}{E_i} \quad (19.15)$$

The vector $\{(\Delta U_i, U_{d,i}^+), i = 1, \dots, N_S\}$ is sorted in the ascending order of $U_{d,i}^+$ utilizations (Figure 19.16). Then, for each element i in the sorted vector, starting from the side of lower $U_{d,i}^+$, if $U'_{\text{eff}}(n) + \Delta U_i < U_d$ then $U'_{\text{eff}}(n+1) = U'_{\text{eff}}(n) + \Delta U_i$ and the period of message i is effectively reduced by one EC. The same test and operation are performed through the whole sorted vector, carrying out the necessary period updates, resulting in a global utilization that is closer to the desired value U_d .

19.5 The Asynchronous Messaging System

The asynchronous traffic is handled by the FTT protocols in the asynchronous phase of the ECs and it is meant to support event-triggered communication with better bandwidth efficiency than possible with a pure time-triggered approach. This type of communication is handled by the AMS, which provides simple send and receive services with queueing, namely the nonblocking immediate *AMSSend* and the blocking *AMSReceive* functions. More complex and reliable exchanges, e.g., requiring acknowledge or requesting data, must be implemented at the application level, using the two basic services referred above. The length of the queues is set up at configuration time, both in the sender and in the receiver sides, according to the number of asynchronous message streams that a node may send or receive, respectively, as well as the number of messages of the same stream that may accumulate in the network interface between transmissions or retrievals.

The particular implementation of the AMS services is dependent on the network technology used and thus it is different among the FTT protocols. With respect to the AMS, the only aspect that is transversal to all FTT implementations is the confinement of this type of traffic to the asynchronous window within the EC, to enforce temporal isolation between different traffic classes. Currently, the FTT-CAN implementation provides an AMS that maps directly on CAN except for the unavailability periods corresponding to the time outside the asynchronous window within each EC. This is very convenient since it allows exploiting all the interesting properties of CAN for event-triggered traffic, namely the asynchronous bus access mechanism, i.e., a node tries to transmit as soon as a send request is issued, and the fixed priorities-based arbitration, relying on the CAN frame identifier. Particular care has been taken to assure that there are no extra penalties in terms of priority inversions arising from the interruption of continuous time in a succession of asynchronous windows. In contrast, FTT-Ethernet uses a polling mechanism that is less bandwidth efficient but is collision-free and which follows a round-robin order. This order can, however, be overridden by different polling sequences that allow scrutinizing some nodes more often than others, resulting in an asymmetrical bandwidth distribution. Moreover, as referred in Section 19.3, the order of phases in the EC can also vary according to the specific protocol implementation.

This section focuses on a generic priorities-based scheduling of asynchronous requests, such as the one implemented on FTT-CAN, which is particularly suited to support asynchronous real-time traffic. This kind of traffic can be application related, e.g., alarms, and also protocol related, such as the management mechanisms of the SMS and the global FTT framework, namely to convey the change requests for synchronous streams and the communication related with replicated master synchronization mechanisms.

Therefore, in the following sections we will present analytical results that allow determining the worst-case response time of such transmissions and thus verifying their adequate timing. Moreover, we will consider that the asynchronous window precedes the synchronous one, as in FTT-CAN. However, adapting the analysis for a different sequence of windows in the EC, such as in FTT-Ethernet, is trivial.

19.5.1 Asynchronous Traffic Model

As referred before, this type of traffic arrives at the network interface asynchronously with respect to the cyclic EC framework. It might be queued at the interface waiting for bus clearance for transmission or be transmitted immediately if the bus is available at that moment. Moreover, we consider a local queueing policy that is coherent with the bus traffic scheduling, which is based on fixed priorities. An important detail is that the message scheduler verifies for each scheduled message whether it can complete transmission before the end of the current window. If it cannot, that message is kept in the local queue, the asynchronous window is closed, i.e., idle-time is inserted until the end of the window, and the message scheduling (or arbitration as in CAN) is resumed at the beginning of the next asynchronous window. This means that the scheduling (arbitration) process is logically continuous despite the interruptions between subsequent windows.

The analysis for this model was initially presented in Ref. 32 and further improved in Ref. 6. It follows closely the one in Ref. 44 for the original CAN protocol but introduces a few modifications to allow coping with inserted idle-time and exclusions. Later, it was generalized for preemptive tasks running in a periodic server resulting in a contribution for the hierarchical scheduling framework [5]. The analysis presented hereafter differs from other analysis for hierarchical task scheduling [24,41] since it considers the impact of: (a) nonpreemption in the transmission of messages; and (b) a variable server capacity, $\text{law}(n)$, arising from the bandwidth reclaiming mechanism that allows the AMS to use the time left free by the SMS in each cycle as described in Section 19.3.4. The effective server capacity in each cycle can be deduced from the expression on the right in Equation 19.5 after knowing the effective length of the synchronous window. However, as shown in Ref. 7, the possible use of inserted idle-time in the construction of the EC-schedules can lead to anomalies in the determination of the critical instant for the asynchronous traffic. To circumvent such anomaly, the worst-case timing analysis must consider $\text{law}(n)$ computed according to Equation 19.16, where C_S is the longest synchronous message, corresponding to an upper bound to the idle-time possibly inserted in the respective synchronous window.

$$\text{law}(n) = \begin{cases} E - \text{LTM} - \text{lsw}(n), & \text{LSW} - \text{lsw}(n) \geq C_S \\ E - \text{LTM} - \text{LSW}, & \text{LSW} - \text{lsw}(n) < C_S \end{cases} \quad (19.16)$$

19.5.2 Schedulability of Asynchronous Traffic

The set of real-time asynchronous communication requirements is held in a table named asynchronous requirements table (ART) (Equation 19.2). Notice that the AMS is also used to convey non-real-time traffic as specified in the NRT (Equation 19.3). This traffic, however, has lower priority and we will account for its impact considering the generic message AM^{NRT} , which has the longest length among the non-real-time messages.

The following analysis does not consider message queueing, at the sender, neither dequeueing at the receiver. The response time to a transmission request for message AM_i is defined as the time lapse from the request instant until complete transmission and it is considered as composed of three parts (Equation 19.17). The parameter σ_i is called dead interval and corresponds to the first exclusion period, between the request and the start of the following asynchronous service window (instant O in Figure 19.7). The parameter w_i , known as *level- i busy window*, allows accounting for exclusions as well as for the interference caused by higher-priority messages within the asynchronous service windows until message AM_i effectively starts transmission.

$$R_i = \sigma_i + w_i + C_i \quad (19.17)$$

An upper bound to the worst-case response time for asynchronous message AM_i , i.e., Rwc_i^a , is obtained by using upper bounds for both σ_i and w_i (Figure 19.7). The first one (σ^{ub}) can be obtained with Expression 19.18 considering that Ca represents the transmission time of the longest asynchronous message. Notice that both the transmission time of AM^{NRT} and the inserted idle-time α last for no longer than Ca .

$$\sigma^{ub} = 2 \times Ca + \text{LSW} + \text{LTM} \quad (19.18)$$

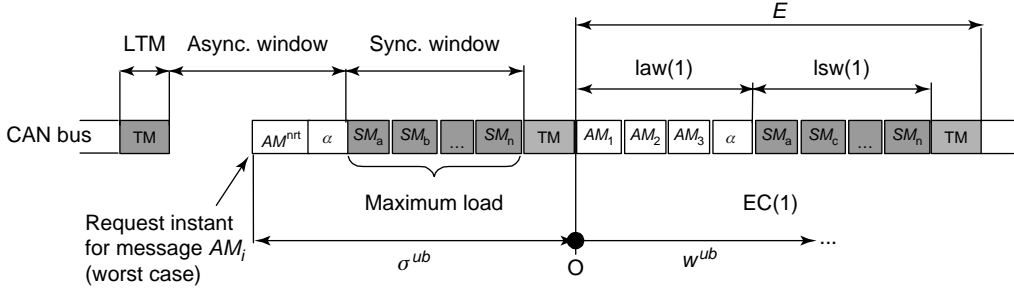


FIGURE 19.7 Maximum dead interval (σ_i) and level- i busy window (w_i).

The upper bound for the level- i busy window, w_i^{ub} , is obtained considering that, at instant O , the message suffers the maximum interference from all higher-priority asynchronous messages and the AMS starts a sequence of minimum service (maximum service requested by the SMS). The level- i busy window can be determined via an iterative process similar to the one in Ref. 44 using a cumulative bus demand function, $H_i(t)$, of the asynchronous traffic with higher priority than AM_i , i.e., hp_i .

$$H_i(t) = \sum_{j \in hp_i} \left\lceil \frac{t + \sigma^{ub}}{\text{mit}_j} \right\rceil C_j \quad (19.19)$$

However, in this case, not the whole bus time is available for this type of traffic but just the AMS service windows. We can, thus, define a lower bound for the service function of the asynchronous traffic, named $A(t)$ (Equation 19.20), to account for both the inserted idle-time, upper bounded by Ca , and the exclusion periods in between the asynchronous windows.

$$A(t) = \begin{cases} \sum_{j=1}^{n-1} (\text{law}(j) - Ca) + (t - (n-1)E) & (n-1)E < t \leq (n-1)E + (\text{law}(n) - Ca) \\ \sum_{j=1}^{n-1} (\text{law}(j) - Ca) & (n-1)E + (\text{law}(n) - Ca) < t \leq nE \end{cases} \quad (19.20)$$

with $n-1 = \lfloor \frac{t}{E} \rfloor$.

The upper bound for w_i (Figure 19.8) is then obtained as the first instant in time, counted after instant O , i.e., end of the dead interval in which the lower bound on the AMS service equals the upper bound on the higher-priority asynchronous traffic demand.

$$w_i^{ub} = \text{earliest } t : H_i(t) = A(t) \quad (19.21)$$

This equation can be solved iteratively by using $t^1 = H_i(0)$ and $t^{m+1} = A^{\text{inv}}(H_i(t^m))$ (Figure 19.8), where $A^{\text{inv}}(t)$ stands for the inverse of $A(t)$.^{*} The process stops when $t^{m+1} = t^m$ (and $w_i^{ub} = t^{m+1}$) or $t^{m+1} > D_i - C_i - \sigma^{ub}$ (deadline cannot be guaranteed). It is easy to verify that, given the lower bound on the increments in $H_i(t)$, the process either converges or crosses the deadline within a finite number of steps.

An upper bound to the worst-case response time for message AM_i , i.e., Rwc_i^a , can be obtained through Equation 19.17, replacing w_i by w_i^{ub} obtained from Equation 19.21, and σ_i by σ^{ub} obtained from Equation 19.18. Knowing the upper bounds for the worst-case response times of all asynchronous

^{*}Formally, $A^{\text{inv}}(t)$ is not defined for $t = \sum_{n=1}^m (\text{law}(n) - Ca)$ for any value of m . In those cases, we consider $A^{\text{inv}}(t = \sum_{n=1}^m (\text{law}(n) - Ca)) = (m-1) * E + \text{law}(m) - Ca$.

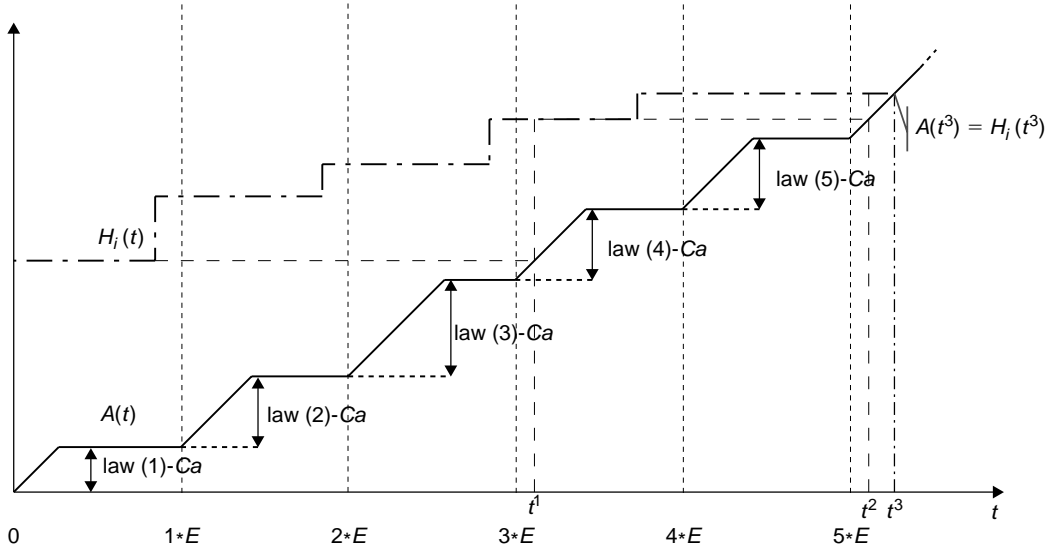


FIGURE 19.8 Calculating the level- i busy window.

messages, a straightforward sufficient schedulability Condition C3 can be derived, consisting on comparing these upper bounds with the respective deadlines.

$$Rwc_i^a \leq D_i, \quad i = 1, \dots, Na \Rightarrow \text{ART is schedulable} \quad (\text{C3})$$

19.6 Case Study: A Mobile Robot Control System

In this section, we briefly present and analyze the control system of a small autonomous robot that allows illustrating the use of FTT to provide dynamic QoS management [2]. The robot must exhibit obstacle avoidance, path following and beacon tracking behaviors, using a subsumption architecture [11] to arbitrate among them and set the active one. The behavior arbitration is as follows: if obstacle detected, avoid it; else if line on floor, follow it; else if beacon detected, track it, else move randomly.

To support the desired behaviors, the robot is equipped with two independent motors, a set of three proximity sensors to detect nearby objects, a beacon detector, a line sensor made of an array of 10 individual sensors, and a main CPU to execute the global control software (Figure 19.9). These elements are integrated in a distributed system, interconnected by an FTT network. The FTT master is executed in the main CPU, jointly with application tasks. The sensor readings are produced by the respective sensors and consumed by the main CPU. However, the main CPU produces the speed set-points that are consumed by the motor controllers, which execute closed-loop speed control. These controllers also produce displacement measures that are consumed by the main CPU to support trajectory control.

19.6.1 Establishing the Communication Requirements

Table 19.1 characterizes the communication requirements. Each sensor produces a 1-byte message with the respective reading except for the motor controllers that produce a 2-byte message with the displacement information from the encoders. The QoS requirements are expressed in terms of admissible ranges for the production rates of each message.

Since the specified periods are integer multiples of 10 ms, this value has been used to define the EC duration. Moreover, the maximum width of the synchronous window was set to 80% of the EC, leaving 20% for the TM as well as for asynchronous traffic, not defined here. The network is CAN at 100 Kbps resulting in the transmission times, minimum and maximum network utilizations in Table 19.2.

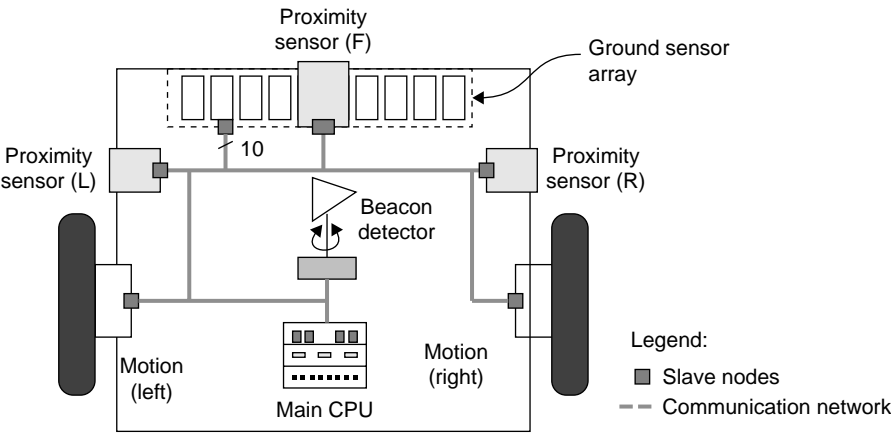


FIGURE 19.9 Robot architecture.

TABLE 19.1 Message Set and Properties

Source	Signal Name	Data Bytes	No. of Messages	Period (ms)	
				Min	Max
Obstacle sensors	OBST _{1..3}	1	3	10	50
Line sensors	LINE _{1..10}	1	10	10	1000
Beacon sensor	BCN_INT	1	1	200	2000
	BCN_ANG	1	1	50	200
Main CPU	SPEED _{1..2}	1	2	10	150
Motor	DISP _{1..2}	2	2	20	500

TABLE 19.2 Network Utilization

Signal Name	Tx Time (μs)	No. of Messages	Period (EC)		Utilization (%)	
			Min	Max	Min	Max
OBST _{1..3}	650	3	1	5	3.90	19.50
LINE _{1..10}	650	10	1	100	0.65	65.00
BCN_INT	650	1	20	200	0.03	0.33
BCN_ANG	650	1	5	20	0.33	1.30
SPEED _{1..2}	650	2	1	15	0.87	13.00
DISP _{1..2}	750	2	2	50	0.26	6.50
Total utilization (%)					6.07	106.63

Considering EDF scheduling and applying the analysis in Section 19.4.2, the upper bound for guaranteed traffic schedulability is 73.5% of the network bandwidth (note that LSW = 80%), which is well above the minimum required utilization but also below the respective maximum requirement. Thus, it is not possible to transmit all the messages at their highest rates but, if the lowest rates are used, there is a significant spare bandwidth. This gives room for QoS management to assign the spare bandwidth to specific message streams according to effective needs, increasing the respective QoS delivered to the application.

19.6.2 Using Online Adaptation

To better understand the use of dynamic QoS management, notice that the robot needs permanently updated information from all sensors but it executes only one behavior at a time (subsumption architecture). Therefore, the communication system should deliver the highest QoS to the active behavior,

TABLE 19.3 Network Utilization for Three Operational Scenarios with Different Active Behaviors

Signal Name	Obstacle Avoidance			Path Following			Beacon Tracking		
	P_{nom_i}	E_i	P_i	P_{nom_i}	E_i	P_i	P_{nom_i}	E_i	P_i
OBST _{1..3}	1	1	1	5	10	1	5	5	1
LINE _{1..10}	100	8	3	1	1	2	50	20	2
BCN_INT	100	20	20	200	20	20	30	10	50
BCN_ANG	10	20	5	20	20	10	5	1	8
SPEED _{1..2}	1	1	1	2	1	1	1	1	1
DISP _{1..2}	4	5	2	10	10	2	2	5	2
Utilization (%)	63.29			73.48			73.44		

increasing the rate of the respective messages. Conversely, inhibited or latent behaviors, may be given lower QoS levels assigning lower transmission rates for the respective messages. For instance, whenever the robot is following a line on the ground, line sensors should be scanned at the highest rate for accurate control but obstacle sensors can be scanned at a lower rate, as long as no near obstacles are detected. Beacon detection is not even relevant in this case. Then if a near obstacle is detected, the robot must switch the active behavior to obstacle avoidance, assigning highest QoS to this behavior and changing the transmission rates of the respective messages.

19.6.3 Experimental Results

The support for dynamic QoS management is achieved using the elastic task model as discussed in Section 19.4.4. Beyond the period bounds defined in Table 19.1 two more parameters are used per message, the nominal period and the elastic coefficient, which correspond to the optimal period within the allowable range and the flexibility given to the QoS manager to adapt the effective periods, respectively. A task running on the main CPU analyzes continuously the sensor readings, determines the active behavior, and generates the QoS adaptation requests. In this case, the requests consist on setting the nominal periods of the messages involved in the active behavior to their minimum value and the respective elastic coefficients to 1 so that they get the highest QoS. Remaining behaviors get a nominal period equal to the maximum value and an elastic coefficient that is a function of the current sensor readings, i.e., the higher the excitation level, the higher the coefficient. This causes the QoS manager to distribute the remaining bandwidth among the remaining behaviors taking into account their current excitation level. Table 19.3 shows three operational scenarios with different active behaviors. The respective QoS parameters are shown together with the effective message periods generated by the QoS manager. The overall network utilization in all three situations is close but below the maximum possible (73.5% in this case), due to the period discretization effect addressed in Section 19.4.4. The results show that the QoS manager is able to dynamically distribute the communication resources, granting a higher QoS to the messages related with the behavior that has more impact on the global system behavior at each instant, i.e., the active one. Care must be taken to avoid an excessive overhead imposed by the QoS manager invocations, e.g., using hysteresis and enforcing a minimum duration of each state.

19.7 Conclusions

During the last decade we have witnessed a progressive trend toward distribution and integration in embedded systems design. Both of these aspects increase the impact of the interconnecting network on the global system properties such as timeliness, dependability, efficiency, and flexibility. This last property, applied to the system operating conditions, has been generating growing interest as a means to improve the system robustness and efficiency by means of increased runtime adaptation. However, building systems

that adapt in a controlled fashion, i.e., keeping timeliness and safety guarantees, requires the use of an adequate underlying infrastructure, from the computing nodes to the network.

In this chapter, the authors presented the FTT paradigm that has been developed recently to support that level of operational flexibility, which enforces global system synchronization and management. This paradigm is centered on the network with a master node scheduling the time-triggered traffic and synchronizing the related activities in the nodes. The scheduling is carried out online as well as the admission control and QoS management. Moreover, event-triggered traffic is also supported, further increasing the flexibility of the framework. Along the past few years, this paradigm has been implemented over CAN, Ethernet, and switched Ethernet, leading to the FTT-CAN, FTT-Ethernet, and FTT-SE protocols, respectively, and extended in several directions to achieve better timeliness and dependability. In this chapter, the authors focus on the schedulability analysis of both the time- and event-triggered communication as well as on the runtime QoS adaptation capabilities. This analysis has been generalized and motivated contributions in the fields of nonpreemptive scheduling with inserted idle-time and hierarchical scheduling. A simple case study concerning the control of a small autonomous robot was also presented, which allowed exposing the use and interest of such runtime QoS adaptation.

References

1. L. Abeni and G. C. Buttazzo. Integrating multimedia applications in hard real-time systems. *IEEE Real-Time Systems Symposium*, pp. 4–13, 1998.
2. L. Almeida. A word for operational flexibility in distributed safety-critical systems. In *Proceedings of IEEE Workshop on Object-Oriented, Real-Time and Dependable Systems (WORDS 2003)*, 2003.
3. L. Almeida and J. Fonseca. FTT-CAN: A network-centric approach for CAN-based distributed systems. In *Proceedings of SICICA 2000, IFAC 4th Symposium on Intelligent Components and Instruments for Control Applications*, 2000.
4. L. Almeida and J. Fonseca. Analysis of a simple model for non-preemptive blocking-free scheduling. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*, 13–15 June, Delft, The Netherlands, 2001.
5. L. Almeida and P. Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT 2004)*, pp. 95–103, 2004.
6. L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Transactions on Industrial Electronics*, 49(6): 1189–1201, 2002.
7. L. Almeida, P. Pedreiras, and R. Marau. Traffic scheduling anomalies in temporal partitions. In *Proceedings of IPES 2006, 5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, 2006.
8. A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8: 284–292, 1993.
9. E. Bini, G. C. Buttazzo, and G. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7): 933–942, 2003.
10. B. Bouyssounouse and J. Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, vol. 3436 of *Lecture Notes in Computer Science*, Springer, Heidelberg, 2005.
11. R. A. Brooks. How to build complete creatures rather than isolated cognitive simulators. *Architectures for Intelligence*, pp. 225–240, 1991.
12. G. Buttazzo and L. Abeni. Adaptive rate control through elastic scheduling. In *Proceedings of IEEE Conference on Decision and Control*, Sydney, Australia, 2000.
13. G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3): 289–302, 2002.
14. M. Calha and J. Fonseca. Adapting FTT-CAN for the joint dispatching of tasks and messages. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, 2002.

15. A. Casimiro and P. Veríssimo. Using the timely computing base for dependable QoS adaptation. In *20th Symposium on Reliable Distributed Systems (SRDS 2001)*, pp. 208–217, 2001.
16. J. D. Decotignie. A perspective on Ethernet as a fieldbus. In *Proceedings of FeT'2001, 4th International Conference on Fieldbus Systems and Their Applications*, Nancy, France, pp. 138–143, 2001.
17. J. Ferreira, L. Almeida, J. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza. Combining operational flexibility and dependability in FTT-CAN. *IEEE Transactions on Industrial Informatics*, 2(2): 95–102, 2006.
18. S. Fischmeister and K. Winkler. Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*, pp. 106–114, 2005.
19. H. Hassan, J. E. Simó, and A. Crespo. Enhancing the flexibility and the quality of service of autonomous mobile robotic applications. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS 2002)*, pp. 213–220, 2002.
20. J.-P. Thomesse and M. L. Chavez. Main paradigms as a basis for current fieldbus concepts. In *Proceedings of FeT'99 (International Conference on Fieldbus Technology)*, Magdeburg, Germany, 1999.
21. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, Dordrecht, 1997.
22. C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QOS in real-time Mach. In *Proceedings of Multimedia*, Japan, 1996.
23. C.-G. Lee, C.-S. Shih, and L. Sha. Online QoS optimization using service class in surveillance radar systems. *Real-Time systems Journal*, 28(1): 5–37, 2004.
24. G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2): 257–269, Apr. 2004.
25. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1): 46–61, 1973.
26. C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1–2): 85–126, 2002.
27. R. Marau, L. Almeida, and P. Pedreiras. Enhancing real-time communication over COTS Ethernet switches. In *Proceedings of the 6th IEEE International Workshop on Factory Communication*, Torino, Italy, 2006.
28. P. Marti. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2002.
29. R. Moghal and M. S. Mian. Adaptive QoS-based resource allocation in distributed multimedia systems. In *Proceedings of Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2003.
30. J. Montgomery. A model for updating real-time applications. *Real-Time Systems*, 27(2): 169–189, 2004.
31. M. Natale. Scheduling the CAN bus with earliest deadline techniques. In *Proceedings of RTSS 2000 (21st IEEE Real-Time Systems Symposium)*, 2000.
32. P. Pedreiras and L. Almeida. Combining event-triggered and time-triggered traffic in FTT-CAN: Analysis of the asynchronous messaging system. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pp. 67–75, 2000.
33. P. Pedreiras and L. Almeida. EDF message scheduling on controller area network. *Computing & Control Engineering Journal*, 13(4): 163–170, 2002.
34. P. Pedreiras and L. Almeida. The flexible time-triggered (FTT) paradigm: An approach to QoS management in distributed real-time systems. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'03)*, pp. 123, 2003.
35. P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency. In *ECRTS'02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, IEEE Computer Society, Washington DC, USA, 2002.

36. P. Pedreiras, P. Gai, L. Almeida, and G. Buttazzo. FTT-Ethernet: A flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems. *IEEE Transactions on Industrial Informatics*, 1(3): 162–172, 2005.
37. D. Prasad, A. Burns, and M. Atkins. The valid use of utility in adaptive real-time systems. *Real-Time Systems*, 25(2–3): 277–296, 2003.
38. A. Qadi, S. Goddard, J. Huang, and S. Farritor. A performance and schedulability analysis of an autonomous mobile robot. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*, pp. 239–248, 2005.
39. D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards adaptive and reflective middleware for network-centric combat systems, crossTalk, Nov. 2001. Available from: <http://www.cs.wustl.edu/~schmidt/PDF/crosstalk.pdf>; Accessed Feb. 21, 2005.
40. C. Shelton and P. Koopman. Improving system dependability with functional alternatives. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, IEEE Computer Society, Washington DC, USA, p. 295, 2004.
41. I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pp. 2–13, 2003.
42. V. Silva, R. Marau, L. Almeida, J. Ferreira, M. Calha, P. Pedreiras, and J. Fonseca. Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In *Proceedings of ETFA'2005 10th IEEE International Conference on Emerging Technologies and Factory Automation, T8 Intelligent Robots and Systems transaction*, Sept. 2005.
43. J. A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4): 751–763, 1996.
44. K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8): 1163–1169, 1995.
45. K. M. Zuberi and K. G. Shin. Scheduling messages on controller area network for real-time CIM applications. *IEEE Transactions on Robotics and Automation*, 13(2): 310–316, 1997.

20

Wireless Sensor Networks

John A. Stankovic
University of Virginia

20.1	Introduction	20-1
20.2	MAC	20-2
20.3	Routing	20-2
20.4	Node Localization	20-4
20.5	Clock Synchronization	20-5
20.6	Power Management	20-5
20.7	Applications and Systems	20-6
	Surveillance and Tracking • Assisted Living Facilities	
20.8	Conclusions	20-9

20.1 Introduction

A wireless sensor network (WSN) is a collection of nodes organized into a cooperative network [10]. Each node consists of processing capability (one or more microcontrollers, CPUs, or DSP chips), may contain multiple types of memory (program, data, and flash memories), have an RF transceiver (usually with a single omnidirectional antenna), have a power source (e.g., batteries and solar cells), and accommodate various sensors and actuators. The nodes communicate wirelessly and often self-organize after being deployed in an *ad hoc* fashion. Systems of 1,000s or even 10,000 nodes are anticipated. Such systems can revolutionize the way we live and work.

Currently, WSNs are beginning to be deployed at an accelerated pace. It is not unreasonable to expect that in 10–15 years that the world will be covered with WSNs with access to them via the Internet. This can be considered as the Internet becoming a physical network. This new technology is exciting with unlimited potential for numerous application areas including environmental, medical, military, transportation, entertainment, crisis management, homeland defense, and smart spaces.

Since a WSN is a distributed real-time system, a natural question is how many solutions from distributed and real-time systems can be used in these new systems? Unfortunately, very little prior work can be applied and new solutions are necessary in all areas of the system. The main reason is that the set of assumptions underlying previous work has changed dramatically. Most past distributed systems research has assumed that the systems are wired, have unlimited power, are not real time, have user interfaces such as screens and mice, have a fixed set of resources, treat each node in the system as very important, and are location independent. In contrast, for WSNs, the systems are wireless, have scarce power, are real time, utilize sensors and actuators as interfaces, have dynamically changing sets of resources, aggregate behavior is important, and location is critical. Many WSNs also utilize minimal capacity devices, which places a further strain on the ability to use past solutions.

This chapter presents an overview of some of the key areas and research in WSNs. In presenting this work, we use examples of recent work to portray the state of art and show how these solutions differ from solutions found in other distributed systems. In particular, we discuss the medium access control

(MAC) layer (Section 20.2), routing (Section 20.3), node localization (Section 20.4), clock synchronization (Section 20.5), and power management (Section 20.6). We also present a brief discussion of two current systems (Section 20.7) to convey overall capabilities of this technology. We conclude in Section 20.8.

20.2 MAC

A MAC protocol coordinates actions over a shared channel. The most commonly used solutions are contention-based. One general contention-based strategy is for a node, which has a message to transmit to test the channel to see if it is busy, if not busy then it transmits, else if busy it waits and tries again later. After colliding, nodes wait random amounts of time trying to avoid recolliding. If two or more nodes transmit at the same time there is a collision and all the nodes colliding try again later. Many wireless MAC protocols also have a doze mode, where nodes not involved with sending or receiving a packet in a given time frame go into sleep mode to save energy. Many variations exist on this basic scheme.

In general, most MAC protocols optimize for the general case and for arbitrary communication patterns and workloads. However, a WSN has more focused requirements that include a local uni- or broadcast, traffic is generally from nodes to one or a few sinks (most traffic is then in one direction), have periodic or rare communication and must consider energy consumption as a major factor. An effective MAC protocol for WSNs must consume little power, avoid collisions, be implemented with a small code size and memory requirements, be efficient for a single application, and be tolerant to changing radio frequency and networking conditions.

One example of a good MAC protocol for WSNs is B-MAC [24]. B-MAC is highly configurable and can be implemented with a small code and memory size. It has an interface that allows choosing various functionality and only that functionality as needed by a particular application. B-MAC consists of four main parts: clear channel assessment (CCA), packet backoff, link layer acks, and low-power listening. For CCA, B-MAC uses a weighted moving average of samples when the channel is idle to assess the background noise and better be able to detect valid packets and collisions. The packet backoff time is configurable and is chosen from a linear range as opposed to an exponential backoff scheme typically used in other distributed systems. This reduces delay and works because of the typical communication patterns found in a WSN. B-MAC also supports a packet-by-packet link layer acknowledgment. In this way only important packets need to pay the extra cost. A low-power listening scheme is employed where a node cycles between awake and sleep cycles. While awake it listens for a long enough preamble to assess if it needs to stay awake or can return to sleep mode. This scheme saves significant amounts of energy. Many MAC protocols use a request to send (RTS) and clear to send (CTS) style of interaction. This works well for *ad hoc* mesh networks where packet sizes are large (1000s of bytes). However, the overhead of RTS-CTS packets to set up a packet transmission is not acceptable in WSNs where packet sizes are on the order of 50 bytes. B-MAC, therefore, does not use an RTS-CTS scheme.

Recently, there has been new work on supporting multichannel WSNs. In these systems, it is necessary to extend MAC protocols to multichannel MACs. One such protocol is MMSN [36]. These protocols must support all the features found in protocols such as B-MAC, but must also assign frequencies for each transmission. Consequently, multifrequency MAC protocols consist of two phases: channel assignment and access control. The details for MMSN are quite complicated and are not described here. In contrast, we expect that more and more future WSNs will employ multiple channels (frequencies). The advantages of multichannel MAC protocols include providing greater packet throughput and being able to transmit even in the presence of a crowded spectrum, perhaps arising from competing networks or commercial devices such as phones or microwave ovens.

20.3 Routing

Multihop routing is a critical service required for WSN. Because of this, there has been a large amount of work on this topic. Internet and MANET routing techniques do not perform well in WSN. Internet

routing assumes highly reliable wired connections so packet errors are rare; this is not true in WSN. Many MANET routing solutions depend on symmetric links (i.e., if node A can reliably reach node B, then B can reach A) between neighbors; this is too often not true for WSN. These differences have necessitated the invention and deployment of new solutions.

For WSN, which are often deployed in an *ad hoc* fashion, routing typically begins with neighbor discovery. Nodes send rounds of messages (packets) and build local neighbor tables. These tables include the minimum information of each neighbor's ID and location. This means that nodes must know their geographic location prior to neighbor discovery. Other typical information in these tables include nodes' remaining energy, delay via that node, and an estimate of link quality.

Once the tables exist, in most WSN routing algorithms messages are directed from a source location to a destination address based on geographic coordinates, not IDs. A typical routing algorithm that works like this is geographic forwarding (GF) [12].

In GF, a node is aware of its location, and a message that it is "routing" contains the destination address. This node can then compute, which neighbor node makes the most progress toward the destination by using the distance formula from geometry. It then forwards the message to this next hop. In variants of GF, a node could also take into account delays, reliability of the link, and remaining energy.

Another important routing paradigm for WSN is directed diffusion [11]. This solution integrates routing, queries, and data aggregation. Here a query is disseminated indicating an interest in data from remote nodes. A node with the appropriate requested data responds with an attribute-value pair. This attribute-value pair is drawn toward the requestor based on gradients, which are set up and updated during query dissemination and response. Along the path from the source to the destination, data can be aggregated to reduce communication costs. Data may also travel over multiple paths increasing the robustness of routing.

Beyond the basics of WSN routing just presented, there are many additional key issues including

- Reliability
- Integrating with wake/sleep schedules
- Unicast, multicast, and anycast semantics
- Real time
- Mobility
- Voids
- Security, and
- Congestion

Reliability. Since messages travel multiple hops it is important to have a high reliability on each link, otherwise the probability of a message transiting the entire network would be unacceptably low. Significant work is being done to identify reliable links using metrics such as received signal strength, link quality index, which is based on "errors," and packet delivery ratio. Significant empirical evidence indicates that packet delivery ratio is the best metric, but it can be expensive to collect. Empirical data also shows that many links in a WSN are asymmetric, meaning that while node A can successfully transmit a message to node B, the reverse link from B to A may not be reliable. Asymmetric links are one reason MANET routing algorithms such as DSR and AODV do not work well in WSN because those protocols send a discovery message from source to destination and then use the reverse path for acknowledgments. This reverse path is not likely to be reliable due to the high occurrence of asymmetry found in WSN.

Integration with wake/sleep schedules. To save power many WSN place nodes into sleep states. Obviously, an awake node should not choose an asleep node as the next hop (unless it first awakens that node).

Unicast, multicast, and anycast semantics. As mentioned above, in most cases a WSN routes messages to a geographic destination. What happens when it arrives at this destination? There are several possibilities. First, the message may also include an ID with a specific unicast node in this area as the target, or the semantics may be that a single node closest to the geographic destination is to be the unicast node. Second, the semantics could be that all nodes within some area around the destination address should receive the message. This is an area multicast. Third, it may only be necessary for any node, called anycast, in the

destination area to receive the message. The SPEED [7] protocol supports these three types of semantics. There is also often a need to flood (multicast) to the entire network. Many routing schemes exist for supporting efficient flooding.

Real Time. For some applications, messages must arrive at a destination by a deadline. Owing to the high degree of uncertainty in WSN it is difficult to develop routing algorithms with any guarantees. Protocols such as SPEED [7] and RAP [16] use a notion of velocity to prioritize packet transmissions. Velocity is a nice metric that combines the deadline and distance that a message must travel.

Mobility. Routing is complicated if either the message source or destination or both are moving. Solutions include continuously updating local neighbor tables or identifying proxy nodes, which are responsible for keeping track of where nodes are. Proxy nodes for a given node may also change as a node moves further and further away from its original location.

Voids. Since WSN nodes have a limited transmission range, it is possible that for some node in the routing path there are no forwarding nodes in the direction a message is supposed to travel. Protocols like GPSR [13] solve this problem by choosing some other node “not” in the correct direction in an effort to find a path around the void.

Security. If adversaries exist, they can perpetrate a wide variety of attacks on the routing algorithm including selective forwarding, black hole, Sybil, replays, wormhole, and denial of service attacks. Unfortunately, almost all WSN routing algorithms have ignored security and are vulnerable to these attacks. Protocols such as SPINS [23] have begun to address secure routing issues.

Congestion. Today, many WSN have periodic or infrequent traffic. Congestion does not seem to be a big problem for such networks. However, congestion is a problem for more demanding WSN and is expected to be a more prominent issue with larger systems that might process audio, video, and have multiple-base stations (creating more cross traffic). Even in systems with a single-base station, congestion near the base station is a serious problem since traffic converges at the base station. Solutions use backpressure, reducing source node transmission rates, throwing out less-important messages, and using scheduling to avoid as many collisions as possible.

20.4 Node Localization

Node localization is the problem of determining the geographical location of each node in the system. Localization is one of the most fundamental and difficult problems that must be solved for WSN. Localization is a function of many parameters and requirements potentially making it very complex. For example, issues to consider include the cost of extra localization hardware, do beacons (nodes, which know their locations) exist and if so, how many and what are their communication ranges, what degree of location accuracy is required, is the system indoors/outdoors, is there line of sight among the nodes, is it a 2D or 3D localization problem, what is the energy budget (number of messages), how long should it take to localize, are clocks synchronized, does the system reside in hostile or friendly territory, what error assumptions are being made, and is the system subject to security attacks?

For some combination of requirements and issues the problem is easily solved. If cost and form factors are not major concerns and accuracy of a few meters is acceptable, then for outdoor systems, equipping each node with GPS is a simple answer. If the system is manually deployed one node at a time, then a simple GPS node carried with the deployer can localize each node, in turn, via a solution called Walking GPS [26]. While simple, this solution is elegant and avoids any manual keying in the location for each node.

Most other solutions for localization in WSN are either range-based or range-free. Range-based schemes use various techniques to first determine distances between node (range) and then compute location using geometric principles. To determine distances, extra hardware is usually employed, for example, hardware to detect the time difference of arrival of sound and radio waves. This difference can then be converted to a distance measurement. In range-free schemes, distances are not determined directly, but hop counts are used. Once hop counts are determined, distances between nodes are estimated using an average distance per hop, and then geometric principles are used to compute location. Range-free solutions are not as

accurate as range-based solutions and often require more messages. However, they do not require extra hardware on every node.

Several early localization solutions include centroid [1] and APIT [5]. Each of these protocols solves the localization problem for a particular set of assumptions. Two recent and interesting solutions are Spotlight [27] and Radio Interferometric Geolocation [20]. Spotlight removes most of the localization code and overhead to a centralized laser device. Spotlight requires line of sight and clock synchronization. Radio interferometric geolocation uses a novel in-network processing technique that relies on nodes emitting radio waves simultaneously at slightly different frequencies. This solution is subject to multipath problems in some deployments and can require many messages. Both of these recent solutions provide a high accuracy in the centimeter range.

20.5 Clock Synchronization

The clocks of each node in a WSN should read the same time within epsilon and remain that way. Since clocks drift over time, they must be periodically resynchronized and in some instances when very high accuracy is required it is even important for nodes to account for clock drift between synchronization periods.

Clock synchronization is important for many reasons. When an event occurs in a WSN it is often necessary to know where and when it occurred. Clocks are also used for many system and application tasks. For example, sleep/wake-up schedules, some localization algorithms, and sensor fusion are some of the services that often depend on clocks being synchronized. Application tasks such as tracking and computing velocity are also dependent on synchronized clocks.

The network time protocol (NTP) [21] used to synchronize clocks on the Internet is too heavyweight for WSN. Placing GPS on every node is too costly. Representative clock synchronization protocols that have been developed for WSN are RBS [3], TPSN [4] and FTSP [19].

In RBS, a reference time message is broadcast to neighbors. Receivers record the time when the message is received. Nodes exchange their recorded times and adjust their clocks to synchronize. This protocol suffers from no transmitter side nondeterminism since timestamps are only on the receiver side. Accuracies are around 30 μ s for 1 hop. This work did not address multihop systems but could be extended.

In TPSN, a spanning tree is created for the entire network. This solution assumes that all links in the spanning tree are symmetric. Then pairwise synchronization is performed along the edges of the tree starting at the root. Since there is no broadcasting as in RBS, TPSN is expensive. A key attribute of this protocol is that the timestamps are inserted into outgoing messages in the MAC layer thereby reducing nondeterminism. Accuracy is in the range of 17 μ s.

In FTSP, there are radio-layer timestamps, skew compensation with linear regression, and periodic flooding to make the protocol robust to failures and topology changes. Both transmission and reception of messages are timestamped in the radio layer and differences are used to compute and adjust clock offsets. Accuracy is in the range of 1–2 μ s.

Considerations in using a clock-synchronization protocol include choosing the frequency of resynchronization, determining if clock drift between synchronization times is required, how to handle the multihop/network problem, and minimizing overhead costs in terms of energy and added network congestion.

20.6 Power Management

Many devices such as Mica2 and MicaZ that are used in WSN run on two AA batteries. Depending on the activity level of a node, its lifetime may only be a few days if no power management schemes are used. Since most systems require much longer lifetime, significant research has been undertaken to increase lifetime while still meeting functional requirements.

At the hardware level, it is possible to add solar cells or scavenge energy from motion or wind. Batteries are also improving. If form factor is not a problem then it is also possible to add even more batteries. Low power circuits and microcontrollers are improving. Most hardware platforms allow multiple power-saving states (off, idle, on) for each component of the device (each sensor, the radio, the microcontroller). In this way, only the components required at a particular time need to be active.

At the software level, power management solutions are targeted at (i) minimizing communications since transmitting and listening for messages is energy expensive and (ii) creating sleep/wake-up schedules for nodes or particular components of nodes.

Minimizing the number of messages is a cross-cutting problem. For example, with a good MAC protocol there are fewer collisions and retries. With good routing, short paths and congestion avoidance or minimization can be achieved and this minimizes the number of messages sent. Efficient neighbor discovery, time synchronization, localization, query dissemination, and flooding can all reduce the number of messages thereby increasing lifetime.

Solutions to schedule sleep/wake-up patterns vary considerably [33]. Many solutions attempt to keep awake the minimum number of nodes, called sentries, to provide the required sensing coverage while permitting all the others to sleep. To balance energy consumption, a rotation is performed periodically where new sentries are selected for the next period of time. Another common technique is to duty cycle nodes. As an example, a node may be awake for 200 ms out of each second for a 20% duty cycle. The duty cycle percentage chosen depends on application requirements, but the end result is usually very significant savings in energy. Note that duty cycle and sentry solutions can be combined as was done in the VigilNet military surveillance system [6,8].

20.7 Applications and Systems

To demonstrate the capabilities of WSNs, we present two examples of applications and associated systems for those applications.

20.7.1 Surveillance and Tracking

The VigilNet system [8] is a long-lived real-time WSN for military surveillance. The general objective of VigilNet is to alert military command and control units of the occurrence of events of interest in hostile regions. The events of interest are the presence of people, people with weapons, and large and small vehicles. Successful detection, tracking, and classification require that the application obtains the current position of an object with acceptable precision and confidence. When the information is obtained, it is reported to a remote base station within an acceptable latency. VigilNet is an operational self-organizing sensor network (of over 200 XSM mote nodes) to provide tripwire-based surveillance with a sentry-based power management scheme, to achieve minimum 3–6 months lifetime. The tripwire also activates additional external (i.e., out of the Vigilnet system proper) sensors, for example, infrared cameras, only when necessary, thereby also increasing their lifetimes as well.

Figure 20.1 provides an overview of the VigilNet architecture in which there are three categories of components: (1) application components, (2) middleware components, and (3) tinyOS system components. The application components are specially designed for surveillance purposes. It includes (1) an entity-based tracking service; (2) classification components, which provide four types of target differentiation; (3) velocity calculation, which provides target speed and bearing estimation; and (4) false alarm filtering, which differentiates between real and false targets.

Middleware components are designed to be application independent. Time synchronization, localization, and routing comprise the lower-level components and form the basis for implementing the higher-level middleware services, such as aggregation and power management. Time synchronization and localization are important for a surveillance application because the collaborative detection and tracking process rely on the spatiotemporal correlation between the tracking reports sent by multiple motes.

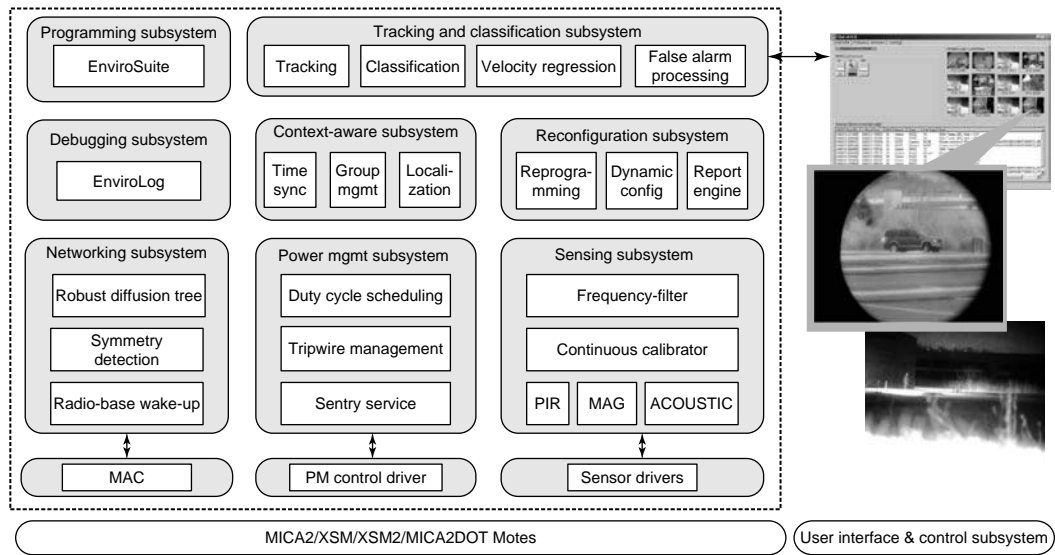


FIGURE 20.1 VigilNet architecture.

The time synchronization module is responsible for synchronizing the local clocks of the motes with the clock of the base station. The localization module is responsible for ensuring that each mote is aware of its location. The configuration module is responsible for dynamically reconfiguring the system when system requirements change. Asymmetric detection is designed to aid the routing module to select high-quality communication links. The radio wake-up module is used to alert nonsentry motes when significant events happen. Power management and collaborative detection are two key higher-level services provided by VigilNet. The sentry service and tripwire management are responsible for power management, while the group management component is responsible for collaborative detection and tracking of events. The sentry and tripwire services conserve energy of the sensor network by selecting a subset of motes, which are defined as sentries, to monitor events. The remaining motes are allowed to remain in a low-power state until an event occurs. When an event occurs, the sentries awaken the other motes in the region of the event and the group management component dynamically organizes the motes into groups to collaboratively track. Together, these two components are responsible for energy-efficient event tracking.

The VigilNet architecture was built on top of TinyOS. TinyOS is an event-driven computation model, written in NesC specifically for the motes platform. TinyOS provides a set of essential components such as hardware drivers, a scheduler, and basic communication protocols. These components provide low-level support for VigilNet modules, which are also written in NesC. Components from TinyOS and VigilNet applications are processed by the NesC compiler into a running executable, which runs (in the VigilNet case) on the XSM (and MICA2) mote platforms.

20.7.2 Assisted Living Facilities

AlarmNet [28,32], a medical-oriented sensor network system for large-scale assisted living facilities, integrates heterogeneous devices, some wearable on the patient and some placed inside the living space. Together they inform the healthcare provider about the health status of the resident. Data are collected, aggregated, preprocessed, stored, and acted upon using a variety of replaceable sensors and devices (activity sensors, physiological sensors, environmental sensors, pressure sensors, RFID tags, pollution sensors, floor sensors, etc.). Multiple-body networks are present in the system. Traditional healthcare provider networks may connect to the system by a residential gateway, or directly to their distributed databases. Some elements of the network are mobile such as the body networks as well as some of the infrastructure network nodes, while others are stationary. Some nodes can use line power, but others depend on batteries.

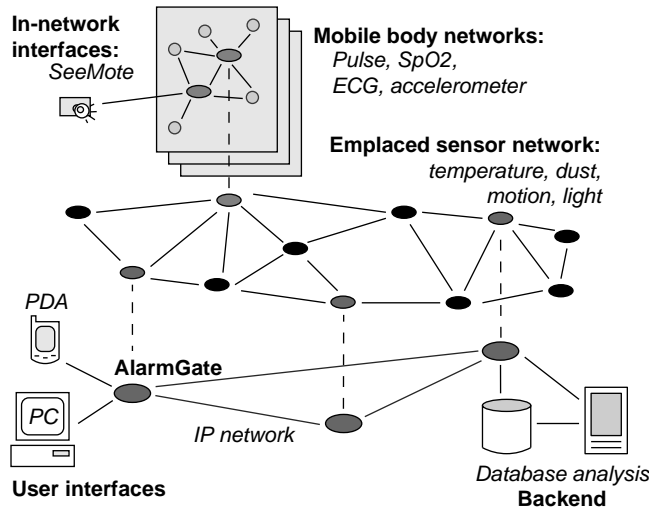


FIGURE 20.2 AlarmNet system architecture.

The system is designed to exist across a large number of living units. The system architecture for AlarmNet is shown in Figure 20.2. Each tier of the architecture is briefly described below.

- *Body networks and Frontends.* The body network is composed of tiny portable devices equipped with a variety of sensors (such as heart rate, heart rhythm, temperature, pulse oximeter, accelerometer), and performs biophysical monitoring, patient identification, location detection, and other desired tasks. Their energy consumption is also optimized so that the battery is not required to be changed regularly. They may use kinetic recharging. Actuators notify the wearer of important messages from an external entity. For example, an actuator can remind an early Alzheimer patient to check the oven because sensors detect an abnormally high temperature. Or, a tone may indicate that it is time to take medication. A node in the body network is designated as the gateway to the emplaced sensor network. Owing to size and energy constraints, nodes in this network have little processing and storage capabilities.
- *Emplaced sensor network.* This network includes sensor devices deployed in the assisted living environment (rooms, hallways, units, furniture) to support sensing and monitoring, including motion, video cameras, temperature, humidity, acoustic, smoke, dust, pollen, and gas. All devices are connected to a more resourceful backbone. Sensors communicate wirelessly using multihop routing and may use either wired or battery power. Nodes in this network may be physically moved and may vary in their capabilities, but generally do not perform extensive calculation or store much data.
- *Backbone.* A backbone network connects traditional systems, such as PDAs, PCs, and in-network databases, to the emplaced sensor network. It also connects sensor nodes by a high-speed relay for efficient routing. The backbone may communicate wirelessly or may overlay onto an existing wired infrastructure. Some nodes such as stargate processors possess significant storage and computation capability, for query processing and location services. Yet, their number, depending on the topology of the building, is minimized to reduce cost.
- *In-network and backend databases.* One or more nodes connected to the backbone are dedicated in-network databases for real-time processing and temporary caching. If necessary, nodes on the backbone may serve as in-network databases themselves. Backend databases are located at the medical center for long-term archiving, monitoring, and data mining for longitudinal studies.
- *Human interfaces.* Patients and caregivers interface with the network using PDAs, PCs, or wearable devices. These are used for data management, querying, object location, memory aids,

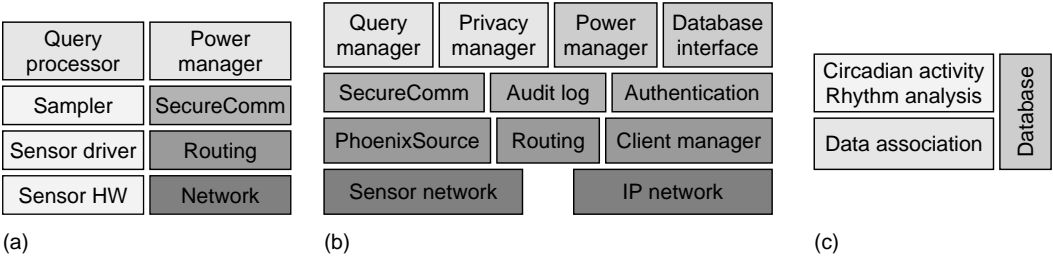


FIGURE 20.3 AlarmNet component software. (a) Sensor device software stack. (b) AlarmGate software stack. (c) Backend analysis and storage.

and configuration, depending on who is accessing the system and for what purpose. Limited interactions are supported with the on-body sensors and control aids. These may provide memory aids, alerts, and an emergency communication channel. PDAs and PCs provide richer interfaces to real-time and historical data. Caregivers use these to specify medical sensing tasks and to view important data.

The software components of the AlarmNet architecture are shown in Figure 20.3. Sensor devices require components for sensing, networking, power management, handling queries, and supporting security. The Stargates implement significantly more functionality including increased security, privacy, query management, and database support. In the backend, many functions are performed including circadian rhythm analysis measuring residents' behaviors on a 24-h basis, data association, security and privacy, and multiple database functions.

20.8 Conclusions

This chapter has discussed WSN issues and example solutions for the MAC layer, routing, localization, clock synchronization, and power management. Why these solutions are different from past networking solutions was stressed. The chapter also provided a short description of two representative WSN systems: a military surveillance, tracking and classification system, and an assisted living facility system.

While these topics are some of the key issues regarding WSN, there are many important topics not discussed in this chapter. For example, security and privacy are critical services needed for these systems [22,23,31]. Programming abstractions and languages for WSN is very active areas of research [14,15,17,29]. Significant and important studies have been collecting empirical data on the performance of WSN [2,30,34,35]. Such data are critical to developing improved models and solutions. Tools for debugging and management of WSN are appearing [18,25]. Several other areas of research are just beginning, but are critical to the eventual success of WSN. These include in-field autocalibration and recalibration of sensors, low-cost signal processing techniques that can be run on microcontrollers with minimum power and memory, and low-cost AI learning schemes to support self-organization, parameter tuning, and calibration.

All of this sensor network research is producing a new technology, which is already appearing in many practical applications. The future should see an accelerated pace of adoption of this technology.

References

1. N. Bulusu, J. Heidemann, and D. Estrin, GPS-Less Low Cost Outdoor Localization for Very Small Devices, *IEEE Personal Communications Magazine*, Vol. 7, No. 5, pp. 28–34, October 2000.
2. A. Cerpa, J. Wong, L. Kuang, M. Potkonjak, and D. Estrin, Statistical Model of Lossy Links in Wireless Sensor Networks, *IPSN*, April 2005.

3. J. Elson, L. Girod, and D. Estrin, Fine-Grained Network Time Synchronization Using Reference Broadcasts, *OSDI*, December 2002.
4. S. Ganeriwal, R. Kumar, and M. Srivastava, Timing-Sync Protocol for Sensor Networks, *ACM SenSys*, November 2003.
5. T. He, C. Huang, B. Blum, J. Stankovic, and T. Abdelzaher, Range-Free Localization and Its Impact on Large Scale Sensor Networks, *ACM Transactions on Embedded Computing System*, Vol. 4, No. 4, pp. 877–906, November 2005.
6. T. He, S. Krishnamurthy, J. Stankovic, T. Abdelzaher, L. Luo, T. Yan, R. Stoleru, L. Gu, G. Zhou, J. Hui, and B. Krogh, VigilNet: An Integrated Sensor Network System for Energy Efficient Surveillance, *ACM Transactions on Sensor Networks*, Vol. 2, No. 1, pp. 1–38, February 2006.
7. T. He, J. Stankovic, C. Lu, and T. Abdelzaher, A Spatiotemporal Communication Protocol for Wireless Sensor Networks, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 10, pp. 995–1006, October 2005.
8. T. He, P. Vicaire, T. Yan, Q. Cao, L. Luo, L. Gu, G. Zhou, J. Stankovic, and T. Abdelzaher, Achieving Long Term Surveillance in VigilNet, *Infocom*, April 2006.
9. T. He, P. Vicaire, T. Yan, L. Luo, L. Gu, G. Zhou, R. Stoleru, Q. Cao, J. Stankovic, and T. Abdelzaher, Real-Time Analysis of Tracking Performance in Wireless Sensor Networks, *IEEE Real-Time Applications Symposium*, May 2006.
10. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, System Architecture Directions for Networked Sensors, *ASPLOS*, November 2000.
11. C. Intanagonwiwat, R. Govindan, and D. Estrin, Directed Diffusion: A Scalable Routing and Robust Communication Paradigm for Sensor Networks, *Mobicom*, August 2000.
12. B. Karp, Geographic Routing for Wireless Networks, PhD Dissertation, Harvard University, October 2000.
13. B. Karp and H. T. Kung, GPSR: Greedy Perimeter Stateless Routing for Wireless Sensor Networks, *IEEE Mobicom*, August 2000.
14. P. Levis and D. Culler, Mate: A Tiny Virtual Machine for Sensor Networks, *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
15. J. Liu, M. Chu, J. J. Liu, J. Reich, and F. Zhao, State-Centric Programming for Sensor and Actuator Network Systems, *IEEE Pervasive Computing*, October 2003.
16. C. Lu, B. Blum, T. Abdelzaher, J. Stankovic, and T. He, RAP: A Real-Time Communication Architecture for Large-Scale Wireless Sensor Networks, *IEEE Real-Time Applications Symposium*, June 2002.
17. L. Luo, T. Abdelzaher, T. He, and J. Stankovic, EnviroSuite: An Environmentally Immersive Programming Framework for Sensor Networks, *ACM Transactions on Embedded Computing Systems*, to appear.
18. L. Luo, T. He, T. Abdelzaher, J. Stankovic, G. Zhou, and L. Gu, Achieving Repeatability of Asynchronous Events in Wireless Sensor Networks with EnviroLog, *Infocom*, April 2006.
19. M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, The Flooding Time Synchronization Protocol, *ACM SenSys*, November 2004.
20. M. Maroti, P. Volgyesi, S. Dora, B. Kusy, A. Nadas, A. Ledeczi, G. Balogh, and K. Molnar, Radio Interferometric Geolocation, *ACM SenSys*, November 2005.
21. D. Mills, Internet Time Synchronization: The Network Time Protocol, In Z. Yang and T. Marsland, editors, *Global States and Time in Distributed Systems*, IEEE Computer Society Press, Los Alamitos, Calif, 1994.
22. A. Perrig, J. Stankovic, and D. Wagner, Security in Wireless Sensor Networks, invited paper, *CACM*, Vol. 47, No. 6, pp. 53–57, June 2004, rated Top 5 Most Popular Magazine and Computing Surveys Articles Downloaded in August 2004, translated into Japanese.
23. A. Perrig, R. Szewczyk, J. Tygar, V. Wen, and D. Culler, SPINS: Security Protocols for Sensor Networks, *ACM Journal of Wireless Networks*, Vol. 8, No. 5, pp. 521–534, September 2002.

24. J. Polastre, J. Hill, and D. Culler, Versatile Low Power Media Access for Wireless Sensor Networks, *ACM SenSys*, November 2004.
25. N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, Sympathy for the Sensor Network Debugger, *ACM SenSys*, November 2005.
26. R. Stoleru, T. He, and J. Stankovic, Walking GPS: A Practical Localization System for Manually Deployed Wireless Sensor Networks, *IEEE EmNets*, 2004.
27. R. Stoleru, T. He, and J. Stankovic, Spotlight: A High Accuracy, Low-Cost Localization System for Wireless Sensor Networks, *ACM SenSys*, November 2005.
28. G. Virone, A. Wood, L. Selavo, Q. Cao, L. Fang, T. Doan, Z. He, R. Stoleru, S. Lin, and J. Stankovic, An Assisted Living Oriented Information System Based on a Residential Wireless Sensor Network, *Proceedings D2H2*, May 2006.
29. M. Welsh and G. Mainland, Programming Sensor Networks with Abstract Regions, *USENIX/ACM NSDI*, 2004.
30. K. Whitehouse, C. Karlof, A. Woo, F. Jiang, and D. Culler, The Effects of Ranging Noise on Multihop Localization: An Empirical Study, *IPSN*, April 2005.
31. A. Wood and J. Stankovic, Denial of Service in Sensor Networks, *IEEE Computer*, Vol. 35, No. 10, pp. 54–62, October 2002.
32. A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic, AlarmNet, *ACM SenSys*, April 2005.
33. T. Yan, T. He, and J. Stankovic, Differentiated Surveillance for Sensor Networks, *ACM SenSys*, November 2003.
34. G. Zhou, T. He, S. Krishnamurthy, and J. Stankovic, Impact of Radio Asymmetry on Wireless Sensor Networks, *MobiSys*, June 2004.
35. G. Zhou, T. He, J. Stankovic, and T. Abdelzaher, RID: Radio Interference Detection in Wireless Sensor Networks, *Infocom*, 2005.
36. G. Zhou, C. Huang, T. Yan, T. He, and J. Stankovic, MMSN: Multi-Frequency Media Access Control for Wireless Sensor Networks, *Infocom*, April 2006.

21

Messaging in Sensor Networks: Addressing Wireless Communications and Application Diversity

Hongwei Zhang
Wayne State University

Anish Arora
Ohio State University

Prasun Sinha
Ohio State University

Loren J. Rittle
Motorola Labs

21.1	Introduction	21-1
21.2	SMA: An Architecture for Sensornet Messaging	21-2
	Components of Sensornet Messaging • Architecture of Sensornet Messaging	
21.3	Data-Driven Link Estimation and Routing	21-6
	ELD: The Routing Metric • LOF: A Data-Driven Protocol • Experimental Evaluation	
21.4	Related Work	21-17
21.5	Concluding Remarks	21-18

21.1 Introduction

In wireless sensor networks, which we refer to as *sensornets* hereafter, nodes coordinate with one another to perform tasks such as event detection and data collection. Since nodes are spatially distributed, message passing is the basic enabler of node coordination in sensornets. This chapter deals with the sensornet system service that is responsible for message passing; this service includes consideration of routing and transport issues and we refer to it as the *messaging* service. Even though messaging has been studied extensively in existing wired networks such as the Internet, it remains an important problem for sensornets. This is primarily due to the complex dynamics of wireless communication, to resource constraints, and to application diversity, as we explain below.

Wireless communication is subject to the impact of a variety of factors such as fading, multipath, environmental noise, and cochannel interference. As a result, wireless link properties (e.g., packet delivery rate) are dynamic and assume complex spatial and temporal patterns [38,41]. The impact of these complex dynamics on messaging is substantial; for instance, it has led to changes in even the primitive concept of neighborhood [32]. Moreover, the mechanisms that deal with the dynamics are constrained in terms of their energy, network bandwidth, space, and time budget. Broadly speaking, the challenge then has been *how to provide dependable, efficient, and scalable messaging despite the complex dynamics of wireless links*.

Sensornets have a broad range of application, in science (e.g., ecology and seismology), engineering (e.g., industrial control and precision agriculture), and our daily life (e.g., traffic control and healthcare).

The breadth of application domains diversifies sensornet systems in many ways, including their traffic patterns and quality of service (QoS) requirements. For instance, in data-collection systems such as those that observe ecology, application data are usually generated periodically, and the application can tolerate certain degrees of loss and latency in data delivery; but in emergency-detection systems such as those for industrial control, data are generated only when rare and emergent events occur, and the application requires that the data be delivered reliably and in real time. The implications of application diversity for messaging include

- Application traffic affects wireless link properties owing to interference among simultaneous transmissions. This impact can vary significantly across diverse traffic patterns [38].
- Different requirements of QoS and in-network processing pose different constraints on the spatial and temporal flow of application data [39].
- Messaging services that are custom-designed for one application can be unsuitable for another, as is evidenced in a study by Zhang et al. [4].
- It is desirable that message services accommodate diverse QoS and in-network processing requirements.

Broadly speaking, the challenge then has been *how to provide messaging that accommodates and potentially adapts to diverse application traffic patterns and QoS requirements as well as supports diverse in-network processing methods*.

The design of messaging services that bridge the challenges of *both* the wireless communications and the application diversity deserves further study and is the focus of this chapter. We particularly emphasize how messaging can be made aware of, or exploit, or adapt to the application characteristics. Specifically, we present an architecture and examine some algorithmic design issues for sensornet messaging in this context.

Our architecture, sensornet messaging architecture (SMA), identifies a messaging component, traffic-adaptive link estimation and routing (TLR), that deals with wireless link dynamics and its interaction with application traffic patterns; it also identifies two other messaging components, application-adaptive structuring (AST) and application-adaptive scheduling (ASC), that support diversified application requirements (such as QoS and in-network processing). After discussing SMA, we present in detail one instantiation of the TLR component, the learn on the fly (LOF) routing protocol, in Section 21.3. LOF deals with link dynamics and its interaction with application traffic patterns in forming the basic routing structure. We discuss related work in Section 21.4, and we make concluding remarks in Section 21.5.

21.2 SMA: An Architecture for Sensornet Messaging

To support diverse applications in a scalable manner, it is desirable to have a unified messaging architecture that identifies the common components as well as their interactions [39]. To this end, we first review the basic functions of sensornet messaging, based upon which we then identify the common messaging components and design the messaging architecture SMA.

21.2.1 Components of Sensornet Messaging

As in the case for the Internet, the objective of messaging in sensornets is to deliver data from their sources to their destinations. To this end, the basic tasks of messaging are, given certain QoS constraints (e.g., reliability and latency) on data delivery, choose the route(s) from every source to the corresponding destination(s) and schedule packet flow along the route(s). As we argued before, unlike wired networks, the design of messaging in sensornets is challenging as a result of wireless communication dynamics, resource constraints, and application diversity.

Given the complex dynamics of sensornet wireless links, a key component of sensornet messaging is precisely estimating wireless link properties and then finding routes of high-quality links to deliver data traffic. Given that data traffic pattern affects wireless link properties owing to interference among

simultaneous transmissions [38], link estimation and routing should be able to take into account the impact of application data traffic, and we call this basic messaging component TLR.

With the basic communication structure provided by the TLR component, another important task of messaging is to adapt the structure and data transmission schedules according to application properties such as in-network processing and QoS requirements. Given the resource constraints in sensornets, application data may be processed in the network before it reaches the final destination to improve resource utilization (e.g., to save energy and to reduce data traffic load). For instance, data arriving from different sources may be compressed at an intermediate node before it is forwarded further. Given that messaging determines the spatial and temporal flow of application data and that data items from different sources can be processed together only if they meet somewhere in the network, messaging significantly affects the degree of processing achievable in the network [16,39]. It is therefore desirable that messaging considers in-network processing when deciding how to form the messaging structure and how to schedule data transmissions. In addition, messaging should also consider application QoS requirements (e.g., reliability and latency in packet delivery), because messaging structure and transmission schedule determine the QoS experienced by application traffic [20,22,30]. In-network processing and QoS requirements tend to be tightly coupled with applications, thus we call the structuring and scheduling in messaging AST and ASC, respectively.

21.2.2 Architecture of Sensornet Messaging

These messaging components are coupled with wireless communication and applications in different ways and in different degrees, so we adopt two levels of abstraction in designing the architecture for sensornet messaging. The architecture, SMA, is shown in Figure 21.1. At the lower level, TLR interacts directly with the link layer to estimate link properties and to form the basic routing structure in a traffic-adaptive manner. TLR can be performed without explicit input from applications, and TLR does not directly interface with applications. At the higher level, both AST and ASC need input from applications, thus AST and ASC interface directly with applications. Besides interacting with TLR, AST and ASC may need to directly interact with link layer to perform tasks such as adjusting radio transmission power level and fetching link-layer acknowledgment to a packet transmission. In the architecture, the link and physical layers support higher-layer messaging tasks (i.e., TLR, AST, and ASC) by providing the capability of communication within one-hop neighborhoods.

In what follows, we elaborate on the individual components of SMA.

21.2.2.1 Traffic-Adaptive Link Estimation and Routing

To estimate wireless link properties, one approach is to use beacon packets as the basis of link estimation. That is, neighbors exchange broadcast beacons, and they estimate broadcast link properties based on the quality of receiving one another's beacons (e.g., the ratio of beacons successfully received, or the receiver signal strength indicator/link quality indicator (RSSI/LQI) of packet reception); then, neighbors estimate unicast link properties based on those of beacon broadcast, since data are usually transmitted

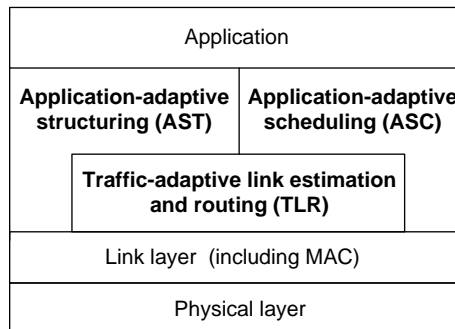


FIGURE 21.1 SMA: a sensornet messaging architecture.

via unicast. This approach of beacon-based link estimation has been used in several routing protocols including expected transmission count (ETX) [12,35].

We find that there are two major drawbacks of beacon-based link estimation. First, it is hard to build high-fidelity models for temporal correlations in link properties [23,31,33], thus most existing routing protocols do not consider temporal link properties and instead assume independent bit error or packet loss. Consequently, significant estimation error can be incurred, as we show in Ref. 38. Second, even if we could precisely estimate unicast link properties, the estimated values may only reflect unicast properties in the absence—instead of the presence—of data traffic, which matters since the network traffic affects link properties owing to interference. This is especially the case in event-detection applications, where events are usually rare (e.g., one event per day) and tend to last only for a short time at each network location (e.g., less than 20 s). Therefore, beacon-based link estimation cannot precisely estimate link properties in a traffic-adaptive manner.

To address the limitations of beacon-based link estimation, Zhang et al. [38] proposed the LOF routing protocol that estimates unicast link properties via MAC feedback* for data transmissions themselves without using beacons. Since MAC feedback reflects *in situ* the network condition in the presence of application traffic, link estimation in LOF is traffic adaptive. LOF also addresses the challenges of data-driven link estimation to routing protocol design, such as uneven link sampling (i.e., the quality of a link is not sampled unless the link is used in data forwarding). It has been shown that, compared with beacon-based link estimation and routing, LOF improves both the reliability and the energy efficiency in data delivery. More importantly, LOF quickly adapts to changing traffic patterns, and this is achieved without any explicit input from applications.

The TLR component provides the basic service of automatically adapting link estimation and routing structure to application traffic patterns. TLR also exposes its knowledge of link and route properties (such as end-to-end packet delivery latency) to higher-level components AST and ASC, so that AST and ASC can optimize the degree of in-network processing while providing the required QoS in delivering individual pieces of application data.

21.2.2.2 Application-Adaptive Structuring

One example of AST is to adjust messaging structure according to application QoS requirements. For instance, radio transmission power level determines the communication range of each node and the connectivity of a network. Accordingly, transmission power level affects the number of routing hops between any pairs of source and destination and thus packet delivery latency. Transmission power level also determines the interference range of packet transmissions, and thus it affects packet delivery reliability. Therefore, radio transmission power level (and thus messaging structure) can be adapted to satisfy specific application QoS requirements, and Kawadia and Kumar have studied this in Ref. 22.

Besides QoS-oriented structuring, another example of AST is to adjust messaging structure according to the opportunities of in-network processing. Messaging structure determines how data flows spatially, and thus affects the degree of in-network processing achievable. For instance, as shown in Figure 21.2a,

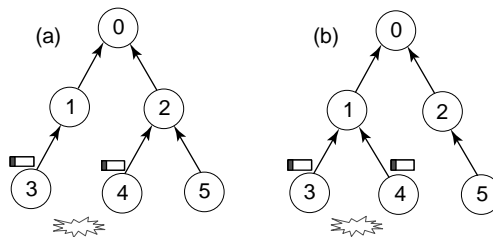


FIGURE 21.2 Example of application-adaptive structuring: (a) before adaptation and (b) after adaptation.

*The MAC feedback for a unicast transmission includes whether the transmission has succeeded and how many times the packet has been retransmitted at the MAC layer.

nodes 3 and 4 detect the same event simultaneously. But the detection packets generated by nodes 3 and 4 cannot be aggregated in the network, since they follow different routes to the destination node 0. However, if node 4 can detect the correlation between its own packet and that generated by node 3, node 4 can change its next-hop forwarder to node 1, as shown in Figure 21.2b. Then the packets generated by nodes 3 and 4 can meet at node 1, and be aggregated before being forwarded to the destination node 0.

In general, to improve the degree of in-network processing, a node should consider the potential in-network processing achievable when choosing the next-hop forwarder. One way to realize this objective is to adapt the existing routing metric. For each neighbor k , a node j estimates the utility $u_{j,k}$ of forwarding packets to k , where the utility is defined as the reduction in messaging cost (e.g., number of transmissions) if j 's packets are aggregated with k 's packets. Then, if the cost of messaging via k without aggregation is $c_{j,k}$, the associated messaging cost $c'_{j,k}$ can be adjusted as follows (to reflect the utility of in-network processing):

$$c'_{j,k} = c_{j,k} - u_{j,k}$$

Accordingly, a neighbor with the lowest adjusted messaging cost is selected as the next-hop forwarder.

Since QoS requirements and in-network processing vary from one application to another, AST needs input (e.g., QoS specification and utility of in-network processing) from applications, and it needs to interface with applications directly.

21.2.2.3 Application-Adaptive Scheduling

One example of ASC is to schedule packet transmissions to satisfy certain application QoS requirements. To improve packet delivery reliability, for instance, lost packets can be retransmitted. But packet retransmission consumes energy, and not every sensor network application needs 100% packet delivery rate. Therefore, the number of retransmissions can be adapted to provide different end-to-end packet delivery rates while minimizing the total number of packet transmissions [8]. To provide differentiated timeliness guarantee on packet delivery latency, we can also introduce priority in transmission scheduling such that urgent packets have high priority of being transmitted [36]. Similarly, data streams from different applications can be ranked so that transmission scheduling ensures differentiated end-to-end throughput to different applications [15].

Besides QoS-oriented scheduling, another example of ASC is to schedule packet transmissions according to the opportunities of in-network processing. Given a messaging structure formation, transmission scheduling determines how data flows along the structure temporally and thus the degree of in-network processing achievable. To give an example, let us look at Figure 21.3a. Suppose node 4 detects an event earlier than node 3 does. Then the detection packet from node 4 can reach node 1 earlier than the packet from node 3. If node 1 immediately forward the packet from node 4 after receiving it, then the packet from node 4 cannot be aggregated with that from node 3, since the packet from node 4 has already left node 1 when the packet from node 3 reaches node 1. In contrast, if node 1 is aware of the correlation between packets from nodes 3 and 4, then node 1 can hold the packet from 4 after receiving it (as shown in Figure 21.3b). Accordingly, the packet from node 3 can meet that from node 4, and these packets can be aggregated before being forwarded.

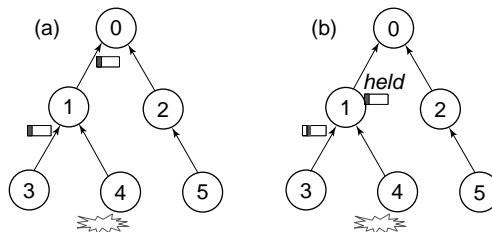


FIGURE 21.3 Example of application-adaptive scheduling: (a) before adaptation and (b) after adaptation.

In general, a node should consider both application QoS requirements and the potential in-network processing when scheduling data transmissions, so that application QoS requirements are better satisfied and the degree of in-network processing is improved. Given that in-network processing and QoS requirements are application specific, ASC needs to directly interface with applications to fetch input on parameters such as QoS specification and utility of in-network processing.

Owing to the limitation of space, we only discuss TLR in detail in the remainder of this chapter. A detailed study of ASC can be found in Ref. 39; AST is still an open issue for further exploration.

Remark

It is desirable that the components TLR, AST, and ASC be deployed all together to achieve the maximal network performance. That said, the three components can also be deployed in an incremental manner while maintaining the benefits of each individual component, as shown in Refs. 38 and 39.

21.3 Data-Driven Link Estimation and Routing

As briefly discussed in Section 21.2, the major drawbacks of beacon-based link estimation and routing are twofold: First, it is hard to precisely estimate unicast link properties via those of broadcast, especially when temporal correlation in link properties is not considered. Second, network condition changes with traffic patterns, and beacon-based link estimation may not be able to converge. To get an intuitive sense of how traffic patterns affect network condition and how temporal correlation affects the fidelity of beacon-based link estimation, we experimentally measure the packet delivery rate of broadcast and unicast in the presence of different traffic patterns, and we calculate the error incurred in link estimation if temporal correlations in link properties are not considered. We conduct the experiments in the sensornet testbed Kansei [7]. Kansei is deployed in an open warehouse with flat aluminum walls (see Figure 21.4a), and it consists of 195 Stargates* organized into a 15×13 grid (as shown in Figure 21.4b) where the separation between neighboring grid points is 0.91 m (i.e., 3 ft). Each Stargate is equipped with the same SMC wireless card as in the outdoor testbed. To create realistic multihop wireless networks similar to the outdoor testbed, each Stargate is equipped with a 2.2 dBi rubber duck omnidirectional antenna and a 20 dB attenuator. We raise the Stargates 1.01 m above the ground by putting them on wood racks. The transmission power level of each Stargate is set as 60, to simulate the low-to-medium density multihop networks where a node can reliably communicate with around 15 neighbors. Figure 21.5 shows that network condition, measured in broadcast reliability, varies significantly with traffic patterns. Figure 21.6 shows the significant error[†] in estimating unicast delivery rate via that of broadcast under different traffic scenarios when temporal correlations in link properties are not considered (i.e., assuming independent bit error and packet loss) [38]. Therefore, it is not trivial, if even possible, to precisely estimate link properties for unicast data via those of broadcast beacons.

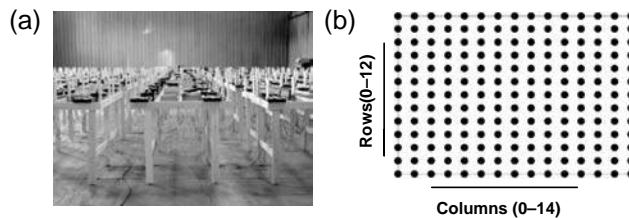


FIGURE 21.4 Sensornet testbed Kansei: (a) Kansei and (b) grid topology.

*Stargates [2] use IEEE 802.11 as its radio, and have been adopted as the backbone nodes in many sensornet systems including ExScal [9] and MASE [1].

[†]The error is defined as actual unicast link reliability minus the estimated link reliability.

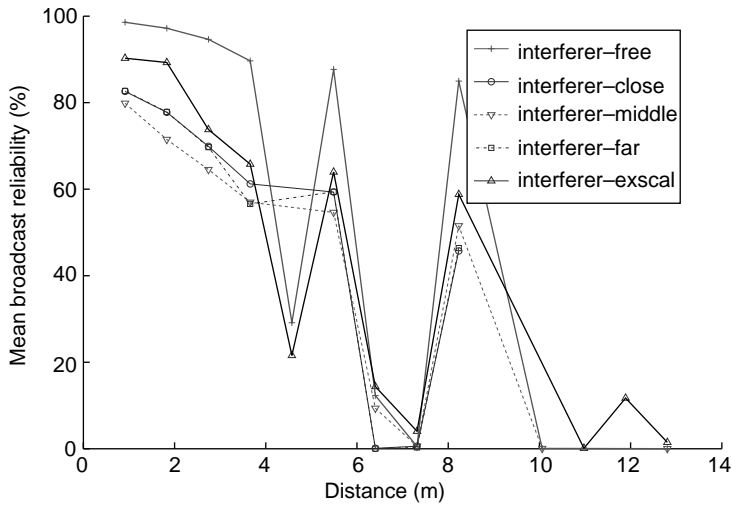


FIGURE 21.5 Network condition, measured in broadcast reliability, in the presence of different traffic patterns.

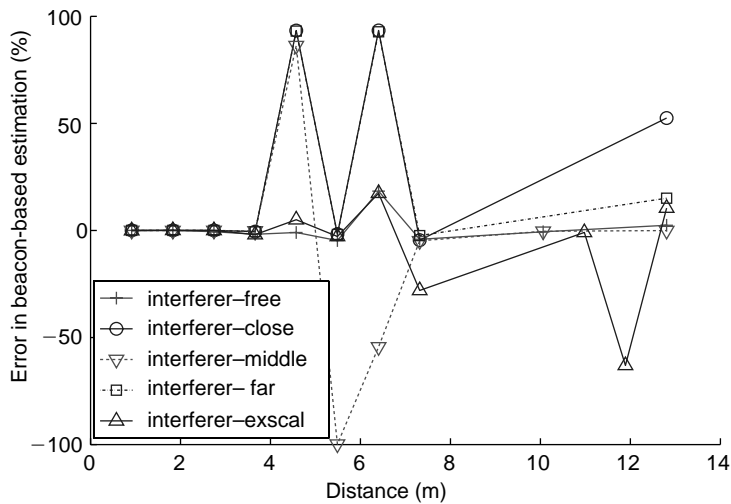


FIGURE 21.6 Error in estimating unicast delivery rate via that of broadcast.

To circumvent the difficulty of estimating unicast link properties via those of broadcast, we propose to directly estimate unicast link properties via data traffic itself. In this context, since we are not using beacons for link property estimation, we also explore the idea of not using periodic beacons in routing at all (i.e., beacon-free routing) to save energy; otherwise, beaconing requires nodes to wake up periodically even when there is no data traffic.

To enable data-driven routing, we need to find alternative mechanisms for accomplishing the tasks that are traditionally assumed by beacons: acting as the basis for link property estimation, and diffusing information (e.g., the cumulative ETX metric). In sensornet backbones, data-driven routing is feasible because of the following facts:

- *MAC feedback.* In MACs, where every packet transmission is acknowledged by the receiver (e.g., in 802.11b and 802.15.4MACs), the sender can determine if a transmission has succeeded by checking

whether it receives the acknowledgment. Also, the sender can determine how long each transmission takes, i.e., MAC latency. Therefore, the sender is able to get information on link properties without using any beacons. (Note: it has also been shown that MAC latency is a good routing metric for optimizing wireless network throughput [10].)

- *Mostly static network and geography.* Nodes are static most of the time, and their geographic locations are readily available via devices such as global positioning systems (GPS). Therefore, we can use geography-based routing in which a node only needs to know the location of the destination and the information regarding its local neighborhood (such as, the quality of the links to its neighbors). Thus, only the location of the destination (e.g., the base station in convergecast) needs to be diffused across the network. Unlike in beacon-based distance-vector routing, the diffusion happens infrequently since the destination is static most of the time. In general, control packets are needed only when the location of a node changes, which occurs infrequently.

In what follows, we first present the routing metric expected MAC latency per unit distance to destination (ELD), which is based on geography and MAC latency, then we present the design and the performance of LOF, which implements ELD without using periodic beacons.

Remarks

- Although parameters such as RSSI, LQI, and signal-to-noise ratio (SNR) also reflect link reliability, it is difficult to use them as a precise prediction tool [6]. Moreover, the aforementioned parameters can be fetched only at packet receivers (instead of senders), and extra control packets are needed to convey these information back to the senders if we want to use them as the basis of link property estimation. Therefore, we do not recommend using these parameters as the core basis of data-driven routing, especially when senders need to precisely estimate *in situ* link properties.
- Routing metric can also be based on other parameters such as ETX [12] or required number of packets (RNP) [11], and detailed study has been reported in Ref. 40. Owing to the limitation of space, however, we only present the routing metric that is based on MAC latency in this chapter.

21.3.1 ELD: The Routing Metric

For messaging in sensor networks (especially for event-driven applications), packets need to be routed reliably and in real-time to the base station. As usual, packets should also be delivered in an energy-efficient manner. Therefore, a routing metric should reflect link reliability, packet delivery latency, and energy consumption at the same time. One such metric that we adopt in LOF is based on MAC latency, i.e., the time taken for the MAC to transmit a data frame. (We have mathematically analyzed the relationship among MAC latency, energy consumption, and link reliability, and we find that MAC latency is strongly related to energy consumption in a positive manner, and the ratio between them changes only slightly as link reliability changes. Thus, routing metrics optimizing MAC latency would also optimize energy efficiency. Interested readers can find the detailed analysis in Ref. 37.)

Given that MAC latency is a good basis for route selection and that geography enables low-frequency information diffusion, we define a routing metric ELD, *the expected MAC latency-per unit-distance to the destination*, which is based on both MAC latency and geography. Specifically, given a sender S , a neighbor R of S , and the destination D as shown in Figure 21.7, we first calculate the *effective geographic progress* from S to D via R , denoted by $L_e(S, R)$, as $(L_{S,D} - L_{R,D})$, where $L_{S,D}$ denotes the distance

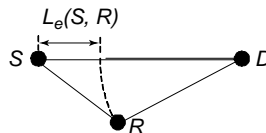


FIGURE 21.7 L_e calculation.

between S and D , and $L_{R,D}$ denotes the distance between R and D . Then, we calculate, for the sender S , the MAC latency per unit-distance to the destination (LD) via R , denoted by $LD(S, R)$, as*

$$\begin{cases} \frac{D_{S,R}}{L_e(S,R)} & \text{if } L_{S,D} > L_{R,D} \\ \infty & \text{otherwise} \end{cases} \quad (21.1)$$

where $D_{S,R}$ is the MAC latency from S to R . Therefore, the ELD via R , denoted as $ELD(S, R)$, is $E(LD(S, R))$, which is calculated as

$$\begin{cases} \frac{E(D_{S,R})}{L_e(S,R)} & \text{if } L_{S,D} > L_{R,D} \\ \infty & \text{otherwise} \end{cases} \quad (21.2)$$

For every neighbor R of S , S associates with R a rank

$$\langle ELD(S, R), \text{var}(LD(S, R)), L_{R,D}, ID(R) \rangle$$

where $\text{var}(LD(S, R))$ denotes the variance of $LD(S, R)$, and $ID(R)$ denotes the unique ID of node R . Then, S selects as its next-hop forwarder the neighbor that ranks the lowest among all the neighbors. (Note: routing via metric ELD is a greedy approach, where each node tries to optimize the local objective. Like many other greedy algorithms, this method is effective in practice, as shown via experiments in Section 21.3.3.)

To understand what ELD implies in practice, we set up an experiment as follows: consider a line network formed by row 6 of the indoor testbed shown in Figure 21.4, the Stargate S at column 0 needs to send packets to the Stargate D at the other end (i.e., column 14). Using the data on unicast MAC latencies in the case of *interferer-free*, we show in Figure 21.8 the mean unicast MAC latencies and the corresponding ELDs regarding neighbors at different distances. From the figure, Stargate D , the destination, which is 12.8 m away from S , offers the lowest ELD, and S sends packets directly to D . From this example, we see that, using metric ELD, a node tends to choose nodes beyond the reliable communication range as forwarders, to reduce end-to-end MAC latency as well as energy consumption.

Remark

ELD is a locally measurable metric based only on the geographic locations of nodes and information regarding the links associated with the sender S ; ELD does not assume link conditions beyond the local neighborhood of S . In the analysis of geographic routing [29], however, a common assumption

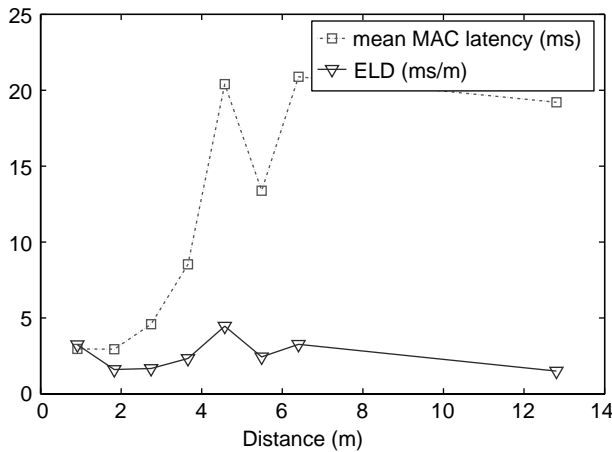


FIGURE 21.8 Mean unicast MAC latency and the ELD.

*Currently, we focus on the case where a node forward packets only to a neighbor closer to the destination than itself.

is *geographic uniformity*—that the hops in any route have similar properties such as geographic length and link quality. As we will show by experiments in Section 21.3.3, this assumption is usually invalid. For the sake of verification and comparison, we derive another routing metric ELR, the *expected MAC latency along a route*, based on this assumption. More specifically,

$$\text{ELR}(S, R) = \begin{cases} E(D_{S,R}) \times \left\lceil \frac{L_{S,R} + L_{R,D}}{L_{S,R}} \right\rceil & \text{if } L_{S,D} > L_{R,D} \\ \infty & \text{otherwise} \end{cases} \quad (21.3)$$

where $\left\lceil \frac{L_{S,R} + L_{R,D}}{L_{S,R}} \right\rceil$ denotes the number of hops to the destination, assuming equal geographic distance at every hop. We will show in Section 21.3.3 that ELR is inferior to ELD.

21.3.2 LOF: A Data-Driven Protocol

Having determined the routing metric ELD, we are ready to design protocol LOF for implementing ELD without using periodic beacons. Without loss of generality, we only consider a single destination, i.e., the base station to which every other node needs to find a route.

Briefly speaking, LOF needs to accomplish two tasks: First, to enable a node to obtain the geographic location of the base station, as well as the IDs and locations of its neighbors; Second, to enable a node to track the LD (i.e., MAC latency per unit-distance to the destination) regarding each of its neighbors. The first task is relatively simple and only requires exchanging a few control packets among neighbors in rare cases (e.g., when a node boots up); LOF accomplishes the second task using three mechanisms: initial sampling of MAC latency, adapting estimation via MAC feedback for application traffic, and probabilistically switching next-hop forwarder.

21.3.2.1 Learning Where We Are

LOF enables a node to learn its neighborhood and the location of the base station via the following rules:

- I. [**Issue request**] Upon boot-up, a node broadcasts M copies of *hello-request* packets if it is not the base station. A *hello-request* packet contains the ID and the geographic location of the issuing node. To guarantee that a requesting node is heard by its neighbors, we set M as 7 in our experiments.
- II. [**Answer request**] When receiving a *hello-request* packet from another node that is farther away from the base station, the base station or a node that has a path to the base station acknowledges the requesting node by broadcasting M copies of *hello-reply* packets. A *hello-reply* packet contains the location of the base station as well as the ID and the location of the issuing node.
- III. [**Handle announcement**] When a node A hears for the first time a *hello-reply* packet from another node B closer to the base station, A records the ID and location of B and regards B as a forwarder-candidate.
- IV. [**Announce presence**] When a node other than the base station finds a forwarder-candidate for the first time, or when the base station boots up, it broadcasts M copies of *hello-reply* packets.

To reduce potential contention, every broadcast transmission mentioned above is preceded by a randomized waiting period whose length is dependent on node distribution density in the network. Note that the above rules can be optimized in various ways. For instance, rule II can be optimized such that a node acknowledges at most one *hello-request* from another node each time the requesting node boots up. Even though we have implemented quite a few such optimizations, we skip the detailed discussion here since they are not the focus of this chapter.

21.3.2.2 Initial Sampling

Having learned the location of the base station as well as the locations and IDs of its neighbors, a node needs to estimate the LDs regarding its neighbors. To design the estimation mechanism, let us first check Figure 21.9, which shows the mean unicast MAC latency in different interfering scenarios. We see that, even though MAC latencies change as interference pattern changes, the relative ranking in the mean MAC latency among links does not change much. Neither will the LDs accordingly.

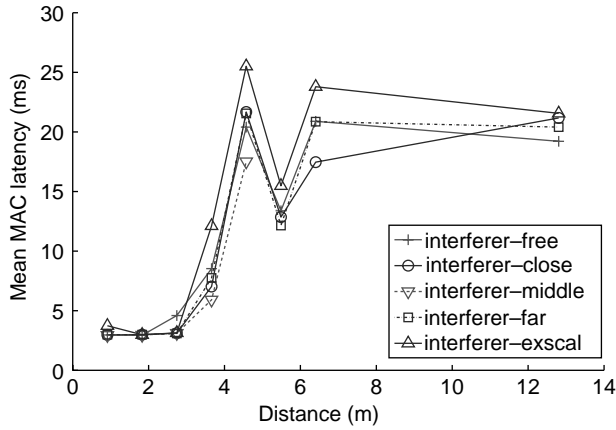
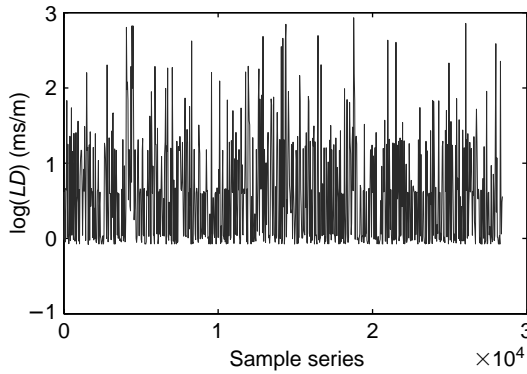


FIGURE 21.9 MAC latency in the presence of interference.


 FIGURE 21.10 A time series of $\log(LD)$.

In LOF, therefore, when a node S learns of the existence of a neighbor R for the first time, S takes a few samples of the MAC latency for the link to R before forwarding any data packets to R . The sampling is achieved by S sending a few unicast packets to R and then fetching the MAC feedback. The initial sampling gives a node a rough idea of the relative quality of the links to its neighbors, to jump start the data-driven estimation.

21.3.2.3 Data-Driven Adaptation

Via initial sampling, a node gets a rough estimation of the relative goodness of its neighbors. To improve its route selection for an application traffic pattern, the node needs to adapt its estimation of LD via the MAC feedback for unicast data transmission. Since LD is lognormally distributed, LD is estimated by estimating $\log(LD)$.

Online estimation. To determine the estimation method, we first check the properties of the time series of $\log(LD)$. Figure 21.10 shows a time series of the $\log(LD)$ regarding a node 3.65 m (i.e., 12 ft) away from a sender. We see that the time series fits well with the *constant-level model* [19], where the generating process is represented by a constant superimposed with random fluctuations. Therefore, a good estimation method is *exponentially weighted moving average* (EWMA) [19], assuming the following form:

$$V \leftarrow \alpha V + (1 - \alpha)V' \quad (21.4)$$

where V is the parameter to be estimated, V' the latest observation of V , and α the weight ($0 \leq \alpha \leq 1$).

In LOF, when a new MAC latency and thus a new $\log(LD)$ value with respect to the current next-hop forwarder R is observed, the V value in the right-hand side of Formula 21.4 may be quite old if R has just been selected as the next-hop and some packets have been transmitted to other neighbors immediately before. To deal with this issue, we define the *age factor* $\beta(R)$ of the current next-hop forwarder R as the number of packets that have been transmitted since V of R was last updated. Then, Formula 21.4 is adapted to be the following:

$$V \leftarrow \alpha^{\beta(R)} V + (1 - \alpha^{\beta(R)}) V' \quad (21.5)$$

(Experiments confirm that LOF performs better with Formula 21.5 than with Formula 21.4.)

Each MAC feedback indicates whether a unicast transmission has succeeded and how long the MAC latency l is. When a node receives a MAC feedback, it first calculates the age factor $\beta(R)$ for the current next-hop forwarder, then it adapts the estimation of $\log(LD)$ as follows:

- If the transmission has succeeded, the node calculates the new $\log(LD)$ value using l and applies it to Formula 21.5 to get a new estimation regarding the current next-hop forwarder.
- If the transmission has failed, the node should not use l directly because it does not represent the latency to successfully transmit a packet. To address this issue, the node keeps track of the unicast delivery rate, which is also estimated using Formula 21.5, for each associated link. Then, if the node retransmits this unicast packet via the currently used link, the expected number of retries until success is $\frac{1}{p}$, assuming that unicast failures are independent and that the unicast delivery rate along the link is p . Including the latency for this last failed transmission, the expected overall latency l' is $(1 + \frac{1}{p})l$. Therefore, the node calculates the new $\log(LD)$ value using l' and applies it to Formula 21.5 to get a new estimation.

Another important issue in EWMA estimation is choosing the weight α , since it affects the stability and agility of estimation. To address this question, we try out different α values and compute the corresponding estimation fidelity, that is, the probability of LOF choosing the right next-hop forwarder for S . Figure 21.11a shows the best α value and the corresponding estimation fidelity for different windows of comparison. If the window of comparison is 20 s, for instance, the best α is 0.8, and the corresponding estimation fidelity is 89.3%. (Since the time span of the ExScal traffic trace is about 20 s, we set α as 0.8 in our experiments.)

For sensitivity analysis, Figure 21.11b shows how the estimation fidelity changes with α when the window of comparison is 20 s. We see that the estimation fidelity is not very sensitive to changes in α over a wide range. For example, the estimation fidelity remains above 85% when α changes from 0.6 to 0.98. Similar patterns are observed for the other windows of comparison too. The insensitivity of estimation fidelity to α guarantees the robustness of EWMA estimation in different environments.

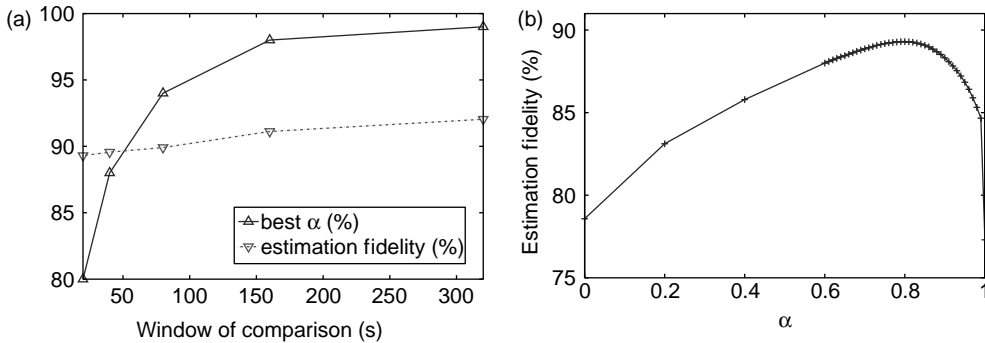


FIGURE 21.11 The weight α in EWMA: (a) best α and (b) sensitivity analysis.

Route adaptation. As the estimation of LD changes, a node S adapts its route selection by the ELD metric. Moreover, if the unicast reliability to a neighbor R is below certain threshold (say 60%), S will mark R as dead and will remove R from the set of forwarder-candidates. If S loses all its forwarder-candidates, S will first broadcast M copies of *hello-withdrawal* packets and then restarts the routing process. If a node S' hears a *hello-withdrawal* packet from S , and if S is a forwarder-candidate of S' , S' removes S from its set of forwarder-candidates and updates its next-hop forwarder as need be. (As a side note, we find that, on average, only 0.9863 neighbors of any node are marked as dead in both our testbed experiments and the field deployment of LOF in project ExScal [9]. Again, the withdrawing and rejoining process can be optimized, but we skip the details here.)

21.3.2.4 Probabilistic Neighbor Switching

Given that the initial sampling is not perfect (e.g., covering 80% instead of 100% of all the possible cases) and that wireless link quality varies temporally, the data-driven adaptation alone may miss using good links, simply because they were relatively bad when tested earlier and they do not get a chance to be tried out later on. Therefore, we propose probabilistic neighbor switching in LOF. That is, whenever a node S has consecutively transmitted $I_{ns}(R_0)$ number of data packets using a neighbor R_0 , S will switch its next-hop forwarder from R_0 to another neighbor R' with probability $P_{ns}(R')$. In contrast, the probabilistic neighbor switching is exploratory and optimistic in nature, therefore, it should be used only for good neighbors. In LOF, neighbor switching only considers the set of neighbors that are not marked as dead.

In what follows, we explain how to determine the switching probability $P_{ns}(R')$ and the switching interval $I_{ns}(R_0)$. For convenience, we consider a sender S , and let the neighbors of S be R_0, R_1, \dots, R_N with increasing ranks.

Switching probability. At the moment of neighbor switching, a better neighbor should be chosen with higher probability. In LOF, a neighbor is chosen with the probability of the neighbor actually being the best next-hop forwarder. We derive this probability in three steps: the probability $P_b(R_i, R_j)$ of a neighbor R_i being actually better than another one R_j , the probability $P_h(R_i)$ of a neighbor R_i being actually better than all the neighbors that ranks lower than itself, and the probability $P_{ns}(R_i)$ of a neighbor R_i being actually the best forwarder. (Interested readers can find the detailed derivation in Ref. 37.)

Switching interval. The frequency of neighbor switching should depend on how good the current next-hop forwarder R_0 is, i.e., the switching probability $P_{ns}(R_0)$. LOF, we set the switching interval $I_{ns}(R_0)$ to be proportional to $P_{ns}(R_0)$, that is,

$$I_{ns}(R_0) = C \times P_{ns}(R_0) \quad (21.6)$$

where C is a constant being equal to $(N \times K)$ with N being the number of active neighbors that S has, and K being a constant reflecting the degree of temporal variations in link quality. We set K to be 20 in our experiments.

The switching probabilities and the switching interval are recalculated each time the next-hop forwarder is changed.

21.3.3 Experimental Evaluation

Via testbeds and field deployment, we experimentally evaluate the design decisions and the performance of LOF. We first present the experiment design, then we discuss the experimental results.

21.3.3.1 Experiment Design

Network setup. In testbed Kansei as shown in Figure 21.4, we let the Stargate at the left-bottom corner of the grid be the base station, to which the other Stargates need to find routes. Then, we let the Stargate S at the upper-right corner of the grid be the traffic source. S sends packets of length 1200 bytes according to the ExScal event trace [38]. For each protocol we study, S simulates 50 event runs, with the interval between consecutive runs being 20 s. Therefore, for each protocol studied, 950 (i.e., 50×19) packets are generated at S .

We have also tested scenarios where multiple senders generate ExScal traffic simultaneously as well as scenarios where the data traffic is periodic; LOF has also been used in the backbone network of ExScal. Interested readers can find the detailed discussion in Ref. 37.

Protocols studied. We study the performance of LOF in comparison with that of beacon-based routing, where the latest development is represented by ETX [12,35] and product of packet reception rate and distance towards destination (PRD) [29]: (For convenience, we do not differentiate the name of a routing metric and the protocol implementing it.)

- *ETX*: expected transmission count. It is a type of geography-unaware distance-vector routing where a node adopts a route with the minimum ETX value. Since the transmission rate is fixed in our experiments, ETX routing also represents another metric ETT [14], where a route with the minimum *expected transmission time* is used. ETT is similar to *MAC latency* as used in LOF.
- *PRD*: product of packet reception rate and distance traversed to the destination. Unlike ETX, PRD is geography based. In PRD, a node selects as its next-hop forwarder the neighbor with the maximum PRD value. The design of PRD is based on the analysis that assumes geographic uniformity.

By their original proposals, ETX and PRD use broadcast beacons in estimating the respective routing metrics. In this chapter, we compare the performance of LOF with that of ETX and PRD as originally proposed in Refs. 12 and 29, without considering the possibility of directly estimating metrics ETX and PRD via data traffic. This is because the firmware of our SMC WLAN cards does not expose information on the number of retries of a unicast transmission. In our experiments, metrics ETX and PRD are estimated according to the method originally proposed in Refs. [12] and [29]; for instance, broadcast beacons have the same packet length and transmission rate as those of data packets.

To verify some important design decisions of LOF, we also study different versions of LOF as follows*:

- *L-hop*: assumes geographic uniformity, and thus uses metric ELR, as specified by Formula 21.3, instead of ELD
- *L-ns*: does not use the method of probabilistic neighbor switching
- *L-sd*: considers, in probabilistic neighbor switching, the neighbors that have been marked as dead
- *L-se*: performs probabilistic neighbor switching after every packet transmission

For easy comparison, we have implemented all the protocols mentioned above in EmStar [3], a software environment for developing and deploying sensor networks.

Evaluation criteria. Reliability is one critical concern in convergecast. Using the techniques of reliable transport discussed in Ref. 37, all the protocols guarantee 100% packet delivery in our experiments. Therefore, we compare protocols in metrics other than reliability as follows:

- *End-to-end MAC latency*: the sum of the MAC latency spent at each hop of a route. This reflects not only the delivery latency but also the throughput available via a protocol [12,14].
- *Energy efficiency*: energy spent in delivering a packet to the base station.

21.3.3.2 Experimental Results

MAC latency. Using boxplots, Figure 21.12 shows the end-to-end MAC latency, in milliseconds, for each protocol. The average end-to-end MAC latency in both ETX and PRD is around three times that in LOF, indicating the advantage of data-driven link quality estimation. The MAC latency in LOF is also less than that of the other versions of LOF, showing the importance of using the right routing metric (including not assuming geographic uniformity) and neighbor switching technique.

To explain the above observation, Figures 21.13 through 21.16 show the route hop length, per-hop MAC latency, average per-hop geographic distance, and the coefficient of variation (COV) of per-hop

*Note: we have studied the performance of geography-unaware distance-vector routing using data-driven estimation trying to minimize the sum of MAC latency along routes, and we found that the performance is similar to that of LOF, except that more control packets are used.

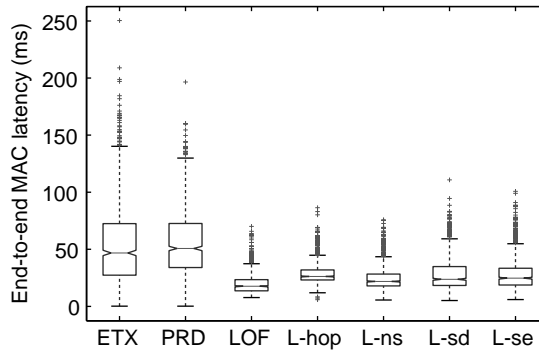


FIGURE 21.12 End-to-end MAC latency.

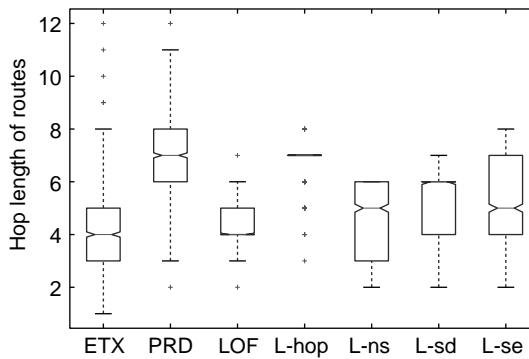


FIGURE 21.13 Number of hops in a route.

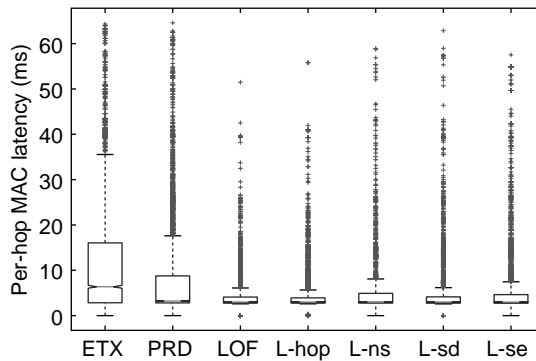


FIGURE 21.14 Per-hop MAC latency.

geographic distance. Even though the average route hop length and per-hop geographic distance in ETX are approximately the same as those in LOF, the average per-hop MAC latency in ETX is about three times that in LOF, which explains why the end-to-end MAC latency in ETX is about three times that in LOF. In PRD, both the average route hop length and the average per-hop MAC latency is about twice that in LOF.

In Figure 21.16, we see that the COV of per-hop geographic distance is as high as 0.4305 in PRD and 0.2754 in L-hop. Therefore, the assumption of geographic uniformity is invalid, which partly explains why PRD and L-hop do not perform as well as LOF. Moreover, the fact that the COV value in LOF is the

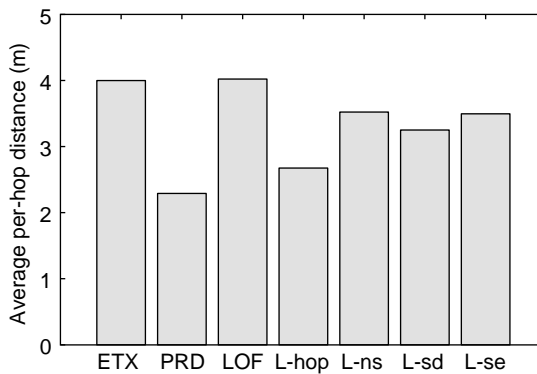


FIGURE 21.15 Average per-hop geographic distance.

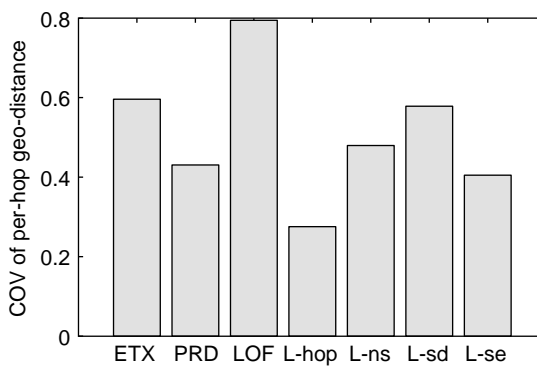


FIGURE 21.16 COV of per-hop geographic distance in a route.

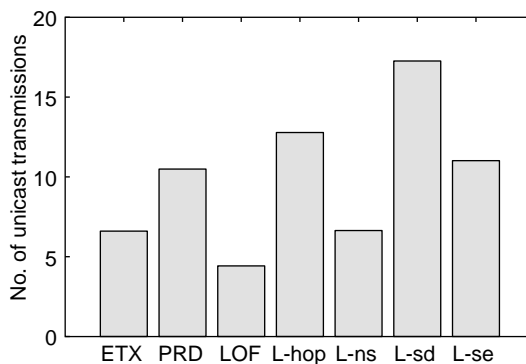


FIGURE 21.17 Number of unicast transmissions per packet received.

largest and that LOF performs the best tend to suggest that the network state is heterogeneous at different locations of the network.

Energy efficiency. Given that beacons are periodically broadcasted in ETX and PRD, and that beacons are rarely used in LOF, it is easy to see that more beacons are broadcasted in ETX and PRD than in LOF. Therefore, we focus our attention only on the number of unicast transmissions required for delivering data packets to the base station rather than on the broadcast overhead. To this end, Figure 21.17 shows

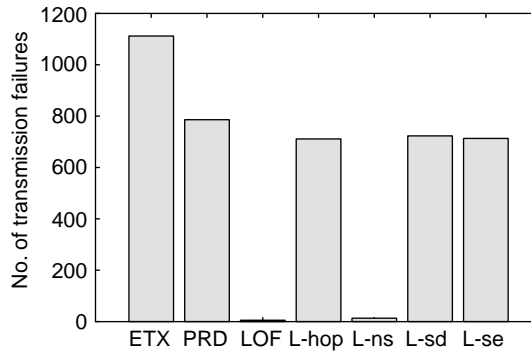


FIGURE 21.18 Number of failed unicast transmissions.

the number of unicast transmissions averaged over the number packets received at the base station. The number of unicast transmissions per packet received in ETX and PRD is 1.49 and 2.37 times that in LOF, respectively, showing again the advantage of data-driven instead of beacon-based link quality estimation. The number of unicast transmissions per packet received in LOF is also less than that in the other versions of LOF. For instance, the number of unicast transmissions in L-hop is 2.89 times that in LOF.

Given that the SMC WLAN card in our testbed uses Intersil Prism2.5 chipset, which does not expose the information on the number of retries of a unicast transmission, Figure 21.17 does not represent the actual number of bytes sent. Nevertheless, given Figure 21.14 and the fact that MAC latency and energy consumption are positively related (as discussed in Section 21.3.1), the above observation on the relative energy efficiency among the protocols still holds.

To explain the above observation, Figure 21.18 shows the number of failed unicast transmissions for the 950 packets generated at the source. The number of failures in ETX and PRD is 1112 and 786, respectively, yet there are only five transmission failures in LOF. Also, there are 711 transmission failures in L-hop. Together with Figure 21.15, we see that there exist reliable long links, yet only LOF tends to find them well: ETX also uses long links, but they are not reliable; L-ns uses reliable links, but they are relatively shorter.

Based on its well-tested performance, LOF has been incorporated in the ExScal sensor network field experiment [9], where 203 Stargates were deployed as the backbone network with the inter-Stargate separation being around 45 m. LOF has successfully guaranteed reliable and real-time convergecast from any number of nonbase Stargates to the base station in ExScal.

21.4 Related Work

Architecture. The sensor network protocol (SP) [28] provides a unified link-layer abstraction for sensor networks. As SP focuses on the interface between link and network layers, SP can be incorporated into the SMA architecture that focuses on higher-layer architecture. Woo et al. [34] discuss network support for query processing in sensor networks, in particular, issues such as query-oriented routing, efficient rendezvous for storage and correlation, and unified in-network processing. Their focus is on query processing, as opposed to the architectural and algorithmic issues involved in supporting a broader range of applications, such as distributed signal processing and computing. Impala [26] considers adapting the communication protocol to changing network conditions and application requirements, using the adaptation finite state machine (AFSM). Nahrstedt et al. [27] consider the provision of application-specific QoS in ubiquitous environments, via a framework for QoS specification and compilation, QoS setup, and QoS adaptation. SMA and LOF complement the proposals in Refs. 26 and 27 by focusing on issues such as application-adaptive link estimation, structuring, and scheduling, which obviate the human from the loop.

In Internet-related research, the concepts of application oriented networking (AON) [5] and application-driven networking [17,21] are being explored to enable coordination among disparate

applications, to enforce application-specific policies, to improve visibility of information flow, and to enhance application optimization and QoS. While these concepts are generic enough to be applied to sensornets, the techniques employed and the problems faced in the Internet context differ from those in sensornets owing to differences in both technology and application domains. For instance, severe resource constraints are unique to sensornets and are not major issues in the Internet.

Routing. Link properties in sensornets and 802.11b mesh networks have been well studied [6,24,41]. These works have observed that wireless links assume complex properties, such as wide-range nonuniform packet delivery rate at different distances, lose correlation between distance and packet delivery rate, link asymmetry, and temporal variations.

In addressing the challenges of wireless communication, great progress has recently been made regarding routing in sensornets and mesh networks. Routing metrics such as ETX [12,35] and ETT/WCETT [14] have been proposed and shown to perform well in real-world wireless networks [13]. The geography-based metric PRD [29] has also been proposed for energy-efficient routing in sensornets. Unicast link properties are estimated using broadcast beacons in these works. LOF differs from existing approaches by avoiding the difficulty of precisely estimating unicast link properties via those of broadcast beacons.

As in LOF, SPEED [18] uses MAC latency and geographic information for route selection. Concurrently with our work, Ref. 25 proposes normalized advance (NADV), which also uses information from MAC layer. While they do focus on real-time packet delivery and a general framework for geographic routing, Refs. 18 and 25 did not focus on the protocol design issues in data-driven link estimation and routing. Nor do they consider the importance of appropriate *probabilistic neighbor switching*. SPEED switches next-hop forwarders after every packet transmission (as in L-se), and NADV does not perform probabilistic neighbor switching (as in L-ns), both of which degenerate network performance, as discussed in Section 21.3.3.

21.5 Concluding Remarks

Being a basic service for sensornets, messaging needs to deal with the dynamics of wireless communication, to support diverse applications, and to cope with the interaction between link dynamics and application traffic patterns. For dependable, efficient, and scalable messaging in sensornets, we have proposed the SMA. SMA employs two levels of abstraction: the lower-level component TLR deals with wireless link dynamics and its interaction with traffic patterns in a unified manner; and the higher-level components AST and ASC directly interface with applications and support application-specific QoS requirements and in-network processing.

SMA is a first step toward the unified SMA. As technologies and applications evolve, it is expected that the architecture and its components will be enriched. The benefits of TLR and ASC have been demonstrated experimentally, yet there are still many fundamental problems associated with modeling the stability and optimality of data-driven link estimation and routing, as well as those of AST and ASC. In the context of link estimation and routing, there are also interesting systems issues deserving further exploration, for instance, comparison of different routing metrics (e.g., number-of-transmission based versus MAC latency based) and metric evaluation mechanisms (e.g., distance-vector routing versus geographic routing). Another important issue in sensornets is power management. It has significant implications to the design of sensornet architecture and algorithms, and should be studied systematically too.

References

1. Broadband seismic network, Mexico experiment (mase). <http://research.cens.ucla.edu/>.
2. Crossbow technology inc. <http://www.xbow.com/>.
3. EmStar: Software for wireless sensor networks. <http://cvs.cens.ucla.edu/emstar/>.
4. Rmase. <http://www2.parc.com/isl/groups/era/nest/Rmase/default.html>.
5. Application-oriented networking. <http://www.cisco.com/>, 2005.

6. D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11b mesh network. *ACM SIGCOMM*, pp. 121–132, 2004.
7. A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal. Kansei: A high-fidelity sensing testbed. *IEEE Internet Computing*, March 2006.
8. A. Arora et al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 46(5): 605–634, 2004.
9. A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridhara, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Gouda, Y. R. Choi, M. Nesterenko, R. Shah, S. Kulkarni, M. Aramugam, L. Wang, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, and K. Parker. Exscal: Elements of an extrem scale wireless sensor network. *IEEE RTCSA*, 2005.
10. B. Awerbuch, D. Holmer, and H. Rubens. High throughput route selection in multi-rate ad hoc wireless networks. Technical Report, Johns Hopkins University, 2003.
11. A. Cerpa, J. Wong, M. Potkonjak, and D. Estrin. Temporal properties of low power wireless links: Modeling and implications on multi-hop routing. *ACM MobiHoc*, pp. 414–425, 2005.
12. D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. *ACM MobiCom*, pp. 134–146, 2003.
13. R. Draves, J. Padhye, and B. Zill. Comparison of routing metrics for static multi-hop wireless networks. *ACM SIGCOMM*, pp. 133–144, 2004.
14. R. Draves, J. Padhye, and B. Zill. Routing in multi-radio, multi-hop wireless mesh networks. *ACM MobiCom*, pp. 114–128, 2004.
15. C. T. Ee and R. Bajcsy. Congestion control and fairness for many-to-one routing in sensor networks. *ACM SenSys*, pp. 148–161, 2004.
16. K.-W. Fan, S. Liu, and P. Sinha. On the potential of structure-free data aggregation in sensor networks. *IEEE INFOCOM*, 2006.
17. J. Follows and D. Straeten. *Application-Driven Networking: Concepts and Architecture for Policy-based Systems*. IBM Corporation, International Technical Support Organization, 1999.
18. T. He, J. Stankovic, C. Lu, and T. Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. *IEEE ICDCS*, 2003.
19. F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill Higher Education, New York, 2001.
20. B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. *ACM SenSys*, pp. 134–147, 2004.
21. IBM. *Application Driven Networking: Class of Service in IP, Ethernet and ATM Networks*. IBM Corporation, International Technical Support Organization, 1999.
22. V. Kawadia and P. R. Kumar. Principles and protocols for power control in ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 23(5):76–88, 2005.
23. A. Konrad, B. Zhao, and A. Joseph. A markov-based channel model algorithm for wireless networks. *Wireless Networks*, 9:189–199, 2003.
24. D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dartmouth College, Computer Science, July 2003.
25. S. Lee, B. Bhattacharjee, and S. Banerjee. Efficient geographic routing in multihop wireless networks. *ACM MobiHoc*, pp. 230–241, 2005.
26. T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. *ACM PPoPP*, 2003.
27. K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. Qos-aware middleware for ubiquitous and heterogeneous environments. *IEEE Communications Magazine*, 2001.
28. J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. *ACM SenSys*, pp. 76–89, 2005.
29. K. Seada, M. Zuniga, A. Helmy, and B. Krishnamacari. Energy-efficient forwarding strategies for geographic routing in lossy wireless sensor networks. *ACM SenSys*, 2004.

30. C. Wan, S. Eisenman, and A. Campbell. CODA: Congestion detection and avoidance in sensor networks. *ACM SenSys*, pp. 266–279, 2003.
31. H. S. Wang and N. Moayeri. Finite-state markov channel—a useful model for radio communication channels. *IEEE Transactions on Vehicular Technology*, 44(1):163–171, 1995.
32. K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. *USENIX/ACM MobiSys*, 2004.
33. A. Willig. A new class of packet- and bit-level models for wireless channels. *IEEE PIMRC*, 2002.
34. A. Woo, S. Madden, and R. Govindan. Networking support for query processing in sensor networks. *Communications of the ACM*, 47(6):47–52, 2004.
35. A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. *ACM SENSYS*, pp. 14–27, 2003.
36. H. Zhang, A. Arora, Y. R. Choi, and M. Gouda. Reliable bursty convergecast in wireless sensor networks. *ACM MobiHoc*, 2005.
37. H. Zhang, A. Arora, and P. Sinha. Learn on the fly: Quiescent routing in sensor network backbones. Technical Report OSU-CISRC-7/05-TR48, The Ohio State University (<http://www.cse.ohio-state.edu/~zhangho/publications/LOF-TR.pdf>), 2005.
38. H. Zhang, A. Arora, and P. Sinha. Learn on the fly: Data-driven link estimation and routing in sensor network backbones. *IEEE INFOCOM*, 2006.
39. H. Zhang, L. J. Rittle, and A. Arora. Application-adaptive messaging in sensor networks. Technical Report OSU-CISRC-6/06-TR63, The Ohio State University, 2006.
40. H. Zhang, L. Sang, and A. Arora. Link estimation and routing in sensor networks: Beacon-based or data-driven? Technical Report OSU-CISRC-6/06-TR64, The Ohio State University, 2006.
41. J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. *ACM SenSys*, pp. 1–13, 2003.

22

Real-Time Communication for Embedded Wireless Networks

Marco Caccamo

University of Illinois at
Urbana-Champaign

Tarek Abdelzaher

University of Illinois at
Urbana-Champaign

22.1	Introduction	22-1
22.2	Basic Concepts for Predictable Wireless Communication	22-2
22.3	Robust and Implicit Earliest Deadline First Terminology and Assumptions • The RI-EDF Protocol • Power-Aware RI-EDF • Dynamic Schedule Updates	22-3
22.4	Higher-Level Real-Time Protocols for Sensor Networks RAP: Real-Time Distance-Aware Scheduling • SPEED: Enforcement of Velocity Constraints • Entity-Aware Transport	22-9
22.5	Real-Time Capacity of Wireless Networks The Single-Path Feasible Region • Approximate Total Capacity	22-11
22.6	Concluding Remarks	22-13

22.1 Introduction

In recent decades, computer systems have been embedded into physical environments for automated real-time monitoring and control. This trend will continue, and even expand, to improve and secure our quality of life in many areas such as defense, emergency rescue, biosensing, pervasive computing, and assisted living. Analyzing application scenarios such as (1) cooperative mobile robots for rescue activities and (2) wireless sensor networks for surveillance/biosensing, some important observations can be made. First, the system requires *end-to-end real-time performance* to ensure emergency response within bounded delay; second, mobility and ease of deployment are mandatory requirements in the described scenarios; hence, wired backbone or wireless links with infrastructure are not viable options. Finally, the described applications are mainly event driven and the runtime workload is dynamic, hence, real-time data flows need to be established on demand, they require bounded delay, low jitter, and guaranteed data throughput. Since timeliness is a property tightly related to the way each shared resource (such as the wireless medium) is managed at a low level, this chapter will first cover protocols and challenges related to the medium access control (MAC) layer, followed by an overview of a light-weight real-time communication architecture (RAP), a delay-sensitive network-layer protocol (SPEED), and a notion of network capacity to quantify the ability of the network to transmit information in time.

MAC protocols have been widely studied in wireless networks; however, most MAC schemes are primarily designed to achieve a high-average throughput and fairness among multiple users; real-time guarantee and jitter control are usually considered secondary goals and several protocol extensions exist that are able to provide some degree of quality of service (QoS). Focusing the attention on MAC protocols that provide real-time or QoS support, it is possible to broadly categorize them into two classes [14]: *synchronous schemes* that require global time synchronization among different nodes and *asynchronous schemes*, which do not require such support.

Synchronous schemes include Implicit-EDF [8], TBMAC [10], Cluster TDMA [17], Cluster Token [16], and MAX [22]. All these protocols work by having nodes that implicitly agree on a transmission slot assignment. The agreement requires both time synchronization and either a regular network structure or some form of clustering. IEEE 802.11 Distributed Coordination Function (DCF) is a widely used standard asynchronous protocol, but its ability to provide reserved bandwidth to differentiated multihop traffic is poor [15]. Several schemes have been developed to improve over DCF by either providing support for differentiated traffic categories (see real-time MAC [6] and DCF with priority classes [11]), or by reducing collisions in multihop networks (see MACA/PR [18]). Though these schemes are easy to implement over the IEEE 802.11 protocol, owing to the nature of 802.11 random backoff they cannot provide delay bounds for real-time traffic.

Among the class of asynchronous schemes, there are several protocols based on Black-Burst (BB) [26,27] and they have been proposed to support real-time traffic [13,24]. However, all such protocols assume a fully connected network, and are not easily extensible to a multihop environment. Robust and implicit earliest deadline first (RI-EDF) [9] is able to provide real-time guarantee, low jitter, and high bandwidth in single-hop networks by relying on a shared packet schedule among all nodes. As such, RI-EDF is also hard to extend to large networks. The implicit contention scheme adopted by RI-EDF and Implicit-EDF introduces small overhead outperforming BB in terms of achievable network throughput [8]. Out-of-band signaling has been used for channel reservations in multihop networks in different works [23,28,32]. Although these protocols use multiple wireless channels to provide better temporal QoS, they need a transceiver that is able to simultaneously listen to two different channels at once, which is not practical in resource-constrained embedded systems.

22.2 Basic Concepts for Predictable Wireless Communication

When deploying a real-time wireless network (single or multihop), both the physical layer and MAC can greatly affect its temporal predictability. In fact, at physical layer radio-frequency (RF) interference, large-scale path loss and fading cause adverse channel conditions by reducing *signal-to-noise ratio* (SNR) of the wireless communication. When the SNR is lower than a certain threshold, the *bit error rate* (BER) of the wireless communication rises over the acceptable limit, thus disrupting the wireless connection. The key to maintain wireless communication under adverse channel conditions is to provide as high SNR as possible. To achieve this, a promising solution lies in the state-of-the-art *direct sequence spread spectrum* (DSSS) [30,31] technology, which allows trade-offs between data throughput versus SNR. It is easy to understand that high BER would cause packet retransmissions and most probably deadline misses. While the physical layer is out of the scope of this chapter, the following sections will focus on MAC techniques, will introduce a light-weight real-time communication architecture, and a notion of network capacity for real-time wireless networks. To make the idea of real-time wireless networks work, two main problems need to be addressed at the MAC layer:

- Packet collisions on wireless channel and
- Priority inversions when accessing the wireless medium

A packet collision occurs when two nearby nodes decide to transmit a packet at the same time resulting in a collision; priority inversion occurs each time a node carrying a high-priority packet loses wireless medium contention against another node that has a lower-priority packet: an ideal real-time wireless network should not experience any packet collision and should experience only bounded priority inversions.

To avoid packet collisions and mitigate the priority inversion problem caused by the multihop nature of large wireless networks, a prioritized medium access scheme should be used, for instance, a suitable class of protocols is the one based on BB [26]. The original BB approach was designed to achieve fair channel sharing in wireless LAN; to achieve real-time performance, the BB channel contention scheme can be used to differentiate among different real-time priority levels. In a single-hop environment, packet collisions can be avoided by using standards such as IEEE 802.15.4 and a network coordinator; similarly, IEEE 802.11 PCF implements the notion of contention-free period to avoid conflicts and enforce a resource reservation mechanism. Both the standards allow to provide temporal QoS. In the next section, RI-EDF will be presented as real-time single-hop MAC scheme; it is able to provide real-time guarantee, low jitter, and high bandwidth by relying on a shared packet schedule among all nodes. Compared to existing standards, RI-EDF has the advantage of having lower overhead and increased robustness; however, this access scheme is suitable for mainly periodic and semiperiodic real-time traffic.

In a multihop environment, temporal predictability is a challenging task and an open research problem the real-time and network communities are looking at. In the subsequent sections, we shall briefly describe routing protocols that address time and distance constraints. We shall also describe appropriate higher-level communication abstractions. Finally, a notion of network capacity will be derived to quantify the ability of the network to transmit information in time. As a final remark, it is important to notice that every wireless communication setting is always subject to the unpredictability of the wireless medium despite any efforts of building an ideal collision-free and priority inversion-free wireless MAC; as such, all the protocols described in this chapter cannot guarantee hard real-time constraints. Instead, they focus on providing soft real-time guarantees. Therefore, all performance bounds and real-time guarantees expressed in the following sections are provided subject to the assumption that the wireless medium is not affected by jamming or electromagnetic interference (EMI).

22.3 Robust and Implicit Earliest Deadline First

The RI-EDF protocol [9] derives a schedule for the network by using EDF [19]. In fact, nodes transmit according to the obtained schedule, implicitly avoiding contention on the medium. A node transmits when it receives the entire packet train from the previous node or receives a recovery packet and is its turn in the schedule for sending a packet; otherwise, a node transmits when it must recover the network.

22.3.1 Terminology and Assumptions

A network consists of q nodes labeled N_i , where $i = 1, 2, \dots, q$. The maximum packet size, in terms of transmission time (seconds), is denoted by θ . M_j is a periodic message characterized by (m_j, T_j, i) , where m_j is the message length, T_j the period of the message, and i the index of the transmitting node. Both m_j and T_j are measured in seconds. Aperiodic messages are denoted by the same tuple, where T_j represents the minimum interarrival time between messages. Aperiodic messages can be scheduled using an aperiodic server, such as a polling server [20]. In this way, each server can be treated as a periodic message for the purpose of schedulability analysis. \mathcal{M} is the set of all periodic messages and aperiodic servers in the system.

The network schedule is derived from \mathcal{M} for a single hyperperiod using the rules of EDF. The hyperperiod, H , is the least common multiple of all the message periods. Nodes are prioritized according to i . A lower i denotes a higher priority. During schedule derivation, ties are broken in favor of the node with the highest priority. The network schedule is represented by a sequence of *packet trains*. Each packet train is a sequence of contiguous packets transmitted by a single node and characterized by the index of sender node and duration of the packet train itself. A packet train includes one or more packets, each denoted by a packet number p_j . Note that the length of the last packet of each packet train is less than the maximum size θ , if the duration of the packet train is not a multiple of the maximum packet size.

More formally, the k th packet train of the network schedule within a single hyperperiod is denoted by $PT_k(t_s, t_f, i)$, for $k = 0, \dots, K$, where t_s denotes the start time of the packet train, t_f the finish time of the packet train, and i the index of the transmitting node. The packet train currently being transmitted by the

Packet train	PT_0		PT_1		PT_2		PT_3	PT_4
Node	N_1		N_2		N_1		N_2	N_3
Duration	2–0		4–2		6–4		7–6	8–7
Packets	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7

FIGURE 22.1 Example of schedule.

current node, N_{curr} , is denoted by PT_{curr} . PT_{prev} denotes the previously transmitted packet train by node N_{prev} . The remaining time a node has to send PT_{curr} is its *budget*. The budget for N_i is defined as a function of time t and is denoted by $C_i(t)$. At the beginning of a packet train transmission, $C_i(t) = t_f - t_s$. At any time t during transmission, $C_i(t) = t_f - t$ is the remaining time for a packet train transmission. Each packet has an identifying *packet number* p_j , where j is incremented starting at 0 within the hyperperiod. Each node has an *internal state* s_i given by the packet number p_j of the last correctly received or transmitted packet. Every node maintains its own internal state. Nodes utilize the network schedule to determine, which node is next to transmit. Figure 22.1 summarizes the network schedule definition and provides the schedule for the example in Section 22.3.2.

RI-EDF is robust in the event of packet losses and node failures; in fact, assuming a fully linked network, it always guarantees a conflict-free schedule and has a distributed recovery mechanism that allows to recover the network communication in the event of node failures. This protocol does not consider security aspects such as nodes transmitting malicious packets, or other forms of Byzantine failures. Packet data are always trusted. The timer set by each node to perform network recovery is the *recovery timer*. The length of the recovery timer for node N_i is proportional to the static priority of the node.

The following assumptions are made about the network: (1) The network is fully linked; that is, every node is within direct transmission range of every other node; (2) All nodes have knowledge of current network schedule or the knowledge of \mathcal{M} (the set of all periodic messages and aperiodic servers in the system); and (3) Nodes that do not successfully receive a transmitted packet are still capable of using carrier sense to detect a busy channel.

22.3.2 The RI-EDF Protocol

This section describes the RI-EDF protocol. Throughout this section, the schedule of Figure 22.1 is used as an example. The example schedule consists of three nodes $\{N_1, N_2, N_3\}$, sending the messages $M_0 = (2, 4, 1)$, $M_1 = (3, 8, 2)$, and $M_2 = (1, 8, 3)$ on the network. The schedule is derived using the rules of EDF. Using node priority to break ties, the schedule, expressed in terms of packet trains is $\{PT_0 = (0, 2, 1)$, $PT_1 = (2, 4, 2)$, $PT_2 = (4, 6, 1)$, $PT_3 = (6, 7, 2)$, $PT_4 = (7, 8, 3)\}$. For simplicity, the examples assume a constant packet size with $\theta = 1$. RI-EDF uses two mechanisms to ensure robustness in cases of varied packet size, node failure, and packet loss: *budget* and *recovery*.

Budget. A node that completes its transmission early causes schedule drift. The subsequent node may not have a packet prepared for transmission and thus forced to forgo transmission. To prevent schedule drift, RI-EDF uses a *budget*. The unused budget of one node may be forwarded to the immediately following node in the schedule. A node may have unused budget if its periodic data transmission is less than the scheduled amount. The forwarded budget may then be used by the subsequent node to transmit any aperiodic messages until its periodic messages are ready. Δ denotes the budget forwarded to one node by its previous node. Unused budget cannot be saved by a node for use in transmitting its later packet trains.

To support the budget mechanism, the RI-EDF protocol uses five fields in each *data packet*: (1) Type indicates that this is a data packet; (2) Packet train completion the final packet of a packet train; (3) Forwarded budget the forwarded budget Δ conveyed by the previously transmitting node; (4) Packet

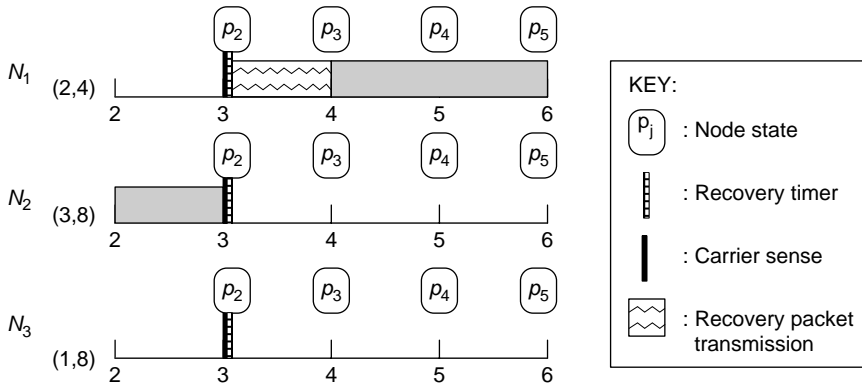


FIGURE 22.2 RI-EDF utilizes recovery mechanism to avoid network stall in the event of N_2 failure.

number the packet number p_j within the hyperperiod; and (5) Payload the variable-length data. When PT_k begins transmission, its budget $C_k(t)$ is initialized to $(t_f - t_s)$.

Recovery. Care must be taken to guarantee that packet transmissions continue, even in the presence of node failures or packet losses. For this purpose, a recovery mechanism is used. Upon the transmission or reception of a packet, each node performs carrier sense on the medium. If the medium is busy, no recovery is performed. The maximum idle time permitted is t_{idle} . If the medium is idle for t_{idle} , it is assumed that transmissions have stalled and recovery must be performed. Each node sets a recovery timer for a time proportional to its static priority. If the recovery timer expires on a given node, the node sends a recovery packet using its own internal state. Nodes that successfully receive the recovery packet update their internal state with the recovery packet number and can thus determine N_{curr} .

To support the recovery mechanism, RI-EDF uses three fields in each *recovery packet*: (1) Type indicates that this is a recovery packet; (2) Packet number the packet number p_j within the hyperperiod; and (3) Payload the variable-length data. Depending on implementation decisions of the designer, the payload can be zero-length or it can be used to send aperiodic data. Notice the recovery operation in Figure 22.2 in which N_2 fails to transmit one of its packets. N_2 transmits successfully at time $t = 2$. Nodes N_1 and N_3 successfully receive packet p_2 and update their internal state. At time $t = 3$, N_2 fails to transmit. Each node performs carrier sense and sets a recovery timer after t_{idle} . Node N_1 has the highest-recovery priority, and its recovery timer expires first. It then sends a recovery packet using its internal state to indicate that p_3 should have been sent. After recovery, N_1 transmits packets p_4 and p_5 as normal.

22.3.2.1 RI-EDF Rules

Figure 22.3 summarizes the rules of the RI-EDF protocol as a finite state machine. The five states are (1) Carrier Sense I; (2) Carrier Sense II; (3) Receive; (4) Transmit; and (5) Recovery.

1. *Carrier Sense I*: the initial state. This state is also reached when a node has completed transmission and it is not N_{curr} , or because a busy medium has been detected. Disable recovery timer. Perform carrier sense.
2. *Carrier Sense II*: this state is reached upon the detection of an idle medium for t_{idle} . Set recovery timer for a time proportional to the static priority of N_i . If a busy medium is detected, return to State 1, Carrier Sense I. If recovery timer expires, proceed to State 5, Recovery.
3. *Receive*: this state has been reached because a packet has been received by the node. Record a MAC-layer timestamp \bar{t} . Set the node's internal state to the received packet number. If the completion bit is set in the received packet, and this node is N_{curr} , set the budget to any forwarded budget plus the transmission time of PT_{curr} , that is, $C_{curr}(t) = \Delta + t_f - t_s$. Proceed to State 4, Transmit. Otherwise, return to State 1, Carrier Sense I.

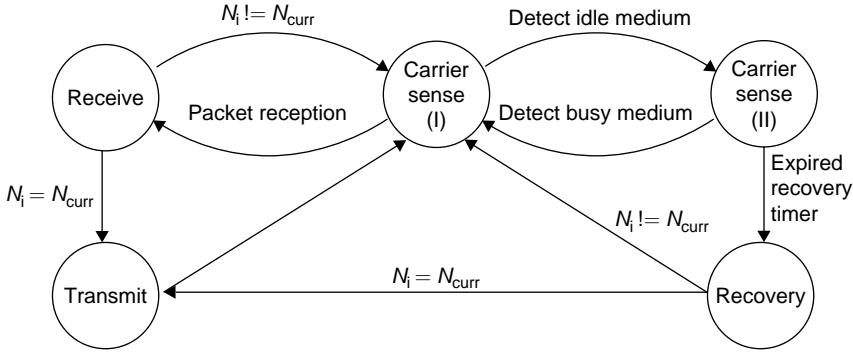


FIGURE 22.3 RI-EDF protocol.

4. *Transmit*: this state has been reached because this node, N_i , is N_{curr} , the currently transmitting node. *If there is a forwarded budget Δ* : Transmit aperiodic messages whose total transmission time is no greater than Δ . If there is a forwarded budget and no aperiodic messages remain, transmit one or more zero-payload data packets to hold the medium for Δ . While holding the medium or transmitting aperiodics, the packet number transmitted is the internal state of the node and the internal state is not altered. *If there is no forwarded budget, or the forwarded budget is exhausted*: The budget is $C_{curr}(t)$. The packet number is equal to the internal state of this node plus one. Send periodic packets, ensuring that the packet transmission time does not exceed the allocated budget of the node. After each packet transmission, set the internal state to the most recently transmitted packet number. The final packet in the packet train sets the completion bit and conveys any forwarded budget to the next node. *If there are no periodics*: Transmit a zero-payload data packet to forward this node's budget. At transmission completion, return to State 1, Carrier Sense I.
5. *Recovery*: this state has been reached because a recovery packet must be transmitted. All forwarded budget is discarded. The packet number is equal to the internal state of this node plus one. Transmit a *recovery* packet. Update the internal state of the recovery node to this packet number. If after updating, the recovery node is now N_{curr} , proceed to State 4, Transmit. Otherwise, return to State 1, Carrier Sense I.

To use the proposed wireless MAC scheme, the network has to be initialized by building the network schedule, disseminating it among all the nodes, and finally by starting the transmission of the first packet. A simple way of initializing the network is to download offline the initial network schedule on all nodes; in fact, as soon as the first node is turned on, it starts to send recovery packets and its reserved packets until one by one all the other nodes are turned on. As more nodes are activated, less recovery packets are transmitted on behalf of the missing nodes. Eventually, all the nodes are turned on and the network schedule is followed by all of them. Basically, the recovery mechanism takes care of bootstrapping the network independently of the activation order of the nodes.

22.3.2.2 Protocol Properties

Property 22.1

Assuming a fully linked network, RI-EDF maintains a conflict-free schedule in the presence of node failures and packet losses without a centralized scheduler.

Proof

In the absence of packet losses and node failures, every node is implicitly aware of the schedule. At every scheduling instant (the instant of packet reception), there is a single node trying to send a packet in the channel. Thus, a conflict-free schedule is maintained.

Packet losses and node failures: Consider a packet loss at a node N_k . That is, N_k does not receive the previously transmitted packet, p_j . Let N_p be the live node, which has the highest static priority. Let N_q be the node, which is scheduled for transmission after the dropped packet.

Case 1: N_q is alive and it has received p_j . N_q becomes N_{curr} , and thus has the highest priority to transmit. Every other node, including N_k , will refrain from transmission after it detects a busy medium. The schedule is unaltered and still conflict-free.

Case 2: $N_q = N_k$. Thus, the node to transmit next did not receive p_j successfully. As a result, N_p becomes the node with the highest static priority to recover the schedule. N_p transmits the sequence number in the recovery packet. When N_p recovers the schedule, every other node, including N_k , will detect a busy medium and refrain from transmission. Thus, the schedule is maintained conflict-free.

An analogous argument holds in the case of node failure at N_k . Note that a conflict-free schedule is maintained since detection of the idle channel is the synchronization point for all nodes belonging to a fully linked network.

Property 22.2

RI-EDF has no central point of failure.

This property follows immediately from the fact that all nodes are involved in the recovery mechanism.

Schedulability Analysis: In the absence of packet loss and node failures, the schedulability of a set \mathcal{M} of real-time periodic messages and aperiodic servers (denoted by (m_j, T_j, i) , where $j = 1, 2, \dots, n$) can be tested by the following sufficient condition [5]:

$$\forall j, \quad 1 \leq j \leq n \quad \sum_{k=1}^j \frac{m_k}{T_k} + \frac{\theta}{T_j} \leq 1$$

which assumes that all the messages are sorted by increasing periods, that is, $T_i \leq T_j$ only if $i < j$. It is worth noting that the blocking time of each message is equal to the maximum packet size θ since packets are nonpreemptable.* Under the assumption that the above schedulability condition is verified, that there are no packet losses and the network is fully linked (ideal case), the delay of all real-time periodic messages is bounded and it is easy to verify that the maximum delay of each message M_j never exceeds twice its period $2T_j$. In fact, according to EDF, two consecutive instances of the same periodic message can never be scheduled more than two periods apart otherwise the schedulability guarantee would fail.

22.3.3 Power-Aware RI-EDF

In some applications targeted by the RI-EDF protocol, low power consumption is mandated. In the RI-EDF protocol, the channel is in constant use, even in cases where there are no packets ready for transmission. As a result, a power-aware extension is introduced for the RI-EDF protocol. This extension is based on differentiating between nodes that are *sources* or *sinks*. Sources are those nodes that transmit data. Sinks are those nodes that are interested in the actual *data* sent in each packet; data that will be processed later in some fashion. Sinks may also transmit data. To support this extension, two additions are made. First, a field is added to the packet headers to denote whether a packet was transmitted by a sink or source node. Second, sinks must have a periodic beacon packet in the schedule that may be optionally transmitted each hyperperiod to solicit data from the sources. Using these additions, the power-aware RI-EDF extension rules are as follows:

After a source node performs recovery, its recovery mechanism is disabled and it is no longer able to recover the network. Once its recovery mechanism is disabled, a source node can send up to \mathcal{P} data packets before it may no longer transmit. \mathcal{P} is based on the design decisions of the network developer. Recovery is

*This real-time guarantee does not hold in the presence of packet loss since RI-EDF does not include reservations for packet retransmissions. Ref. 9 summarizes the effects of packet loss and hidden nodes on RI-EDF, and experimentally shows a gradual degradation of performance.

reenabled in sources only when they receive a packet sent by a sink. Sinks have a higher static priority than the sources in the event that recovery must take place, allowing sources to reenabling their recovery mechanisms as quickly as possible. In addition, sinks do not disable their recovery mechanisms unless they want to sleep. This extension has two advantages. First, if a sink does not require any data, sources will stop transmitting, thus avoiding power consumption for unnecessary transmissions. Second, source nodes that may move out of the range of sinks will stop transmitting after a maximum of \mathcal{P} packets after which the sources may sleep. In addition, sources may sleep at any time and wake up periodically to determine if they can hear a sink in which case they will immediately rejoin the schedule. This sleeping conserves power in the sources.

The current consumption of TinyOS and power-aware RI-EDF protocols were compared. Five sources generated real-time random traffic requests for throughput ranging from 200 to 1000 bytes/s. Each source disabled its radio between requests. The state of the radio was monitored, observing the time it was in a transmit or receive state. Using the time spent in each of these states, current consumption was estimated for each protocol. As summarized in Tables 22.1 and 22.2, at low channel utilization, RI-EDF consumed more current than the TinyOS protocol. With increased utilization, RI-EDF consumes less current while transmitting more packets than TinyOS. This is a direct result of RI-EDF's fewer dropped packets due to reduced contention on the medium. This experiment also demonstrates that RI-EDF experiences very few deadline misses compared to TinyOS. In particular, RI-EDF experienced deadline misses only when the wireless channel was overloaded. Finally, in Ref. 9 experiments were performed to compare the communication throughput of RI-EDF versus the TinyOS MAC protocol. In the throughput experiments, seven Motes were employed, all within the range of each other. An eighth Mote was used to monitor all packet transmissions. The packet size was chosen such that no budget would be forwarded during normal operation. The RI-EDF protocol achieves transmission rates close to the maximum speed of the radio (19.2 Kbps). Although RI-EDF has more overhead, data throughput is greater for all tested payload sizes. For the largest payload size, RI-EDF has an additional 10.8% overhead over TinyOS, including the overhead required for the Dynamic Schedule Update extension described in Section 22.3.4. Despite the overhead, RI-EDF allows a 26% increase in transmitted data.*

TABLE 22.1 Tiny OS Current Consumption

Target Bps	Actual Bps	% Collisions	% Missed Deadlines	% Time in		Current (mA)
				Trans.	Recv.	
200	199.0	0.33	0	3.66	7.52	1.338
400	392.9	0.81	0	7.09	19.22	3.037
600	597.0	1.80	0	10.67	36.89	5.334
800	780.9	3.62	29.28	13.60	70.12	9.017
1000	820.9	4.32	69.60	14.22	81.79	10.240

TABLE 22.2 RI-EDF Current Consumption

Target Bps	Actual Bps	% Collisions	% Missed Deadlines	% Time in		Current (mA)
				Trans.	Recv.	
200	204.9	0	0	3.77	11.60	1.746
400	404.8	0	0	7.42	24.46	3.595
600	602.1	0	0	11.03	39.82	5.675
800	802.1	0	0	14.68	55.65	7.809
1000	1001.3	0	0.25	18.31	72.25	10.012

*The restricted packet sizes of MICA2 Motes skews the throughput statistics against RI-EDF owing to its 2-byte packet header overhead. This overhead would be insignificant on implementations permitting larger packet sizes.

22.3.4 Dynamic Schedule Updates

Dynamically updating the RI-EDF schedule is necessary in applications where a node's participation in the network is changing. Owing to the mobility of some networks, certain applications may be interested in removing nodes from or adding nodes to the schedule. Other applications may be interested in updating the schedule based on changing demands of existing nodes. A simple mechanism to update the system schedule is presented here. The mechanism for adding new nodes utilizes a reservation for a packet train at the end of a hyperperiod, or once in several hyperperiods. This packet train reservation, denoted as PT_{res} , is dedicated to node addition requests and replies, schedule update requests, and new schedule announcements.

Requests are serviced by a leader node, the node with the highest static priority in the system.* This node could also be elected by approaches in Refs. 29 and 33, or even simply using node address-based election. Nodes intending to join contend to transmit during PT_{res} . Nodes transmit a random request identifier, which will be used in the acknowledgment to differentiate between requests. All requests are processed in FIFO order. The leader node replies to the join requests with a positive or negative acknowledgment, depending on if the schedule can accommodate the request. The priorities for transmission during the PT_{res} slot are as follows, with 1 being the highest static priority: (1) *Schedule announcements after (a) join requests and (b) dynamic schedule update requests*. This priority is mapped to the priority of the leader node. (2) *Dynamic schedule update requests and join requests*. This new static priority level is called *join priority*. (3) *Unrequested schedule announcements*. This priority is mapped to the standard fixed priorities of the nodes. If there are no schedule updates or join requests, the node with the highest fixed priority that is alive, including the leader, will transmit this *unrequested schedule announcement* to announce the current schedule.

This mechanism can also be used for removing nodes from the schedule. Nodes make an exit request during the reserved slot in the hyperperiod. The leader node acknowledges the request, and the schedule is recomputed and announced appropriately. The schedule can also be updated without receiving requests during PT_{res} . Existing nodes can make requests to change their utilization in the schedule.

This mechanism employs a *new schedule* bit, or *NS* bit in the data packet header to indicate that a newly derived schedule will be used starting at the next hyperperiod. Once the leader node acknowledges a request and announces a new schedule, all nodes set the *NS* bit in their subsequent data packet transmission until the new schedule begins in the following hyperperiod. This is done to increase the chance that all nodes receive a packet with a set *NS* bit before the start of the next hyperperiod at which point the new schedule begins. Also added to the data packet header is the *schedule identifier*. The schedule identifier is a value identifying the schedule with which the transmitted data packet is associated. Each time a new schedule is announced by the leader node, an accompanying schedule identifier is included. Each node keeps an internal copy of the current schedule identifier. This internal copy of the schedule identifier is updated only when a node successfully receives the accompanying schedule announcement during PT_{res} . The *NS* bit and schedule identifier help to mitigate the problems that arise from a protocol such as RI-EDF, which does not employ consensus. These two header fields help to insure that nodes do not transmit if they do not have the correct schedule.

22.4 Higher-Level Real-Time Protocols for Sensor Networks

This section briefly reviews higher-level communication protocols in sensor networks. An earlier draft was presented at the DDRTS 2003 workshop [3].

22.4.1 RAP: Real-Time Distance-Aware Scheduling

Message communication in sensor networks must occur in bounded time, for example, to prevent delivery of stale data on the status of detected events or intruders. In general, a sensor network may simultaneously carry multiple messages of different urgency, communicated among destinations that

*The leader node is equivalent to a sink node described in Section 22.3.3.

are different distances apart. The network has the responsibility of ordering these messages on the communication medium in a way that respects both time and distance constraints.

A protocol that achieves this goal is called RAP [21], developed at the University of Virginia. It supports a notion of packet velocity and implements velocity-monotonic scheduling (VMS) as the default packet scheduling policy on the wireless medium. Observe that for a desired end-to-end latency bound to be met, an in-transit packet must approach its destination at an average velocity given by the ratio of the total distance to be traversed to the requested end-to-end latency bound. RAP prioritizes messages by their required velocity such that higher velocities imply higher priorities. Two flavors of this algorithm are implemented. The first, called *static velocity-monotonic scheduling*, computes packet priority at the source and keeps it fixed thereafter regardless of the actual rate of progress of the packet toward the destination. The second, called *dynamic velocity-monotonic scheduling*, adjusts packet priority *en route* based on the remaining time and the remaining distance to destination. Hence, a packet's priority will increase if it suffers higher delays on its path and decrease if it is ahead of schedule.

To achieve consistent prioritization in the wireless network, not only does one need priority queues at nodes, but also a MAC layer that resolves contention on the wireless medium in a manner consistent with message priorities. The authors of RAP [21] used a scheme similar to Ref. 1 to prioritize access to the wireless medium. The scheme is based on modifying two 802.11 parameters, namely the DIFS counter and the backoff window, such that they are priority-aware. An approximate prioritization effect is achieved by letting backoff depend on the priority of the outgoing packet at the head of the transmission queue. Alternatively, some version of BB could have been used.

A detailed performance evaluation of this scheme can be found in Ref. 21. It is shown that velocity-monotonic scheduling substantially increases the fraction of packets that meet their deadlines taking into consideration distance constraints.

22.4.2 SPEED: Enforcement of Velocity Constraints

RAP leads the idea that a network can support multiple predefined velocities. An application chooses a velocity level for each message. The network guarantees that the chosen message velocity is observed with a very high probability as long as the message is accepted from the application. A network-layer protocol with the above property, called SPEED [12], has been developed at the University of Virginia. The protocol defines the velocity of an in-transit message as the rate of decrease of its straight-line distance to its final destination. Hence, for example, if the message is forwarded away from the destination, its velocity at that hop is negative.

The main idea of SPEED is as follows. Each node i in the sensor network maintains a neighborhood table that enumerates the set of its one-hop neighbors. For each neighbor, j , and each priority level, P , the node keeps a history of the average recently recorded local packet delay, $D_{ij}(P)$. Delay $D_{ij}(P)$ is defined as the average time that a packet of priority P spends on the local hop i before it is successfully forwarded to the next-hop neighbor j . Given a packet with some velocity constraint, V , node i determines the subset of all its neighbors that are closer to the packet's destination. If L_{ij} is the distance by which neighbor j is closer to the destination than i , the velocity constraint of the packet is satisfied at node i if there exists some priority level P and neighbor j , such that $L_{ij}/D_{ij}(P) \geq V$. The packet is forwarded to one such neighbor nondeterministically. If the condition is satisfied at multiple priority levels, the lowest priority level is chosen. If no neighbor satisfies the velocity constraint, we say that a local deadline miss occurs.

A table at node i keeps track of the number of local deadline misses observed for each velocity level V . This table is exchanged between neighboring nodes. Nodes use this information in their forwarding decisions to favor more appropriate downstream hops among all options that satisfy the velocity constraint of a given packet. No messages are forwarded in the direction of nodes with a high miss-ratio. The mechanism exerts backpressure on nodes upstream from congested areas. Congestion increases the local miss-ratio in its vicinity, preventing messages from being forwarded in that direction. Messages that cannot be forwarded are dropped thus increasing the local miss-ratio upstream. The effect percolates toward the source until a node is found with an alternative (noncongested) path toward the destination, or the source

is reached and informed to slow down. The mentioned scheme is therefore effective in exerting congestion control and performing packet rerouting that guarantees the satisfaction of all velocity constraints in the network at steady state [12]. The protocol is of great value to real-time applications where different latency bounds must be associated with messages of different priority.

22.4.3 Entity-Aware Transport

Although RAP and SPEED allow velocity constraints to be met, the abstractions provided by them are too low level for application programmers. A transport layer is needed whose main responsibility is to elevate the degree of abstraction to a level suitable for the application. One such transport layer is described in Ref. 7. The authors propose a transport layer in which connection endpoints are directly associated with events in the physical environment. Events represent continuous external activities, such as the passage of a vehicle or the progress of a fire, which is precisely what an application might be interested in. By virtue of this layer, the programmer can describe events of interest and logically assign “virtual hosts” to them. Such hosts export communication ports and execute programs at the locations of the corresponding events. The programmer is isolated from the details of how these hosts and ports are implemented. When an external event (e.g., a vehicle) moves, the corresponding virtual host migrates with it transparently to the programmer.

We call the virtual host associated with an external event of interest an *entity*. Sensor nodes that can sense the event are called *entity members*. Members elect an *entity leader* that uniquely represents the entity and manages its state. Hence, an entity appears indivisible to the rest of the network. The fact that it is composed of multiple nodes with a changing membership is abstracted away. It is key to maintain a unique entity (with a single leader) to describe any specific external object or activity. This is called *entity uniqueness*. It is needed to maintain the abstraction that communication endpoints are logically attached to mobile targets.

An evaluation of this architecture reveals that entity uniqueness is maintained as long as the target event moves in the environment at a speed slower than half the nodes’ communication radius per second [7]. For example, if sensor nodes can communicate within a 200-m radius, the transport layer can correctly maintain endpoints attached to targets that move as fast as 100 m/s (i.e., 360 km/h). The combination of this transport layer and the guaranteed velocity protocols described earlier provides invaluable support to real-time applications. For example, communication regarding moving targets can be made to proceed in the network at a velocity that depends on target velocity itself. Hence, positions of faster targets, for example, can be reported quicker than those of slower ones.

22.5 Real-Time Capacity of Wireless Networks

The protocols described above attempt to provide real-time communication in sensor networks such that data are delivered on time. An interesting theoretical question is to quantify the capacity of a sensor network to deliver information by deadlines. This is called *real-time capacity*.

In this section, the theoretical foundations and approach are presented for deriving expressions of real-time capacity of sensor networks. We begin by defining real-time capacity more formally. Real-time capacity, first introduced in Ref. 2, refers to the total byte-meters that can be delivered *by their deadlines*. To make the capacity expressions independent of the details of the workload (such as the deadline values themselves), a normalized bound is defined that quantifies the total byte-meters that can be delivered for each time unit of the relative deadline. To illustrate the notion of real-time capacity, consider a network with two flows, *A*, and *B*. Flow *A* must transfer 1000 bytes a distance of 50 m (i.e., a total of 50,000 byte-meters) within 200 s. It is said to have a real-time capacity requirement of $50,000/200 = 250$ byte-meters/s. Flow *B* must transfer 300 bytes a distance of 700 m within 100 s. Its capacity requirement is thus $300 * 700/100 = 2100$ byte-meters/s. Hence, the total real-time capacity needed is $2100 + 250 = 2350$ byte-meters/s. An interesting question is whether one can establish the total real-time capacity a communication

network can support as a function of different network parameters, such that all flows meet their deadlines as long as their collective capacity requirements do not exceed the derived capacity bound.

22.5.1 The Single-Path Feasible Region

To compute the capacity bound, we first establish the schedulability condition for one path in the network. We then use it to find the total amount of network traffic that can meet deadlines. Consider a sensor network with multiple data sources and data sinks. Packets traverse the network concurrently, each following a multihop path from some source to some destination. Each packet T_i has an arrival time A_i defined as the time at which the sending application injects the packet into the outgoing communication queue of its source node. The packet must be delivered to its destination no later than time $A_i + D_i$, where D_i is called the relative deadline of T_i . Different packets may generally have different deadlines. We call packets that have arrived but whose delivery deadlines have not expired as *in-transit* packets. Each packet T_i has a transmission time C_i that is proportional to its length. This transmission time is incurred at each forwarding hop of its path.

Increasing the amount of data in the network or reducing some of the deadlines will decrease schedulability. A metric called *synthetic utilization* is defined that captures the impact (on schedulability) of both the resource requirements and urgency associated with packets. Each packet contributes an amount C_i/D_i to the synthetic utilization of each hop along its path in the interval from its arrival time A_i to its absolute deadline $A_i + D_i$. Observe that schedulability decreases with increased synthetic utilization. This leads to the idea that a bound on synthetic utilization may exist that separates schedulable from (potentially) unschedulable workloads.

At any time t , let $S(t)$ be the set of packets that are in-transit in the entire sensor network. Hence, $S(t) = \{T_i | A_i \leq t < A_i + D_i\}$. Let $S_j(t) \in S(t)$ be the subset of $S(t)$ that passes through node j . The synthetic utilization, $U_j(t)$ of node j is $\sum_{T_i \in S_j(t)} C_i/D_i$, which is the sum of individual contributions to synthetic utilization (on this node) accrued over all in-transit packets passing through that node.

Consider an arbitrary path P through the sensor network without loss of generality. Let us number the hops of that path $1, \dots, N$ from source to destination. An interesting question is to find a function $g(U_1, \dots, U_N)$ of the synthetic utilization of each hop along the path, such that the end-to-end deadlines of all packets transmitted along that path are met when $g(U_1, \dots, U_N) \leq B$, where B is a constant bound.

The feasible region was computed in Ref. 4 for an arbitrary fixed-priority scheduling policy. To quantify the effect of the scheduling policy, a parameter α is defined. Intuitively, α represents the degree of *urgency inversion* in the priority assignment observed under the given scheduling policy. An urgency inversion occurs when a less-urgent packet (i.e., one with a longer relative deadline) is given an equal or higher priority compared to a more urgent one. Let us call them packet T_{hi} and T_{lo} , respectively. Formally, we define $\alpha = \min_{T_{hi} \geq T_{lo}} D_{lo}/D_{hi}$, which is the minimum relative deadline ratio across all priority-sorted packet pairs. In the absence of urgency inversion (e.g., for deadline monotonic scheduling), $\alpha = 1$. Otherwise, $\alpha < 1$. For example, if priorities are assigned randomly, $\alpha = D_{least}/D_{most}$, where D_{least} and D_{most} are the minimum and maximum relative deadlines in the packet set, respectively. Hence, the objective is to find a function $g(U_1, \dots, U_N, \alpha)$, such that $g(U_1, \dots, U_N, \alpha) \leq B$ implies that all packets are schedulable for the particular fixed-priority scheduling policy.

It was shown in Ref. 4 that the feasible region for such a scheduling policy is

$$\sum_{j=1}^N \frac{U_j(1 - U_j/2)}{1 - U_j} < \alpha \quad (22.1)$$

In particular, for deadline monotonic scheduling, $D_n/D_{max} \geq 1$, or $\alpha = 1$. The feasible region of path P under deadline monotonic scheduling is therefore

$$\sum_{j=1}^N \frac{U_j(1 - U_j/2)}{1 - U_j} < 1 \quad (22.2)$$

22.5.2 Approximate Total Capacity

The results derived above represent exact sufficient conditions on path schedulability. In Ref. 2, these results were used to derive a network real-time capacity bound. A packet *always* meets its end-to-end deadline as long as Equation 22.1 holds for its path. The main contribution of Equation 22.1 lies in relating end-to-end delay to a bound on the sum of throughput-like metrics (synthetic utilizations). These metrics can now be related to real-time capacity, hence establishing the real-time capacity bound. In the following, we give a brief sketch of how the above results can be used to derive an approximate real-time capacity for the network.

From Equation 22.1, the average synthetic utilization U of a node on a communication path of length N satisfies

$$\frac{U(1 - U/2)}{1 - U} < \alpha/N \quad (22.3)$$

Solving for U , we get

$$U < 1 + \frac{\alpha}{N} - \sqrt{1 + \left(\frac{\alpha}{N}\right)^2} \quad (22.4)$$

Remember that, by definition, $U = \sum_i C_i/D_i$ over all in-transit packets through a node. Since multiplying the packet transmission time, C_i , by the channel transmission speed, W , yields packet size, multiplying both sides of the above equation by W establishes the average number of bytes that can be transmitted by an average node for each unit of time of the relative deadline. In a load-balanced network of n nodes, the capacity of the network is nU byte-hops per unit of relative deadline (or equivalently, nUr_{av} byte-meters per unit of relative deadline, where r_{av} is the average distance between communication hops). Hence, the real-time capacity of the sensor network, denoted C_{RT} , is bounded by

$$C_{RT} < nr_{av} \left(1 + \frac{\alpha}{N} - \sqrt{1 + \left(\frac{\alpha}{N}\right)^2} \right) W \quad (22.5)$$

Some interesting observations are apparent. First, rewriting $1 + \alpha/N$ as $\sqrt{1 + 2\alpha/N + (\alpha/N)^2}$, observe that when N is large, the term $(\alpha/N)^2$ can be neglected leading to

$$C_{RT} < nr_{av} \left(\sqrt{1 + \frac{2\alpha}{N}} - 1 \right) W \quad (22.6)$$

We know from series expansion that for a small x , the term $\sqrt{1+x}$ is approximately equal to $1 + x/2$. Hence, substituting for the square root in Equation 22.6 when N is large, we get

$$C_{RT} < \frac{nr_{av}\alpha}{N} W \quad (22.7)$$

The above expression is the first known bound that establishes real-time capacity limits as a function of network size and node radio transmission speed. The bound errs on the safe side. It may be possible to communicate more traffic in a timely manner than what is predicted above.

22.6 Concluding Remarks

In this chapter, some basic principles for providing real-time wireless communication were addressed followed by an overview of real-time wireless protocols able to provide soft real-time guarantees. Finally, the notion of real-time capacity was introduced to quantify the capacity of a sensor network to deliver

information by deadlines. It is worth noting that while several protocols are available to support single-hop real-time wireless communication, the case of *ad hoc* real-time multihop still represents a challenging task and an open research problem the real-time and network communities are looking at. To the best of our knowledge, SPEED and RAP are the first delay-sensitive protocols devised for sensor networks, but they are still subject to unpredictable delays introduced at the MAC layer (due to the random backoff). Some interesting works exist that make the assumption of having globally synchronized nodes (see Refs. 8 and 25); they can avoid packet conflicts on the wireless medium by relying on a static and slotted packet schedule (TDMA), but they do not handle efficiently mobility or joining/leaving of nodes.

References

1. I. Aad and C. Castelluccia. Differentiation mechanisms for IEEE 802.11. In *IEEE Infocom*, Anchorage, Alaska, April 2001.
2. T. Abdelzaher, S. Prabh, and R. Kiran. On real-time capacity of multihop wireless sensor networks. In *IEEE Real-Time Systems Symposium*, Lisbon, Portugal, December 2004.
3. T. Abdelzaher, J. Stankovic, S. Son, B. Blum, T. He, A. Wood, and C. Lu. A communication architecture and programming abstractions for real-time sensor networks. In *Workshop on Data Distribution for Real-Time Systems*, Providence, Rhode Island, May 2003.
4. T. Abdelzaher, G. Thaker, and P. Lardiery. A feasible region for meeting aperiodic end-to-end deadlines in resource pipelines. In *IEEE International Conference on Distributed Computing Systems*, Tokyo, Japan, December 2004.
5. T. P. Baker. Stack-based scheduling of real-time processes. *The Journal of Real-Time Systems*, 3(1): 67–100, 1991.
6. R. O. Baldwin, I. V. Nathaniel, J. Davis, and S. F. Midkiff. A real-time medium access control protocol for ad hoc wireless local area networks. *SIGMOBILE Mobile Computer Communication Review*, 3(2): 20–27, 1999.
7. B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, J. Stankovic, and S. Son. An entity maintenance and connection service for sensor networks. In *The 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, May 2003.
8. M. Caccamo, L. Zhang, L. Sha, and G. Buttazzo. An implicit prioritized access protocol for wireless sensor networks. In *IEEE RTSS*, December 2002.
9. T. L. Crenshaw, A. Tirumala, S. Hoke, and M. Caccamo. A robust implicit access protocol for real-time wireless collaboration. In *IEEE Proceedings of the ECRTS*, July 2005.
10. R. Cunningham and V. Cahill. Time bounded medium access control for ad hoc networks. In *Principles of Mobile Computing*, 2002.
11. J. Deng and R. S. Chang. A priority scheme for IEEE 802.11 DCF access method. *IEICE Transactions on Communications*, E82-B(1): 96–102, 1999.
12. T. He, J. Stankovic, C. Lu, and T. Abdelzaher. Speed: A stateless protocol for real-time communication in sensor networks. In *International Conference on Distributed Computing Systems*, Providence, Rhode Island, May 2003.
13. S. Jang-Ping, L. Chi-Hsun, W. Shih-Lin, and T. Yu-Chee. A priority MAC protocol to support real-time traffic in ad hoc networks. *Wireless Networking*, 10(1): 61–69, 2004.
14. S. Kumar, V. S. Raghavan, and J. Deng. Medium access control protocols for ad-hoc wireless networks: A survey. *Elsevier Ad-Hoc Networks Journal*, 4(3): 326–358, May 2006.
15. J. Li, C. Blake, D. S. J. De Couto, H. I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *ACM MobiCom*, 2001.
16. C. H. Lin. *A multihop adaptive mobile multimedia network: Architecture and protocols*. PhD thesis, University of California at Los Angeles, 1996.
17. C. R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal of Selected Areas in Communications*, 15(7): 1265–1275, 1997.

18. C. R. Lin and M. Gerla. Real-time support in multihop wireless networks. *Wireless Networks*, 5(2): 125–135, March 1999.
19. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1): 40–61, 1973.
20. J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, New Jersey, USA, 2000.
21. C. Lu, B. Blum, T. Abdelzaher, J. Stankovic, and T. He. Rap: A real-time communication architecture for large-scale wireless sensor networks. In *Real-Time Technology and Applications Symposium*, San Jose, CA, September 2002.
22. R. Mangharam and R. Rajkumar. Max: A maximal transmission concurrency MAC for wireless networks with regular structure. In *IEEE Broadnets*, October 2006.
23. J. P. Monks, V. Bharghavan, and W. Hwu. A power controlled multiple access protocol for wireless packet networks. In *Proceedings of IEEE Infocom*, 2001.
24. A. Pal, A. Dogan, and F. Ozguner. MAC layer protocols for real-time traffic in ad-hoc wireless networks. In *ICPP*, 2002.
25. A. Rowe, R. Mangharam, and R. Rajkumar. Rt-link: A time-synchronized link protocol for energy-constrained multi-hop wireless networks. In *IEEE SEACON*, September 2006.
26. J. Sobrinho and A. Krishnakumar. Real-time traffic over the IEEE 802.11 medium access control layer. *Bell Labs Technical Journal*, 1(2): 172–187, 1996.
27. J. Sobrinho and A. Krishnakumar. Quality-of-service in ad hoc carrier sense multiple access networks. *IEEE Journal on Selected Areas in Communications*, 17(8): 1353–1368, August 1999.
28. F. A. Tobagi and L. Kleinrock. Packet switching in radio channels: Part II—the hidden terminal problem in carrier sense multiple-access and the busy-tone solution. *IEEE Transactions on Communications*, 23(12): 1417–1433, 1975.
29. N. Vaidya, N. Malpani, and J. L. Welch. Leader election algorithms in mobile ad hoc networks. In *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, August 2000.
30. A. J. Viterbi. *CDMA: Principles of Spread Spectrum Communication*. Prentice-Hall, April 1995.
31. Q. Wang, X. Liu, W. Chen, W. He, and M. Caccamo. Building robust wireless LAN for industrial control with DSSS-CDMA cellphone network paradigm. In *IEEE RTSS*, December 2005.
32. X. Yang and N. Vaidya. Priority scheduling in wireless ad hoc networks. In *ACM MobiHoc*, 2002.
33. J. Zatoński, T. Jurdziński, and M. Kutylowski. Efficient algorithms for leader election in radio networks. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002.

23

Programming and Virtualization of Distributed Multitasking Sensor Networks

23.1	Introduction	23-1
	Motivation and Challenge • The SN workBench	
23.2	The SNAFU Programming Language	23-4
	Cycle-Safe Iteration • Simulated Recursion • Let Assignments • Runtime Types • SNAFU Compilation	
23.3	Sensorium Task Execution Plan	23-7
23.4	The Sensorium Service Dispatcher	23-9
	Resource Management • Scheduling and Dispatch of STEP Programs	
23.5	Sensorium Execution Environments	23-12
	Implementation Overview • Sensorium Task Execution Plan Admission • STEP Interpretation • STEP Node Evaluation • STEP Program/Node Removal	
23.6	Putting It All Together	23-16
23.7	Related Work	23-18
23.8	Conclusion	23-19
	Catalytic Work	

Azer Bestavros

Boston University

Michael J. Ocean

Boston University

23.1 Introduction

We envision the emergence of general-purpose, well-provisioned sensor networks (SNs)—which we call “Sensoria”—that are embedded in (or overlayed atop) physical spaces, and whose use is *shared* among autonomous users of that space for independent and possibly conflicting missions. Our conception of a Sensorium stands in sharp contrast to the commonly adopted view of an embedded SN as a special-purpose infrastructure that serves a well-defined, fixed mission. The usefulness of a Sensorium will not be measured by how highly optimized its various protocols are, or by how efficiently its limited resources are being used, but rather by how *flexible* and *extensible* it is in supporting a wide range of applications. To that end, in this chapter, we overview and present a first-generation implementation of SNBENCH: a programming environment and associated runtime system that support the entire life cycle of programming sensing-oriented applications.

The SNBENCH abstracts a collection of dissimilar and disjoint resources into a shared virtual SN. The SNBENCH provides an accessible high-level programming language that enables users to write “macro-level”

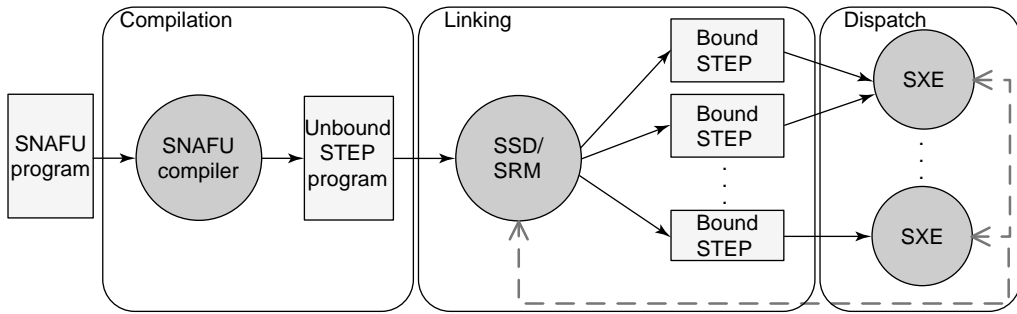


FIGURE 23.1 The SN program life cycle as enabled by the SNBENCH. Rectangles represent data, circles represent tasks/processes, and the broken lines represent control communication (i.e., dependency).

programs for their own virtual SN (i.e., programs are written at the scope of the SN rather than its individual components and specific details of the components or deployment need not be specified by the developer). To this end, SNBENCH provides execution environments and a runtime support infrastructure to provide each user a Virtual SN characterized by efficient automated program deployment, scheduling, resource management, and a truly extensible architecture.

The components of SNBENCH are analogous to those commonly found in traditional, stand-alone general-purpose computing environments (Figure 23.1). SNAFU (SensorNet Applications as Functional Units) is a high-level strongly typed functional language that supports stateful, temporal, and persistent computation. SNAFU is *compiled* into an intermediate, abstract representation of the processing graph, called a STEP (Sensorium Task Execution Plan). The STEP graph is then *linked* to the available Sensorium eXecution Environments (SXEs). A Sensorium Service Dispatcher (SSD) decomposes the STEP graph into a linked execution plan, *loading* STEP subgraphs to appropriate individual SXEs and binding those loaded subgraphs together with appropriate network protocols. The SSD may load many such programs onto a Sensorium simultaneously, taking advantage of programs' shared computation and dependencies to make more efficient use of sensing, computation, network, and storage resources.

23.1.1 Motivation and Challenge

To date, the emphasis of most research and development efforts targeting SNs has been on protocol refinements and algorithmic optimizations of how underlying resources in SNs are utilized to achieve a single static task or mission, subject to stringent constraints imposed by the typically impoverished resources of these SNs (e.g., battery lifetime, noisy radio communication, and limited memory). In many ways, an inherent assumption of most SN research is that *both* the SN infrastructure as well as the applications deployed on this infrastructure are owned and controlled by a *single* entity. This “closed system” mind-set allows the designers of such SNs to think of an SN as a “special-purpose” stand-alone system.

With the increased commoditization of sensing, computing, networking, and storage devices, a different model of SN infrastructures is emerging, whereby (1) the owner of the SN and its users may be different entities, and (2) the users of the SN may autonomously deploy independent applications that share the SN infrastructure.

To motivate this alternative “open-system” view of an SN, consider a public space such as an airport, shopping mall, museum, parking garage, subway transit system, among many other such examples. Clearly, there are many different constituents who may have an interest in monitoring these public spaces, using a variety of sensing modalities (e.g., video cameras, motion sensors, temperature sensors, and smoke detectors). In the case of a shopping mall, these constituents may include mall tenants, customers, mall security, local police, and fire departments. One alternative is for all these different constituents to overlay the public space they share with independent SN infrastructures—comprising separate sensors, actuators, processing and storage server nodes, and networks—each of which catering to the specific mission or

application of interest to each respective constituent. Clearly, this is neither efficient nor practical, and for many constituents may not be even feasible. Rather, it is reasonable to expect that such a shared SN infrastructure will be an integral part of the public space in which it is embedded, operated, and managed by an entity different from the entities interested in “programming” it for their own use.

Harnessing the power of such shared SN infrastructures will hinge on our ability to streamline the process whereby relatively unsophisticated “third parties” are able to rapidly develop and deploy their applications without having to understand or worry about the underlying, possibly complex “plumbing” in the SN infrastructure supporting their applications.

Today, programming SNs suffers from the same lack of organizing principles as did programming of stand-alone computers some 40 years ago. Primeval programming languages were expressive but unwieldy; software engineering technology improved with the development of new high-level programming languages that support more expressive and useful abstraction mechanisms for controlling computational processes, as well as through the wide adoption of common software development platforms that leverage these abstractions. For SNs, we believe that the same evolutionary path need not be (painfully) retraced, if proper abstractions, expressive languages, and software engineering platforms are developed in tandem with advances in lower-level SN technologies.

23.1.2 The SN workBench

In this chapter, we overview and present such a platform, which we call `SNBENCH` (SN workBench). The `SNBENCH` provides each user to access his or her own “private” SN, easily tasked with new programs and handling the scheduling, deployment, and management concerns transparently (i.e., offering each user an abstract, Virtual SN). `SNBENCH` embodies an accessible, flexible, and extensible programming platform and runtime system that support the entire life cycle of distributed sensory applications. Our primary focus is *not* on optimizing the various algorithms or protocols supporting an *a priori* specified application or mission, but rather on providing *flexibility* and *extensibility* for the rapid development and deployment of a wide range of applications.

While `SNBENCH` is conceived (by design) to be oblivious to the sensing modalities in an SN, in our current implementation we are naturally focusing on the particular SN infrastructure in our laboratories, which we call the “Sensorium,” consisting of a web of wired and wireless networked video cameras and motes [1] spanning several rooms, processing units, and a terabyte database, all of which are managed together to be able to execute *ad hoc* programs (or queries) specified over the Sensorium’s monitored spaces. In many ways, we view our Sensorium as prototypical of emerging SN infrastructures whose use is *shared* among autonomous users with independent and possibly conflicting missions.

Programmatic access to the Sensorium is provided via a high-level task-centric programming language that is compiled, distributed, scheduled, and monitored by `SNBENCH`. A `SNBENCH` program is specified in the SNAFU language. SNAFU is a strongly typed functional-style programming language that serves as an accessible, high-level language for developers to glue together the functionalities of sensors, actuators, processing, storage, and networking units to create stateful, temporal, and persistent programs. In effect, `SNBENCH` presents programmers with an abstract, “single-system” view of an SN, in the sense that SNAFU code is written for the Sensorium as a whole, and not separately for each of its various subsystems.

A SNAFU program is compiled into a STEP, which takes the form of a directed acyclic graph (DAG) in which the nodes are the sampling and computation operations required to execute the program. An execution plan may consist of nodes that are either bound (i.e., must be deployed on a specific resource) or unbound (i.e., free to be placed wherever sufficient resources may be found). A STEP is analogous to a program that has not been linked and has a straightforward serialized XML representation.

The SSD is responsible for the linking and scheduling of a STEP onto the SN infrastructure. In general, the SSD solves a restricted form of a graph embedding problem, finding resources capable of supporting subgraphs of the STEP graph and allocating them as appropriate. The SSD optimizes the use of resources and identifies common subexpressions across already deployed execution plans such that computation resources may be shared or reused. The SSD relies heavily on the Sensorium Resource Manager (SRM),

a registrar of computing and sensing resources available in the system at present. The SSD decomposes a single “unbound” STEP graph into several smaller “bound” STEP graphs and dispatches those graphs onto available Sensorium functional units.

Each Sensorium functional unit features an SXE, which is a runtime system that realizes the abstract functionalities presented to SNAFU programmers as basic building blocks. Such realizations may rely on native code (e.g., device drivers or local libraries) or may entail the retrieval of programmer-supplied code from remote repositories. An SXE interprets and executes partial STEP graphs that have been delegated to it by the SSD. Such a partial STEP graph may involve computing, sensing, storage, or communication with other SXEs.

The SNBENCH provides:

- I. A high-level, functional-style network programming language (SNAFU) and a compiler to produce a STEP as output. SNAFU exists to ease development, to provide type safety, and to allow the developer a clear shift toward programming the network rather than the nodes.
- II. The Sensorium Task Execution Plans (or STEPs), a task-oriented, cycle-safe language used to assign computations to individual execution environments within the Sensorium. STEPs chain together core capabilities supported by the runtime environment of the SN to form a logical path of execution. Formally, a STEP is a DAG of the computations requested of a single computing element. Given a program’s representation as a DAG, parallelization and other optimizations become tractable. Note that although a STEP is represented as a DAG we do support a limited iteration construct that can be used to emulate recursion.
- III. Various runtime support components that monitor SN resources and schedule newly submitted STEP programs to available SN resources. The SRM is responsible for maintaining a current snapshot of the available resources in the SN, while the SSD is responsible for accepting new STEP programs for the SN and scheduling the tasks of the STEP to available resources. Optimally, the scheduler must identify tasks within the new STEP that match currently deployed tasks and reuse them as appropriate and find a partitioning of the STEP program that can be deployed to physical resources.
- IV. A runtime virtualization environment for heterogeneous computing and sensing resources. The SXE is a common runtime that provides sensing or computing elements of the SN the ability to interpret and execute dynamically assigned task execution plans. In part, the execution environment hides irrelevant differences between various physical components and exposes a unified virtualized interface to their unique capabilities. The sensor execution environment also allows the resource to be dynamically loaded with new functionalities via the network.

In this chapter, we provide a bird’s eye view—as well as some implementation details—of the various SNBENCH components. We frame the components with motivating examples in the domain of networked computer vision applications. Further motivation and high-level details may be found in Refs. 2 and 3.

23.2 The SNAFU Programming Language

The current SNBENCH programming interface is a high-level, network-oriented language called SNAFU. SNAFU is a strongly typed programming language designed to specify the interdependence of sensors, computational resources, and persistent state that comprises a Sensorium application.

SNAFU programs are written in a simple functional style; all SNAFU program terms are expressions whose evaluation produces values. Consider the following example SNAFU program that inspects a single video frame and returns the number of faces detected in the frame.

```
facecount ( snapshot ( sensor ( "s02" , "cam0" ) ) ) )
```

A novice user will specify an SNAFU program written for the SN (not the individual nodes), and the infrastructure will transparently handle translation, resource allocation, and dissemination of the

program to involved nodes of the SN. SNAFU is provided as a convenient and accessible interface to the STEP language. In future iterations of STEP, we plan for STEP and SNAFU to diverge to a greater extent, as we wish to use the SNAFU interface as a restriction on the power of STEP.*

SNAFU’s direct benefit lies in its type-checking engine and STEP compiler. SNAFU programs are implicitly typed and disallow explicit data type annotation. The type engine statically type-checks SNAFU programs to identify typing errors and inserts permissible type promotions. SNAFU forbids explicit recursion (including transitive cases) instead providing an iteration construct that enables “simulated” recursion as described in the next section.

23.2.1 Cycle-Safe Iteration

SNAFU programs do not allow traditional recursion but instead provide iterative execution through the “trigger” construct. A trigger indicates that an expression should be evaluated more than once, given some predicate expression. The repetition supported in SNAFU can be divided into two larger classes: terminating and persistent.

A terminating `trigger(x, y)` will repeatedly evaluate `x` until it is evaluated to be true at which point `y` will be evaluated and the value of `y` is returned as the value of the trigger expression. The trigger will not execute `x` again after it has evaluated to be true (i.e., it will terminate).

Alternatively, persistent triggers continue to reevaluate their predicate “indefinitely.” We provide two persistent triggers; the `level_trigger(x, y)` will continually evaluate `x` and every time `x` evaluates to true, `y` is reevaluated. The “edge-trigger” construct `edge_trigger(x, y)` will continually evaluate `x` and when `x` transitions to become true from false (or if it initially evaluates to true), `y` is evaluated. The trigger’s value is initially NIL and is updated to the value of `y` every time `y` is evaluated. The SNAFU trigger example below runs “indefinitely,” repeatedly counting the number of faces found at the specified sensor.

```
level_trigger(true,
  facecount(snapshot(sensor("s02", "cam0")))
)
```

In fact, persistent triggers typically live for a configuration-specific period of time (e.g., 1 hour). To terminate a persistent trigger based on some runtime predication, the programmer may wrap the persistent trigger within a terminating trigger. Alternatively, a persistent trigger may be wrapped in a “flow-type” function allowing the programmer to specify a particular temporal persistence policy.

23.2.2 Simulated Recursion

For some tasks, the body of a persistent trigger expression may need to make use of its own prior evaluations (i.e., utilizing prior state, similar to tail recursion). SNAFU supports this via the `LAST_TRIGGER_EVAL` token, which acts as a nonblocking read of the closest enclosing parent trigger’s value.

Naturally, the results of persistent triggers may be used by other expressions as data sources for some other task. As the values of persistent triggers are transient and the temporal needs of the dependent expression may vary, we provide three different `read` functions that allow the programmer to specify a synchronization rule for the read of the trigger with respect to the trigger’s own evaluation.[†]

Specifying a “non-blocking” read to a trigger requests the read immediately returns the result of the last completed evaluation of the trigger’s target expression, blocking if and only if the target expression has never completed an evaluation. A “blocking” read waits until the currently ongoing evaluation of

*Ref. 4 shows that STEP is Turing complete; hence, we intend to use SNAFU to restrict the programmer from the full potential (and pitfalls) of STEP.

[†]The value of a persistent trigger should always be read using one of these primitives. A program term containing an expression that directly accesses the value of a persistent trigger will be rejected by the SNAFU-type engine. In contrast, terminating triggers have an implicit blocking semantic and should not be wrapped by read primitives.

the target expression completes, then returns that value. Finally, a “fresh” read waits for a complete reevaluation of the trigger’s predicate and target expressions before returning a value.

23.2.3 Let Assignments

SNAFU provides the ability to bind a value to a recurring symbol to either some persistent value (constant) or commonly occurring subexpression (macro).

The `letconst` directive assigns a constant term or expression to a symbol such that the value of an expression is evaluated only once (when first encountered). All further occurrences of the symbol are assigned the computed value and will not be evaluated. In the following example, the resolution of the sensor `cam1` will only be performed once, at the first instance of `cam1` in `Z`. All other instances of `cam1` in `Z` will refer to this same sensor resolution request.

```
letconst cam1 = sensor(ANY,IMAGE) in Z
```

Alternatively, symbols may also be used as a shorthand to represent longer (sub)expressions, each instance of which is to be independently evaluated (i.e., macros). The `leteach` binding, “`leteach x = y in z`,” replaces every occurrence of `x` in `z` with an independently evaluated instance of the expression `y`. Note in the example below every instance of `cam2` in `Z` may refer to a different sensor.

```
leteach cam2 = sensor(ANY,IMAGE) in Z
```

Finally, SNAFU allows a symbol to be assigned within the scope of a trigger such that the symbol obtains a new value once per each evaluation of the trigger (i.e., once per iteration). The `letonce` bindings, for use with trigger contexts, have the form “`letonce x = y in z`” and allows the expression `y` to be evaluated for the symbol `x` once per iteration of the containing trigger defined in `z`. Consider the use of the `letonce` binding in the following program fragment that continues from the previous example, the intent of which is to take an image sample from each camera once per iteration and then return the image sample that has the most faces in it in a given iteration.

```
letonce x = snapshot(cam1) in
letonce y = snapshot(cam2) in
  level_trigger(true,
    if-then-else(greater(facecount(x),facecount(y)),x,y)
  )
```

23.2.4 Runtime Types

SNAFU allows program terms to be wrapped by “flow-type” functions. Flow types provide explicit constraints for program deployment and execution in the Sensorium, providing type information for the control flow as well as the data flow. As examples, the programmer may require a particular periodicity for a trigger term’s evaluation, or may wish to ensure that some computations are only assigned to a trusted set of resources. Other example flow types are given throughout this chapter; however, an exhaustive account of the nature (and semantic) of these flow types is beyond the scope of this chapter. We refer the reader to our work on using strong-typing for the compositional analysis of safety properties of networking applications [5] for some insight as to how flow types could be type-checked for safety.

23.2.5 SNAFU Compilation

SNAFU has been designed to ensure that the abstract syntax tree (AST) of a SNAFU program maps to a task dependency diagram in the form of a DAG with a single root.* Nodes in the DAG represent values, sensors,

*Although the SNAFU AST is a tree, the execution semantic of SNAFU is actually a graph. Consider the `letconst` binding that allows a single node to have multiple parents.

or tasks while edges represent data communication/dependency between nodes. The graph is evaluated by lower nodes executing (using their children as input) and producing values to their parents such that values percolate up toward the root of the graph from the leaves. The SNAFU compiler transforms the AST of the SNAFU program into such a representation, which we call a “Sensorium Task Execution Plan” or STEP. The terms and expressions of SNAFU have analogous constructs (nodes) in STEP or clear encoding in the structure of the STEP graph. For example, the single-evaluation `letconst` construct is a directive to link a single subtree onto several parents, and the `if-then-else` function refers to the placement of a conditional (`cond`) STEP node.

23.3 Sensorium Task Execution Plan

A STEP is a specification of a Sensorium program in terms of its fundamental sensing, computing, and communication requirements. A STEP is serialized as an XML document that encodes the DAG of the explicit task dependency (evaluation strategy) of a Sensorium program. The STEP language is used to describe (1) whole programs written in the scope of the entire Sensorium (i.e., programs compiled from SNAFU that are either largely or entirely agnostic as to the specific resources on which their constituent operations are hosted) and (2) (sub)programs to be executed by specific individual sensor execution environments to achieve some larger programmatic task (i.e., to task the individual Sensorium resources in support of (1)).

STEP is the preferred target language for the compilation of SNAFU (and other future languages) and as such we refer to STEP as the Sensorium “assembly language” (i.e., STEP is our “lowest-level” Sensorium programming language). That said, STEP is a relatively high-level interpreted language.* We note that although STEP is the preferred target language for SNAFU compilation, for some constrained resources, running a STEP interpreter may not be desirable. In such situations, we rely on gateway nodes to interpret STEP and relay requests on the node’s behalf. We use this approach within our current Sensorium to integrate Berkley Motes for temperature sensing. In a future generation of the SNBENCH, we may consider providing an SNAFU compiler that produces targets more suitable for constrained devices (e.g., C99 and Intel asm).

Individual tasks within a STEP (i.e., nodes within the STEP graph) may be “bound” to a particular SN resource (e.g., some sensor sampling operation that must be performed at a specific location), while others are “unbound” and thus free to be placed anywhere in the SN where requisite resources are available. In general, SNAFU compilation results in the creation of an “unbound” STEP—a STEP graph containing one or more “unbound” nodes. Unbound STEPs are analogous to unlinked binaries insofar as they can not be executed until required resources are resolved. Unbound STEPs are posted to an SSD, the entity responsible for allocating resources and dispatching STEP programs. Given the state of the available system resources and the resources required by the nodes comprising this graph, the SSD will fragment the unbound STEP graph into several smaller bound STEP subgraphs.

In the remainder of this section, we describe the “classes” of tasks (nodes) that are supported by the STEP programming language and convey with broad strokes the runtime semantic they convey. The reader should note a correlation between the node classes presented in this section and the presentation of the SNAFU semantic. Indeed, these constructs are a direct encoding of the functionalities presented in that section.

We frame this discussion within the context of the example STEP program given in Figure 23.2. This STEP program is the result of compiling the following SNAFU snippet, which returns the maximum number of faces detected/observed from any one of the two cameras mounted on `s05(.sensorium.bu.edu)`.

*Although STEP programs are technically human-readable, their lack of type-checking and XML representation (including attributes, which the user may have no interest or business assigning) make direct program composition in the STEP language inadvisable at best.


```

<step id="202219@s00.sensorium.bu.edu">
  <exp opcode="max" id="abcd">
    <flowtype name="persist" value="Dec 25 23:59:59 EDT 2005" />
    <exp opcode="facecount" id="bcde">
      <exp opcode="snapshot" id="cdef"><value id="defg">
        <sensor type="snbench/image" uri=
          "http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/1"/>
        </value></exp></exp>
      <exp opcode="facecount" id="efgh">
        <exp opcode="snapshot" id="fghi"><value id="ghij">
          <sensor type="snbench/image" uri=
            "http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/2"/>
          </value></exp></exp>
        </exp>
      </step>
    </exp>
  </step>

```

FIGURE 23.2 An unbound STEP program for computing the maximum number of faces detected from one of the two cameras mounted on s05.

```

max( facecount( snapshot( sensor( "s05", "cam1" ) ) ) ,
    facecount( snapshot( sensor( "s05", "cam2" ) ) ) )

```

step node: The `step` node is the root node of a STEP and contains the entire STEP program. The node has an `id` attribute that is a globally uniquely identifier (GUID) generated by the SNAFU compiler, uniquely identifying this program (and all nodes of this program) from other programs. The immediate child of the `step` node is the true root node of the program.

exp nodes: An `exp` (*expression*) node conveys a single computing, sensing, storage, or actuator function to be performed by some SXE. An expression node has an `opcode` attribute that identifies, which function should be performed, and the immediate children of the expression node are the arguments to the function. For example, opcodes include addition, string concatenation, image capture, image manipulation, and detecting motion. Opcodes are core library operations distributed with the SXE. If an SXE does not have the opcode required, the `jar` attribute may specify a URL where a Java byte-code implementation of this opcode can be found. Similarly, the `source` attribute may be used to specify the location of the Java source code for this opcode.*

cond nodes: A `cond` (*conditional*) node has three children: an expression that evaluates to a Boolean value (i.e., a condition), an expression that will be evaluated if the condition is true, and an expression that will be evaluated if the condition is false. The *conditional* node has an evaluation semantic that ensures the first child (subtree) is evaluated first and, depending on the result, only the second or the third child will be evaluated.

sensor nodes: A `sensor` node conveys a specific physical sensor within the SN and is used to provide the sensor as an argument to some expression node. In the example given (Figure 23.2), the two snapshot expression nodes each have a `sensor` node as a child to specify on which particular image sensor they will operate. Sensor nodes may have a `uri` attribute to indicate where the sensor can be found and will have a `type` attribute to indicate the type of input device that this node provides (e.g., image, video, audio, and temperature). Sensor nodes only appear as leaves in a STEP graph. A sensor node requires additional processing on the SSD to resolve and reserve “wildcard” sensors (i.e., when the `uri` of the sensor is omitted).

trigger, edge_trigger, and level_trigger nodes: All `trigger` nodes specify that their descendants are subject to iteration as indicated by the corresponding *trigger* construct (explained in Section 23.2). Trigger nodes have two children: the predicate and the body. A trigger may also have zero or

*The dynamic migration of opcode implementations raises clear security/trust concerns. We expect the SXE owner will maintain a white-list of trusted hosts from which opcodes can be safely retrieved.

more `flowtype` nodes to convey the runtime/deployment QoS constraints of this trigger. Related to trigger functions, **read nodes** may appear as the parent of a `trigger` node to explicitly request specific temporal access to the values produced by that trigger. The `read` node's `opcode` attribute determines whether the trigger will be read via a “blocking,” “nonblocking,” or “fresh” semantic (also described in Section 23.2).

flowtype nodes: The `flowtype` nodes are used to encode runtime security, performance, and persistence constraints. These nodes appear as children of the nodes that they constrain.

socket nodes: `socket` nodes may be inserted into unbound STEP DAGs by the SSD during the binding (scheduling) process to allow distribution of a program's evaluation across SXEs. The `socket` node connects the computation graph from one SXE to another SXE across the network. A `socket` node has a `role` attribute, which is set to either `sender` or `receiver`. A `sender` node takes the value passed up by a child node and sends it across the network to another SXE specified by the node's `peeruri` attribute. Assuming the peer SXE is hosting a corresponding `receiver` node, that `receiver` node sends this value along to its parent node allowing a STEP “edge” to span SXEs. A `protocol` attribute specifies, which specific communication protocol should be used for data transfer (e.g., HTTP/1.1 pull and HTTP/1.1 push).

splice nodes: A `splice` node is used as a pointer to another node, allowing the encoding of graphs within the tree-centric XML. The `splice` node indicates that the parent of the splice node should have an edge that is connected to the “target” of the splice node (the splice node has a `target` attribute that specifies an id of another existing node). The splice node only exists when a STEP is serialized as XML, and when deserialized, the edge is connected to the target node. Splice nodes may occur within a compiled STEP graph if some node/subgraph has multiple parents (e.g., the “let” binding provided in SNAFU) or a splice may occur as a result of computational reuse allowing one STEP program to be grafted onto another. Splice nodes allow the SSD to reuse components of previously deployed STEP graph within newly deployed STEP graphs by replacing a common subgraphs in the new STEP program with a splice node.

const nodes: The `const` node class is used to block the propagation of “clear” events during evaluation, in effect preventing the reevaluation of its descendants (e.g., to support a `letonce` binding). A `const` node will have exactly one child, namely the subgraph that we wish to limit to a single evaluation.

23.4 The Sensorium Service Dispatcher

The SSD is the administrative authority of, and single interface to each “local-area” Sensorium. The SSD is responsible for allocating available concrete Sensorium resources to process STEP (sub)programs (i.e., scheduling) and dispatching STEP (sub)programs to those resources. Each SSD is tightly coupled with an SRM that maintains the state and availability of the resources within the Sensorium.

The current SSD/SRM implementation is Java based and utilizes HTTP as its primary communication model; an HTTP server provides an interface to managed resources and end users alike. Communications are XML formatted and, for end users, responses are transformed into viewable interactive web page by extensible Stylesheet Language Transformation (XSLT). The HTTP namespace is leveraged to provide a natural interface to the hierarchical data and functionality offered by the SSD.

The SSD/SRM has two primary directives: resource management and STEP scheduling and dispatch. Both are described below.

23.4.1 Resource Management

The SRM monitors the state of the resources of its local Sensorium and reports changes to the SSD. Each computing or sensing component in the managed domain that hosts an SXE sends a heartbeat to its SRM, the result of which is used to populate a directory (hashtable) of all known SXEs and their attached sensory resources. The heartbeat includes the SXE's uptime, sensing capabilities, and a scaled score indicating available computing capacity. Should an SXE miss a configurable number of heartbeats, or the SXE report an unexpected computing capacity change without notification of a “STEP complete,” the SRM assumes the

SXE has failed or restarted and informs the SSD of the change. The SRM's knowledge of the state of the managed Sensorium is essential to the SSD's correct operation in deploying and maintaining STEP programs.

When an SXE leaves the Sensorium (e.g., SXE shutdown, reboot, or graceful exit), there may be impact to one or more running STEP programs as multiple STEP applications may be dependent on a single STEP node or resource. When the SRM detects that an SXE has left the Sensorium, the SSD will treat all STEP tasks deployed on that SXE resource as a new STEP program submission and try to reassign it (in part or in whole) to other available resources.* Updated socket nodes are sent to redirect those SXEs hosting sockets connected to the exiting SXE.† If no sufficient resources can be found to consume the STEP nodes that had been hosted by the old resource, we must traverse the dependency graph and remove all impacted STEP nodes (i.e., programs).

23.4.2 Scheduling and Dispatch of STEP Programs

The SSD maintains a master, nonexecutable STEP graph consisting of all STEP programs currently being executed by the local Sensorium. Each STEP node in this graph is tagged with the GUID of the SXE on which that STEP node is deployed, such that this master STEP graph indicates what tasks are deployed in the local Sensorium and on to what resources.

When a STEP program is submitted to the SSD, the SSD must locate available resources for the newly submitted STEP graph, fragmenting the newly submitted STEP into several smaller STEPs that can be accommodated by available resources. We approach this task as a series of modules that process the unbound STEP program (Figure 23.3). We present our approach for each of these modules, yet emphasize the benefit of this modular approach is that any module may be replaced with a different or improved algorithm to achieve the same goal.

Code reuse: In this generation of the SNBENCH, we assume that the reuse of computation is paramount. When a new STEP graph is posted, the SSD first tries to ensure that the scheduling of unbound expression nodes does not result in unnecessary instances of new tasks. Unfortunately, checking for all instances of “functionally equivalent” tasks in a language as expressive as STEP is NP-hard.

Indeed, as many sensing functionalities will be dependent on the use of fresh data, our current code reuse algorithm is intentionally conservative to avoid the reuse of stale data. Unless explicitly specified with flow types, the SSD will only reuse nodes that are “temporally compatible.”‡ Thus, all new nodes that match already deployed nodes are replaced with splices (i.e., pointers) to the previously dispatched nodes.§ Regardless of how the code-reuse module is implemented, after it is complete the “new” computation cost of the submitted STEP graph should be reduced.

Admission control: The SSD must deny admission if any bound STEP node refers to an unavailable resource or if the remaining resource cost exceeds total available resources. The SSD first iterates over all bound nodes of the STEP graph to ensure that requested SXEs are known by the SRM and have available computing resources to consume these computations (including available sensors where sensors are prerequisites for a computation). Once complete, the SSD ensures that the total free resources in the Sensorium are sufficient to accommodate the total cost of the remaining unbound nodes.

Graph partitioning: The SSD must bind all unbound nodes in the STEP graph to specific resources (Figure 23.4), a task analogous to a graph partitioning where each partition represents deployment on some SXE (i.e., physical resource) the goal of minimizing the total cost of edges between partitions (i.e., minimize induced communication cost between SXEs). Fortunately, the computations represented

*STEP nodes that are submitted to the SSD prebound to some specific resource cannot be migrated to another SXE and therefore must be terminated.

†Synchronization issues abound when this occurs, so a “reset” is sent to all nodes involved to restart communication in this event.

‡At present, temporal compatibility is ensured by reusing only identical, trigger-rooted subexpressions in the *local* Sensorium (giving us a tractable problem).

§There are certainly instances in which such blind bias toward computational reuse will result in a communication penalty that outweighs the benefit of code reuse, however, have not accounted for this in our current SSD iteration.

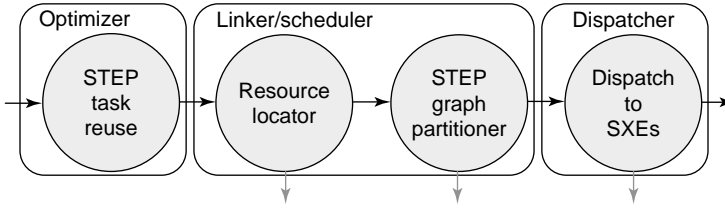


FIGURE 23.3 The SSD scheduling and dispatch process for STEP submission. Circles represent modules, while downward arrows indicate these modules may reject a STEP program owing to insufficient resources.

```

<step id="202219@s00.sensorium.bu.edu">
  <exp opcode="max" id="abcd" bindto="http://c02.sensorium.bu.edu:8080">
    <flowtype name="persist" value="Dec 25 23:59:59 EDT 2005" />
    <exp opcode="facecount" id="bcde" bindto="http://c02.sensorium.bu.edu:8080">
      <socket id="facel:in" protocol="POST" role="receiver" peerid="facel:out"
        peeruri="http://s05.sensorium.bu.edu:8080/snbench/sxe/node/facel:out/" /></exp>
    <exp opcode="facecount" id="efgh" bindto="http://c02.sensorium.bu.edu:8080">
      <socket id="face2:in" protocol="POST" role="receiver" peerid="facel:out"
        peeruri="http://s05.sensorium.bu.edu:8080/snbench/sxe/node/face2:out/" /></exp></exp>
  </step>

<step id="200507230943:a">
  <socket id="face1:out" protocol="POST" role="sender" peerid="facel:in"
    peeruri="http://c02.sensorium.bu.edu:8080/snbench/sxe/node/facel:in/">
    <exp opcode="snapshot" id="cdef"><value id="defg">
      <sensor type="snbench/image" uri=
        "http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/1/" /></value></exp>
    </socket>
  </step>

<step id="200507230943:b">
  <socket id="face2:out" protocol="POST" role="sender" peerid="face2:in"
    peeruri="http://c02.sensorium.bu.edu:8080/snbench/sxe/node/face2:in/">
    <exp opcode="snapshot" id="fghi"><value id="ghij">
      <sensor type="snbench/image" uri=
        "http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/2/" /></value></exp>\\
    </socket>
  </step>

```

FIGURE 23.4 Our previous STEP program for computing the maximum number of faces detected from either of the two cameras mounted on s05. In this instance, the SSD has split the STEP graph into three STEP subgraphs for deployment on two separate SXEs, c02 and s05.

at each node have associated data types and that type information yields a bound on the “cost” of each edge. For example, if a STEP node returns an Image, the communication cost of spanning this edge across the two different physical resources (i.e., adding this edge to the cut) will be greater than cutting an edge of a node that produces an Integer value (Figure 23.5).

Our initial graph partitioning algorithm makes only a nominal attempt to reduce communication cost. The procedure tries to assign the entire unbound region of the graph to any single available resource. Failing that, the unbound region of the graph is recursively split into smaller subgraphs, trying to find resources large enough to consume the “whole” parts.

Our second generation partitioning algorithm uses a relaxed form of spreading metrics [6] to produce partitions. A spreading metric defines a geometric embedding of a graph where a length is assigned to every edge in the graph such that nodes connected via inexpensive edges are mapped to be geometrically close, while nodes across expensive edges are physically “spread apart” from each other.

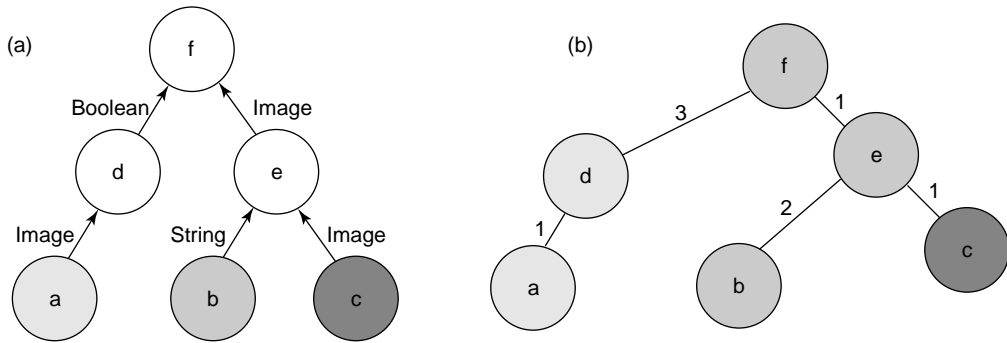


FIGURE 23.5 Generating colored partitions in a STEP graph. Coloring nodes is analogous to assigning a task to a particular SXE. Uncolored nodes should be colored to minimize communication between SXEs (colors)—there is no communication cost when adjacent nodes are the same color.

The optimization detailed in Ref. 6 relies on a linear program to assign lengths to edges. Instead, we employ a “quick-and-dirty” approximation of the spreading metric in which weights and distances for edges are derived entirely from the type information of the nodes (Figure 23.5). Although this approximation will not yield partitions with the same bounds on minimizing the cut, our approach is favorable in running time and we can compute, offline, the minimum cut of the spreading metric to use as benchmark for comparison against our approximation algorithm. Again we point out that any graph-partitioning solution may replace our existing partitioning logic and are investigating some “off-the-shelf” solutions.

Dispatch: Once all STEP nodes are annotated with bindings (Figure 23.4), the SSD must generate the STEP subgraphs to dispatch to each individual resource. During this phase, the SSD inserts socket nodes to maintain the dataflow of the original STEP graph after the partitioning. As each SXE receives only a part of the larger computation (and sockets to SXEs with which it shares an edge) each is unaware of the larger task it helps to achieve.

To dispatch the STEP subgraphs, the SSD performs an HTTP post of the STEP to the SXE’s web server. If all SXEs respond to the dispatch with success, the SSD’s dispatch is complete and the STEP program is live. If not, all partial STEPs of the larger STEP that had been posted to SXEs before this failed partial STEP are deleted from those SXEs and the user must resubmit.*

23.5 Sensorium Execution Environments

The SXE is a runtime execution environment that provides its clients remote access to a participating host’s processing and sensing resources. An SXE receives XML-formatted STEPs and the SXE schedules and executes the tasks described. Indeed an SXE is a virtual machine (Figure 23.6) providing multiple users remote access to virtualized resources including sensing, processing, storage, and actuators via the STEP program abstraction.

The SXE communicates its capabilities and instantaneous resource availability to its local SRM, allowing the SSD to best utilize each SXE. Each SXE maintains a local STEP graph containing a composite of all the STEP graphs (and subgraphs) tasked to this node by the SSD. In this section, we describe the essential functionalities of the SXE and our current implementation of these functionalities. As is the case with the

*We do not attempt to reoptimize the Sensorium’s global STEP graph (i.e., all computations on the current Sensorium) when a new STEP is submitted. It is possible that a better, globally optimal assignment may exist by reassigning nodes across the global STEP graph; however, we expect the computational cost will far outweigh the benefit. At present, we do not intend to move computations once they have been initially assigned unless absolutely necessary (e.g., in the event of resource failures).

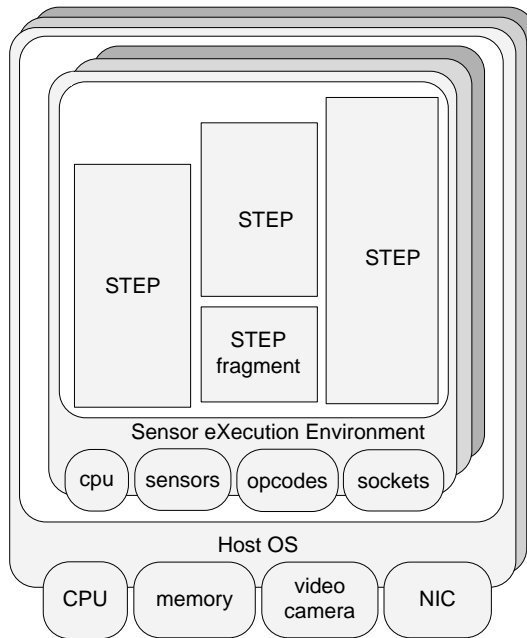


FIGURE 23.6 The SXE abstracts the devices provided by the OS, allowing clients to task these devices through STEP abstraction.

SSD, the SXE is also implemented with extensibility as a chief goal. We describe the SXE in terms of its necessary actions in support of the larger Sensorium via the STEP interface: STEP Program Admission, STEP Program Interpretation, STEP Node Evaluation, and STEP Program Removal.

23.5.1 Implementation Overview

Our current implementation of the SXE uses Java technologies, in particular Java 1.5 for the runtime, Java Media Framework (JMF) for sensor interaction, and Java-based NanoHTTPD for HTTP communications. The use of Java provides natural programming benefits of a strongly typed language with exception handling. Java provides straightforward mechanisms for runtime loading of dynamic functionality over the network (via jar files or dynamically compiled source code). In addition, Java provides the protection benefits of its sand-boxed runtime environments and a virtual machine profiling API. The GCJ suite allows us to compile Java programs into native byte-code if performance is an issue.

To provide access to the sensing resources of an SXE, all physical sensors are abstracted as generic sensors (data sources) with specific functionalities implemented via classes on top of the generic sensor (e.g., ImageSensor and AudioSensor).

The size of the jar file for the execution environment (SXE) containing all basic functionality including execution plan interpretation, evaluation, web server, and client is about 200 k uncompressed. We expect the SXE to be deployed on low-end desktop machines (Pentium Pro) and have not attempted to port to microdevices at present. Instead, we have implemented opcodes that act as gateways to communicate with restricted devices. When the participating SXE host also provides video-sensing functionality, the system requirements are increased by those of the JMF.

The SXE's primary mode of communication is via HTTP, acting as both a server and a client, as appropriate. The SSD communicates with constituent SXEs via their HTTP interfaces and each SXE utilizes an HTTP client to communicate with the SSD, SRM, and other SXEs. Each SXE may also utilize other communication protocols to communicate with nonstandard SXEs or non-SXE Sensorium participants (e.g., motes and IP video cameras). Data transfer between SNBENCH components is almost exclusively XML

formatted, including Base64/MIME encoding of binary data. The SXE sends an XML-structured heartbeat to the the SRM via an HTTP post from the SXE to the SSD. STEP graphs are uploaded to an SXE via HTTP POST of an XML object.

23.5.2 Sensorium Task Execution Plan Admission

When a new STEP graph is posted to an SXE via the SSD, the new tasks to be executed may be independent of or dependent on previously deployed tasks. Within a newly posted STEP graph, the SSD may embed “splice nodes” in the new STEP graph specifying edges that are to be spliced onto previously deployed STEP nodes (i.e., for task reuse) or nodes that should replace previously deployed STEP nodes with new nodes (task replacement).

1. *Task reuse*: A newly posted STEP graph may contain one or more “splice” nodes with target IDs that point to previously deployed STEP nodes, indicating some new computations will reuse the results computed by existing tasks. Although the splice is specified by the SSD through its seemingly omniscient view of the local Sensorium, each SXE maintains local scheduling control to avoid race/starvation issues.*
2. *Task replacement*: If a new STEP graph includes nonsplice STEP nodes with the same (unique) IDs as nodes already deployed, this indicates these new nodes should replace the existing nodes of the same ID. The replacement operation may result in either removal or preservation of children (dependencies) of the original node, while parent nodes are unaffected (although those may be modified through iterative replacement).†

23.5.3 STEP Interpretation

Recall a STEP is a DAG in which data propagate up through the edges from the leaves toward the root. Tasks appearing higher in the STEP graph are not be able to be executed until their children have been evaluated (i.e., their arguments are available). Likewise, the need for a node to be executed is sent down from a root (parents need their children before they can execute; however, once executed they do not necessarily need to be executed again).

The SXE’s local STEP graph may have several roots, as its graph may be the confluence of several independent STEP subprograms; however, there is not necessarily a one-to-one mapping between the number of STEP graphs posted and the number of roots in the local STEP graph.

Each STEP node may be in one of the four possible states: ready, running, evaluated, and blocked (Figure 23.7a). The SXE’s role in interpreting a STEP program consists of (1) maintaining and updating the control flow of a STEP graph, advancing them through their state transitions (described in this section) and (2) the actual execution of STEP nodes that are in the “running” state to enable the dataflow of a STEP graph (described in the next section).

The SXE interprets its current STEP nodes by continually iterating over all nodes and checking if they are “ready” to be evaluated. A generic node is determined to be ready to be evaluated if it (1) is wanted by a parent node, (2) has fresh input from all immediate children, and (3) has not been already executed (this can be reset by a parent node to enable a node to run more than once as in the case of a node with a parent trigger).

Within our present implementation, all nodes are iterated over in a round-robin fashion to determine if they are “ready.” When nonexpression nodes are ready they are evaluated immediately, while expression nodes are placed in a separately serviced FIFO run-queue. This approach to evaluating STEP nodes is not unique. Indeed, the selection of which nodes to consider next amounts to a scheduling decision that may be constrained by QoS requirements, or other considerations (e.g., frame rates). In fact, any scheduling

*It may also be interesting to consider admission-control algorithms for determining when a new partial STEP DAG is eligible for splicing onto an existing STEP DAG. At present, the SXE takes a “the customer (SSD) is always right” policy toward admission control.

†Notice that such node replacements may cause synchronization difficulties when replacements involve nodes that communicate across multiple SXEs. In general, we limit our use of replacement to redirect communication nodes to a replacement SXE when another SXE has left the Sensorium.

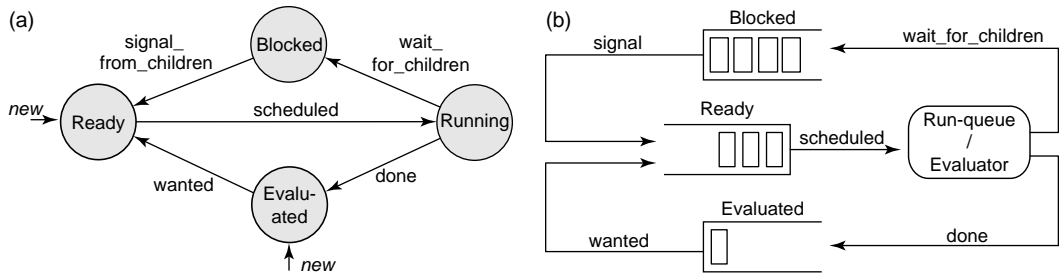


FIGURE 23.7 The SXE's STEP node evaluation state transition diagram. During evaluation, STEP expression nodes move between three buffers on the SXE.

algorithm may be swapped in to service the run-queue without adverse effect on graph evaluation or the dataflow (Figure 23.7b).

A ready node is evaluated by the evaluation function for its node class. When no STEP nodes are ready, the iterator sleeps until a new STEP graph is admitted or some other event (e.g., clock, network, and shutdown) wakes the iterator. Once a node has been evaluated, it produces a value that is pushed up the graph (possibly enabling parent nodes that are waiting for fresh input from their children). For some node classes, the ready function may be overridden to accommodate a nonstandard execution semantic. Persistent triggers, for example, are always wanted if they are orphans; however, if a trigger has a parent node this node is only wanted if its parent in turn wants it. We call the reader's attention to the subtle detail that, although persistent triggers should run indefinitely, they must not run asynchronously from a parent lest nested triggers would easily result in synchronization issues and race conditions.*

23.5.4 STEP Node Evaluation

Each STEP node class specifies its own evaluation function. The evaluation function of most node types maintains the runtime semantic of the STEP graph by updating any needed internal state (including the execution state flag) and passing up values received from children. The exception to this trivial evaluation model is the evaluation of STEP expression (*exp*) nodes. In all cases, the expectation is that the evaluation function for the node will produce a value to its parents.

The evaluation of *trigger* nodes requires updating the trigger's internal state, ensuring that first the predicate is evaluated and that the postcondition will be evaluated (and reevaluated) as per the trigger type and result of the predicate. Similarly, evaluation of a conditional (*cond*) node maintains state and determines whether the second or third branch should be evaluated and returned depending on the evaluation of the first branch. A *socket* node's evaluation sends or receives data along the socket, a *value* merely passes a serialized value up the tree and similarly *sensor* nodes are like *value* nodes in that they merely act as an argument to the immediate parent (*exp* node).

The evaluation of an expression (*exp*) node may take some time and as such the evaluation function for an expression node merely schedules the later execution of the expression node by a separate scheduler and execution thread (*exp* node evaluation should not block the entire resource). Expression nodes are tasks, analogous to the opcodes of the STEP programming language. These nodes are calls to fundamental operations supported by the SXEs (e.g., addition, string concatenation, and image manipulation), yet the opcode implementations themselves may be dynamically migrated to the SXE at runtime as needed.

The SXE is distributed with a core library of basic "opcodes" implemented in the Java programming language known as the `sxe.core` package. For example, there is a class `/sxe/core/math/add.java`

*We are considering a flow-type synchronization annotation that would allow a persistent trigger to run "independently" of its parent node, as in some cases, where the called function may not be able to keep up with the rate at which data is being generated yet a greater sampling rate is desirable (e.g., triggers shared by two different programs running at different rates).

corresponding to the opcode “`sxe.core.math.add`” as there is for each opcode known to the SXE. We implement a custom Java `ClassLoader` to support the dynamic loading of new opcodes from trusted remote sources (i.e., JAR files over HTTP).*

Internally, all opcode methods manipulate `snObjects`, a first-class Java representation of various STEP data types. The `snObject` itself is a helper class that provides common methods that allow data to be easily serialized as XML for transmission between SXEs and for viewing results via a standard web browser (using standard mime-type appropriate content). Similarly, `snObjects` implement a method to parse an object from its XML representation. Specific `snObjects` exist including `snInteger`, `snString`, `snImage`, `snBoolean`, `snCommand`, etc. Opcode implementations are responsible for accepting `snObjects`, and returning `snObjects` such that the result may be passed further up the STEP graph. A sample example opcode is given below for illustrative purposes.

```
/* The addition Opcode */
snObject Call(snObjectArgList argv)
    throws CastFailure,
           InvalidArgumentCount

/* pop two integers from the args and return the
 * result as a snInteger */
return (snInteger) (argv.popInt() + argv.popInt());
```

While the implementation of an opcode handler must be in Java, within the body of the opcode, computations are not limited to Java calls (e.g., communication with remote hosts, execution of C++ code via the Java Native Interface, and generation and transmission of machine code to a remote host).

23.5.5 STEP Program/Node Removal

The removal of STEP nodes from the SXE may occur due to local or external events. When the evaluation of a STEP graph completes (either successfully or in error) the SXE reports the completion event with the STEP program ID to the SSD. The SXE may mark the local nodes for deletion if no other programs depend on these nodes (i.e., if any ancestor node attached to these nodes has a different program ID than that of this program, the SXE knows other programs are dependent on this computation). Externally requested node removal may be signaled by the SSD for operational reasons or by the request of an end user. Removal may be specified at the granularity of single nodes; however, removal of a node signals removal of any parent nodes (dependent tasks) including those from different programs (assuming the SSD knows best).

In either case, the SXE does not immediately delete the nodes from its URI namespace, rather deletion is a two-phase operation, consisting of garbage marking followed by a later physical deletion. The garbage marking algorithm is a straightforward postfix DAG ascent, while the cleanup algorithm simply iterates over all nodes, removing those which have expired.

23.6 Putting It All Together

We now illustrate the usage of `SNBENCH` by following a sample-sensing application throughout its life cycle. We assume that a Sensorium has been deployed, with SSD/SRM located at `ssd(.sensorium.bu.edu)` and with several participant SXEs. In particular, an SXE deployed on named host `lab-east` is online and with an attached video sensor. `lab-east` advertises its computational and sensing resources via periodic heartbeats to `ssd`.

An end user, Joe, would like to see the names and faces of people in that lab. Perhaps Joe does not know everyone’s name yet, such that an image of the people currently in the lab, with the faces of people detected

*We imagine that applets style restrictions could be used to allow opcodes from untrusted remote sources, instances or even separate VM created to ensure complete protection from this untrusted code.

superimposed on the image would be useful to our user. As opcodes that support these functionalities (e.g., grabbing frames and finding faces) are available to the SXEs, this program is easily composed in SNAFU.* The SNAFU program to accomplish this goal would read:

```
letconst x =
  snapshot(sensor("lab-east", "cam0")) in
  imgdraw(x, identify_faces(x))
```

The SNAFU compiler generates an unbound STEP graph, stored as XML. A shorthand of the STEP XML graph is shown below (some attributes have been removed for clarity). Note the usage of the `let` binding in the SNAFU program results in the second instance of “x” in the STEP program being stored as a splice onto the same node.

```
<step id="joes-program">
  <exp opcode="imgdraw" id="root1">
    <exp opcode="snapshot" id="snap">
      <sensor uri="http://lab-east/sxe/sensor/image/0"/>
    </exp>
    <exp opcode="identify_faces">
      <const>
        <splice target="snap"/>
      </const>
    </exp>
  </exp>
</step>
```

To submit the STEP program to the infrastructure, Joe will launch a web browser and navigate to the SSD that administers the Sensorium deployed in the lab[†] in this case: `http://ssd.sensorium.bu.edu:8081/snbench/ssd/`. An XSLT-rendered HTML interface is presented by the SSD, one option of which allows Joe to upload the STEP program (XML file) to the SSD through a standard web-based POST interface.

The SSD then parses the posted STEP graph looking for reusable STEP components (i.e., nodes). Assuming no STEP programs are deployed elsewhere in the Sensorium, the SSD proceeds to try and satisfy prebound computations. In our example, the sensor node is “bound” to `lab-east.sensorium.bu.edu` and the SSD’s scheduler requires that any opcode immediately dependent on a sensor node should be dispatched to that same resource as the sensor. In practice, this is a reasonable restriction, as it ensures that the SXE hosting the sensing device will be responsible for getting data from the sensor. This will not create a bottleneck as additional STEP programs needing data from this sensor will share the need for the same opcode and reuse will occur attaching new computations to that opcode. In our example, the `snapshot` opcode will be bound to `lab-east` and the `identify_faces` and `imgdraw` opcodes are free to be scheduled to any available SXE resource (potentially including `lab-east`).

To make things more interesting, we assume `lab-east` is only able to accommodate the `snapshot` opcode, so the STEP graph must be split across multiple SXEs. Fortunately, another SXE host on `c02` has available resources for both the `identify_faces` and the `imgdraw` opcodes. Note that if we were to split across three SXEs for these computations, the Sensorium would pay the communication penalty for transferring

*Recall, the SXE is extensible such that if the functionality Joe wishes to accomplish on an SXE is not defined in the SXE core opcode library, Joe may develop his own opcode implementations in Java and make the jar file available via web server. Such opcodes are accessible within SNAFU using a stub function `new_opcode` (“uri,” args); however, the usage cannot be type-checked so we warn against the general usage of this feature by anyone other than opcode developers during testing.

[†]We plan to implement a DNS-style “root” SSD that will allow users looking to dispatch a STEP program to a particular resource to locate the SSD that administers that SXE (i.e., SXE resolution that returns the location where programs for that node should be submitted).

the image twice (despite the splice). In this case, we only transfer the image once and the socket is reused as a splice target. This is illustrated in the shorthand STEP subgraphs given below:

```
lab-east.sensorium.bu.edu:
<step id="joes-program:a">
  <socket role="sender" id="a" peer="b">
    <exp opcode="snapshot">
      <sensor>http://lab-east/sxe/sensor/image/o</sensor>
    </exp>
  </socket>
</step>

c02.sensorium.bu.edu:
<step id="joes-program:b">
  <exp opcode="imgdraw" id="root1">
    <socket role="receiver" id="b" peer="a"/>
    <exp opcode="identify_faces">
      <const>
        <splice target="b"/>
      </const>
    </exp>
  </socket>
</exp>
</step>
```

The SSD dispatches each STEP subgraph to the appropriate SXE (via HTTP/1.1 POST). If the POST at either SXE fails (e.g., the SXE does not respond and fails to accept the STEP), the SSD deletes the graph posted at the other SXE by sending a DELETE of the STEP graph's program ID. If both SXEs respond with success codes (200 OK), the SSD and SRM commit their changes and are updated to maintain this new program. The SSD presents Joe with a web page containing a successful POST result and an HTTP link to the SXE node where he may (eventually) find the result of the computation: <http://c02.sensorium.bu.edu/snbench/sxe/node/root1>. Optionally, as a security measure, the SSD may be used as a relay to prevent end users from directly connecting to SXEs. Joe may now navigate to that link or other presented links to the node in the original STEP program tree and will see the current value or runtime state of each of the STEP subcomputations.

As soon as the SXE has accepted the posted STEP program, its own web namespace will be updated to include the posted nodes and their current execution state and values. The SXE on *lab-east* has the lower part of the STEP graph (and no external dependency) such that it can immediately start executing its portion of the STEP graph. When the socket node is encountered, *lab-east* tries to contact *c02* and in doing so, provides *c02* with the data it needs to begin its execution. When *c02* computes a result for the orphan node “root1” it will contact the SSD informing it that the program “joes-program” is complete.

As a single-run non-trigger program, the STEP evaluators on each SXE will only run the computation nodes through once and after a configurable amount of time both the nodes and the result are expunged from the SXEs.

23.7 Related Work

We restrict the focus of our discussion of related work to only those efforts focused on the development of programming paradigms for the composition of services for a general-purpose SN, as opposed to efforts focusing on application development frameworks for a particular class of SNs, or for a special-purpose architecture (e.g., motes) [7,8].

TAG [9] and Cougar [10] are examples of works wherein the SN is abstracted as a distributed data acquisition/storage infrastructure (i.e., “SN as a database” [11]). These solutions are limited to query style programs and thus lacking extensibility and arbitrary programmability.

Regiment [12] provides a macroprogramming functional language for SNs that is compiled to a Distributed Token Machines (DTMs) model [13] (DTMs are akin to Active Messages [14]). Although Regiment abstracts away many of the low-level management concerns of the mote platform, the DTM work is a highly customized solution aimed at the particular constraints of motes [1] in which multitasking and sharing resources is not a concern.

MagnetOS [15] provides greater flexibility with respect to the programs that can be deployed, virtualizing the resources of the SN to a single Java virtual machine (JVM). While this approach supports extensible dynamic programming, it lacks the ability to share SN system resources across autonomous applications. One may also argue that a JVM is not the best abstraction for an SN. Impala [16] uses a variety of runtime agents to provide a modular and adaptive framework, though their solution is programmed at the granularity of the individual sensor, and it too lacks support for multitasking an entire SN.

The work of Liu and Zhao [17] most closely resembles our vision and approach toward a shared, multi-tasking, high-powered SN. Microsoft's SONGs approach differs from ours insofar as (1) their semantic macroprogramming approach does not lend itself toward provisioning arbitrary computation and (2) reuse of computation is seemingly not on their radar.

The network virtualization of Ref. 18 must be mentioned as they face graph-embedding challenges for their resource resolution goals. Their goal of network emulation on dedicated hardware is significantly different enough from our goal of a unified SN that it should be no surprise that our solution is more lightweight and requires less hardware infrastructure (i.e., we do not require a dedicated system with the transfer of entire system images). We also plan to pursue the potential benefits of simulated annealing for our graph-embedding challenges.

23.8 Conclusion

SNBENCH provides a foundation for research that occurs both on top of and within the SNBENCH platform. Users of the SNBENCH framework may develop distributed sensing applications that run on the provided infrastructure. Researchers developing new sensing or distributed computation methodologies (e.g., the development of distributed vision algorithms and distributed hash tables) may take for granted the communication, scheduling, and dispatch services provided by the SNBENCH, freeing them to spend their energy investigating their area of interest and expertise. These modules can be provided as opcode implementations and plugged into the architecture with ease. Instead, in this section, we focus on the research taking place within the components of the SNBENCH itself; that is, the development and research that extends the SNBENCH to improve Sensorium functionalities and meet the unique challenges of this environment.

Runtime-type Specifications: As data-type annotations convey the requirements and safety of a dataflow, our notion of flow types extends type checking and safety to the control flow. Our work on flow types proceeds in two simultaneous directions: (1) the analysis and generation of a palette of useful deployment and runtime constraints/types and (2) the use of the TRAFFIC [5] engine to check and enforce control flow sanity and safety.

Runtime support for flowtypes: To support flowtypes, the SXEs must be modified to accommodate such scheduling parameters, including a monitoring infrastructure that ensures tasks are receiving the performance they have requested advertising an “accurate” real-time resource availability. Work is complete on the development of a hierarchical scheduler within the SXE, allowing STEP programs and even individual nodes to specify the scheduling model they require.

Performance profiling/benchmarking: Our present performance monitoring uses stub code to represent the free computational resources an SXE has and the computational cost of each opcode. It is clear that an accurate characterization of the computational availability of resources and each opcodes computational requirements will be needed to enable the SSD to *accurately* allocate resources and dispatch programs. We envision a solution in which SXEs generate simple performance statistics about each opcode as it is run, and these statistics are reported to the local SRM to build opcode performance profiles. Such an

approach allows new opcodes to be developed with their profiles dynamically built and probabilistically refined.

Expressive naming and name resolution: At present, we support naming of sensors via URI, relative to the physical SXE (host) that the sensor is connected to, or the use of wildcards to specify “any” sensor of a given type. The use of URIs requires the Resource Manager to maintain knowledge of all sensors connected to each host and perform some priority computation to resolve resources to compute and reserve physical sensor resources. Assuming sensor resolution processing, we wish to generalize this sensor resolution further with more powerful functions to support naming by identity (e.g., “The webcam in Azer’s Office”), naming by property (e.g., “Any two cameras aimed at Michael’s chair by 90° apart”), naming by performance characteristics (e.g., “Any processing element within 2 ms from WebCam1 and WebCam2”), and naming by content (e.g., “Any webcam, which sees Assaf right now”). Such naming conventions will require persistent, prioritized STEP queries to be running as the basis for these results; however, it is unknown, which such persistent queries should be instantiated, the resource cost of allocating sensors for these tasks, and how we can express these tasks as more commonly used expressions such that we produce the highest odds of success at the lowest exclusive computation cost.

Graph partitioning and optimality: From the perspective of communication cost, there is performance pressure to generate STEP schedules (STEP graph partitions) in which contiguous regions of the graph remain in the same partition, to minimize communication between SXEs. Although we may use the data types as an indication of the communication cost, it may be the case that those expressions that receive large amounts of data as input may have computation costs that dwarf their communication cost (e.g., pattern matching and face-finding). The deployment of such resource-intensive expressions may generate graphs in which we have many small regions and high communication cost. We anticipate several iterations of algorithms that attempt to achieve the “right” (or configurable) balance between network and computation cost, including heuristics borrowed from spreading metrics [6], simulated annealing [19], and others.

Security and safety: The emergence of embedded SNs in public spaces produces a clear and urgent need for well-planned, safe and secure infrastructure as security and safety risks are magnified. For example, a hacker gaining access to private emails or crashing a mail server is certainly bad; however, it is clearly worse if that same hacker can virtually case an office via stolen video feed, disable the security system, remotely unlock the door, and steal both the physical mail server and the data it contains. The Sensorium is an ideal test-bed for dealing with the inherent security issues in this novel domain, requiring the incorporation of mechanisms that provide authentication support for privacy, constraints, and trust. Currently, we are considering the implementation of some of the more basic security functionalities—e.g., using digest authentication for SSDs and SXEs, public key authentication for SXEs and SSL authentication for the SSD, and using SSL (https) to preserve the privacy communication between resources.

Scalability of networked SSDs: As mentioned in the previous sections, the SSD/SRM maintains resources for a “local-area” Sensorium. Although this hierarchal division seems rather natural, the number of resources to be monitored by an SSD must be “within reason.” We do not yet have experiments to establish what “reasonable” number of resources our local-area Sensorium can support. Moreover, as more Sensoria come online, there will inevitably be demand for computations that involve resources of disjoint Sensoria (e.g., nodes on the periphery between two SSD regions). Our initial approach is a tiered, DNS-like solution in which a root SSD can resolve specific resources beyond the scope of the local SSD and possible (when a local Sensorium is exhausted) out-source computations to another Sensorium. Such algorithms must be implemented, verified, and evaluated for scalability.

23.8.1 Catalytic Work

We envision SNBENCH as a catalyzing agent for a number of interesting research directions both intrinsic (i.e., research that aims to improve future generations of SNBENCH) as well as extrinsic (i.e., research that advances the state of the art in other areas). The following are examples, inspired by the projected trajectories of active research projects currently being pursued within our department.

Extrinsically, `snBENCH` abstracts out the details of the SN infrastructure allowing researchers to work on the problems they are best suited to deal with. For example, vision researchers do not need to understand communication protocols, real-time schedulers, or network resource reservation to research HCI approaches for assistive environments [20]. Similarly, `snBENCH` provides researchers in motion mining [21] and in-stream database applications [22] with a unique opportunity to implement and test-proposed approaches and algorithms in a real setting. The functional, and strongly typed nature of `snBENCH` programs may inspire the development of `snBENCH` (domain-specific) programming languages that are more expressive than `SNAFU`. In particular, `SNAFU` maps to `STEP` in a fairly straightforward way; additional, more expressive frontend languages with less intuitive mappings could also be developed.

Intrinsically, the ability of the SSD to guarantee system performance could leverage advances in overlay network QoS management [23], distributed scheduling [24], and online measurement, inference, and characterization of networked system performance [25]. Moreover, the algorithmic efficiency of the SSD will depend upon finding efficient solutions to labeled graph-embedding problems [26], where those labels will have interesting interactions with the scheduling and performance issues already raised. `SXEs` ought to be high-performance runtime systems, and thus can benefit significantly from operating systems virtualization [27] and optimization techniques [28].

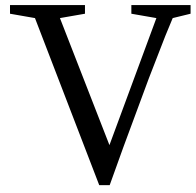
Acknowledgments

We would like to acknowledge Assaf Kfoury for his continued support and efforts, and Adam Bradley for his immense help and contribution to the initial conception of this work. We would also like to acknowledge the members of the larger `iBench` initiative at BU of which this work is part (<http://www.cs.bu.edu/groups/ibench/>).

References

1. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System Architecture Directions for Networked Sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000.
2. A. Bestavros, A. Bradley, A. Kfoury, and M. Ocean, "SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications," in *IEEE International Workshop on Broadband Advanced Sensor Networks (Basenets)*, Boston, October 2005.
3. M. Ocean, A. Bestavros, and A. Kfoury, "snBench: Programming and Virtualization Framework for Distributed Multitasking Sensor Networks," in *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 2006)*, New York, 2006, pp. 89–99, ACM Press, Ottawa, Canada.
4. Y. Gabay, M. Ocean, A. Kfoury, and L. Liu, "Computational Properties of `SNAFU`," Tech. Rep. BUCS-TR-2006-001, CS Department, Boston University, February 6, 2006.
5. A. Bestavros, A. Bradley, A. Kfoury, and I. Matta, "Typed Abstraction of Complex Network Compositions," in *ICNP'05: The 13th IEEE International Conference on Network Protocols*, Boston, November 2005.
6. G. Even, J. Naor, S. Rao, and B. Schieber, "Divide-and-Conquer Approximation algorithms via Spreading Metrics (Extended Abstract)," in *IEEE Symposium on Foundations of Computer Science*, 1995, pp. 62–71.
7. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The `nesC` Language: A Holistic Approach to Networked Embedded Systems," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PDLI)*, 2003.
8. P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002.
9. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, 2002.

10. Y. Yao and J. Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks," *SIGMOD Rec.*, vol. 31, no. 3, 2002.
11. R. Govindan, J. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker, "The Sensor Network as a Database," Tech. Rep. 02-771, CS Department, University of Southern California, 2002.
12. R. Newton and M. Welsh, "Region Streams: Functional Macroprogramming for Sensor Networks," in *DMSN'04: Proceedings of the 1st International Workshop on Data Management for Sensor Networks*, New York, 2004, pp. 78–87, ACM Press, Toronto, Canada.
13. R. Newton, Arvind, and M. Welsh, "Building Up to Macroprogramming: An Intermediate Language for Sensor Networks," in *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.
14. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992.
15. R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer, "On the Need for System-Level Support for Ad Hoc and Sensor Networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 2, pp. 1–5, 2002.
16. T. Liu and M. Martonosi, "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems," in *PPoPP'03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, 2003, ACM Press, San Diego, California.
17. J. Liu and F. Zhao, "Towards Semantic Services for Sensor-rich Information Systems," in *Second IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks (Basenets 2005)*, 2005.
18. R. Ricci, C. Alfeld, and J. Lepreau, "A Solver for the Network Testbed Mapping Problem," 2003.
19. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
20. J. J. Magee, M. R. Scott, B. N. Waber, and M. Betke, "EyeKeys: A Real-Time Vision Interface Based on Gaze Detection from a Low-Grade Video Camera," in *IEEE Workshop on Real-Time Vision for Human-Computer Interaction (RTV4HCI)*, July 2004.
21. T.-P. Tian, R. Li, and S. Sclaroff, "Tracking Human Body Pose on a Learned Smooth Space," in *IEEE Workshop on Learning in Computer Vision and Pattern Recognition*, 2005.
22. J. Considine, F. Li, G. Kollios, and J. W. Byers, "Approximate Aggregation Techniques for Sensor Databases," in *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE'04)*, Boston, April 2004.
23. M. Guirguis, A. Bestavros, I. Matta, N. Riga, G. Diamant, and Y. Zhang, "Providing Soft Bandwidth Guarantees Using Elastic TCP-based Tunnels," in *Proceedings of ISCC'04: IEEE Symposium on Computer and Communications*, Alexandria, Egypt, 2004.
24. A. Bestavros, "Load Profiling: A Methodology for Scheduling Real-Time Tasks in a Distributed System," in *ICDCS'97: The IEEE International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997.
25. A. Bestavros, J. Byers, and K. Harfoush, "Inference and Labeling of Metric-Induced Network Topologies," *IEEE Transactions on Parallel and Distributed Systems*, 2005.
26. J. Considine, J. W. Byers, and K. Mayer-Patel, "A Case for Testbed Embedding Services," in *Proceedings of HotNets-II*, November 2003.
27. Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West, "Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation," in *1st ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.
28. R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers," *IEEE Trans. Comp.*, vol. 53, no. 6, 2004.



Real-Time Database/ Data Services

24

Data-Intensive Services for Real-Time Systems

Krithi Ramamritham
Indian Institute of Technology

Lisa Cingiser DiPippo
University of Rhode Island

Sang Hyuk Son
University of Virginia

24.1	Introduction	24-1
	Data, Transaction, and System Characteristics • Scheduling and Transaction Processing • Distribution • Quality of Service and Quality of Data • Related Real-Time Data Services	
24.2	Data Freshness and Timing Properties	24-4
24.3	Transaction Processing	24-6
	Scheduling and Concurrency Control • Distributed Databases and Commit Protocols • Recovery Issues • Managing I/O and Buffers	
24.4	Quality of Service in Real-Time Data Services	24-12
	QoS Management • QoS Metrics • QoS Management in Distributed Data Services • QoS Management for Data Streams	
24.5	Data Services in Sensor Networks	24-17
	Real-Time Data Services in Sensor Networks	
24.6	Mobile Real-Time Databases	24-18
24.7	Dissemination of Dynamic Web Data	24-20
24.8	Conclusion	24-21

24.1 Introduction

Typically, a real-time system consists of a *controlling system* and a *controlled system*. In an automated factory, the controlled system is the factory floor with its robots, assembling stations, and the assembled parts, while the controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor. Thus, the controlled system can be viewed as the *environment* with which the computer interacts.

The controlling system interacts with its environment based on the data available about the environment, say from various sensors, for example, temperature and pressure sensors. It is imperative that the state of the environment, as perceived by the controlling system, be consistent with the actual state of the environment. Otherwise, the effects of the controlling systems' activities may be disastrous. Hence, timely monitoring of the environment as well as timely processing of the sensed information is necessary. The sensed data are processed further to derive new data. For example, the temperature and pressure information pertaining to a reaction may be used to derive the rate at which the reaction appears to be progressing. This derivation typically would depend on past temperature and pressure trends and so some of the needed information may have to be fetched from archival storage. Based on the derived data, where the derivation may involve multiple steps, actuator commands are set. For instance, in our example, the derived reaction rate is used

to determine the amount of chemicals or coolant to be added to the reaction. In general, the history of (interactions with) the environment are also logged in archival storage.

In addition to the timing constraints that arise from the need to continuously track the environment, timing correctness requirements in a real-time (database) system also arise because of the need to make data available to the controlling system for its decision-making activities. If the computer controlling a robot does not command it to stop or turn on time, the robot might collide with another object on the factory floor. Needless to say, such a mishap can result in a major catastrophe.

Besides robotics, applications such as medical patient monitoring, programmed stock trading, and military command and control systems like submarine contact tracking require timely actions as well as the ability to access and store complex data that reflects the state of the application's environment. That is, data in these applications must be valid, or *fresh*, when it is accessed in order for the application to perform correctly. In a patient monitoring system, data such as heart rate, temperature, and blood pressure must be collected periodically. Transactions that monitor the danger level of a patient's status must be performed within a specified time, and the data must be accessed within an interval that defines the validity of the data. If not, the computations made by the transactions do not reflect the current state of the patient's health.

Traditional databases do not provide for enforcement of the timing constraints imposed by the applications. A real-time database (RTDB) has all the features of a more traditional database, but it can also express and maintain time-constrained data and time-constrained transactions. In the last 15 years, there has been a great deal of research toward developing RTDBs that provide this functionality. In more recent years, much of the research that has been performed in the area of RTDBs has been applied, and extended in related fields, generally known as real-time data services. This chapter surveys the highlights of the research that has been performed to advance the state of the art in data-intensive real-time applications.

In developing RTDB solutions that provide the required timeliness of data and transactions, there are various issues that must be considered. Below is a discussion of some of the concerns that have been the subject of research in this field.

24.1.1 Data, Transaction, and System Characteristics

An RTDB must maintain not only the logical consistency of the data and transactions, it must also satisfy transaction timing properties as well as data temporal consistency. Transaction timing constraints include deadlines, earliest start times, and latest start times. Transactions must be scheduled such that these constraints are met. Data temporal consistency imposes constraints on how old a data item can be and still be considered valid. There are various ways to characterize real-time transactions. If an application requires that the timing constraints on transactions be met in order to be correct, these transactions are considered to be *hard*. If a *firm* transaction misses a timing constraint, it no longer has value to the system, and therefore can be aborted. A *soft* transaction provides some, likely reduced, value to the system after its constraints have been violated. Transactions can also be characterized by their timing requirements. That is, a transaction can be periodic or aperiodic. Finally, an RTDB system can be static or dynamic. In a static system, all data and requirements are known *a priori*, and the transactions can be designed and analyzed for schedulability up front. A dynamic system does not have this *a priori* knowledge, and therefore must be able to adapt to changing system requirements.

24.1.2 Scheduling and Transaction Processing

Scheduling and transaction processing techniques that consider data and transaction characteristics have been a major part of the research that has been performed in the field of RTDBs. Many of them consider the inherent trade-offs between quality of data versus timeliness of processing. For example, in a weather monitoring RTDB, it may be necessary to sacrifice image processing time and store lower-quality satellite images of areas that have no important weather activity going on. This will allow for more important areas, such as those with hurricane activity, to be more precise. Further, this will also allow for transactions accessing these important images to meet their access timing constraints. For another example, in a

programmed stock-trading application, a transaction that updates a stock price may be blocked by another transaction that is reading the same piece of data, and has a shorter deadline. If the stock price in question is getting “old” it would be in the interest of its temporal consistency to allow the updating transaction to execute. However, this execution could violate the logical consistency of the data or of the reading transaction. Thus, there is a trade-off between maintaining temporal consistency and maintaining logical consistency. If logical consistency is chosen, then there is the possibility that a stock price may become old, or that a transaction may miss a deadline. If, however, temporal consistency is chosen, the consistency of the data or of the transactions involved may be compromised.

24.1.3 Distribution

With the advent of high-speed, relatively low-cost networking, many of the applications that require an RTDB are not located on a single computer. Rather, they are distributed, and may require that the real-time data be distributed as well. Issues involved with distributing data include data replication, replication consistency, distributed transaction processing, and distributed concurrency control. When real-time requirements are added, these systems become much more complex. For instance, in a collaborative combat planning application where sensor data are collected by several ships in a region, and various decision makers are viewing the collected data, it is critical that all collaborators share the same view of the scenario. The degree of distribution is also an issue that researchers have been examining recently. That is, a distributed RTDB that exists on three stationary nodes will have different requirements than an RTDB made up of hundreds of mobile computers each of which can sense local data and share it with the others.

24.1.4 Quality of Service and Quality of Data

It is often not possible to maintain the logical consistency of a database and the temporal consistency as well. Therefore, there must be a trade-off made to decide, which is more important. In dynamic systems, where it is not possible to know and control the real-time performance of the database, it may be necessary to accept levels of service that are lower than optimal. It may also be necessary to trade-off the quality of the data (QoD) to meet specified timing constraints. The RTDBs must provide mechanisms to specify allowable levels of service and quality of data, and it must also provide a way to adjust these levels when it is necessary.

24.1.5 Related Real-Time Data Services

Many of the issues described above has dealt with the application of real-time techniques and theory to databases. Recently, there has been a trend toward applying the results of this core RTDB research to other, related applications. The key ingredients of these applications are the requirements for real-time response to requests for real-time data. While these newer technologies are not necessarily RTDBs, many of the important issues described above also apply to them. Sensor networks are a natural application for real-time data services because their main purpose is to provide sensed data to some requesting entity, very often with real-time constraints on the data and on the requests. A similar application that has recently benefited from early RTDB research is real-time mobile databases in which some or all of the nodes in the database can change location and are connected via a wireless network. Both of these applications bring new constraints. They both must deal with the unpredictability and lower bandwidth of wireless networks and may need to consider power issues in some or all of the devices used in the system. A third area of research deals with dissemination of streaming real-time data. A key goal in these applications is delivering temporally valid real-time data to requestors within specified timing constraints. More generally, large enterprise applications, such as telecom, healthcare, and transportation systems have embraced the notion of *real-time e-business* with large volumes of data being produced, processed, correlated, and aggregated in real time [101] to handle the current situation, while also predicting what is to come, creating a nimble responsive system in the process. We believe that the techniques described in this chapter will prove useful in many of the solutions that are being or will be developed to meet their needs.

This chapter describes the important research results that have been produced in the above areas. It starts out in Section 24.2 by defining data freshness and timing properties. Section 24.3 presents research that has been done in the area of transaction processing for RTDBs. As mentioned above, this is where much of the RTDB focus has been in the past. Quality of service (QoS), and also QoD issues are addressed in Section 24.4. In Section 24.5, we present current research in the field of real-time data management in sensor networks. Section 24.6 describes how the core RTDB research has been leveraged to develop real-time mobile databases. In Section 24.7, we present the current research involved with real-time data dissemination and data streaming. Finally, Section 24.8 concludes with a summary of the current research, and a look at real-time data services.

24.2 Data Freshness and Timing Properties

The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of *temporal consistency*: the need to maintain coherency between the state of the environment and its reflection in the database. This arises from the need to keep the controlling system's view of the state of the environment consistent with the actual state of the environment.

A RTDB is composed of real-time objects, which are updated by periodic sensor transactions. An *object* in the database models a real-world entity, for example, the position of an aircraft. A real-time object is one whose state may become invalid with the passage of time. Associated with the state is a temporal validity interval. To monitor the states of objects faithfully, a real-time object must be refreshed by a sensor transaction before it becomes invalid, that is, before its temporal validity interval expires. The actual length of the temporal validity interval of a real-time object is application dependent.

Here, we do not restrict the notion of sensor data to the data provided by physical sensors. Instead, we consider a broad meaning of sensor data. Any data item, whose value reflects the time-varying real-world status, is considered a sensor data item.

Sensor transactions are generated by intelligent sensors, which periodically sample the value of real-time objects. When sensor transactions arrive at RTDBs with sampled data values, their updates are installed and real-time data are refreshed. So one of the important design goals of RTDBs is to guarantee that temporal data remain fresh, that is, they are always valid. RTDBs that do not keep track of temporal validity of data may not be able to react to abnormal situations in time. Therefore, efficient design approaches are needed to guarantee the freshness of temporal data in RTDBs while minimizing the CPU workload resulting from periodic sensor transactions.

Whereas in the rest of this section we will be dealing with sensor transactions, in particular, with the derivation of their timing properties, it must be pointed out that many RTDBs also possess *user transactions*, transactions that access temporal as well as nontemporal data. User transactions typically arrive aperiodically. User transactions do not write any temporal data object, but they can read/write nontemporal data and their execution time and data access pattern can be time-varying. For example, in a decision support system a transaction can read different sets of dynamic, that is, temporally varying, data and perform different operations according to the situation.

From here on, $\mathcal{T} = \{\tau_i\}_{i=1}^m$ refers to a set of periodic sensor transactions $\{\tau_1, \tau_2, \dots, \tau_m\}$ and $\mathcal{X} = \{X_i\}_{i=1}^m$ to a set of temporal data. All temporal data are assumed to be kept in main memory. Associated with X_i ($1 \leq i \leq m$) is a validity interval of length α_i ; transaction τ_i ($1 \leq i \leq m$) updates the corresponding data X_i . Because each sensor transaction updates different data, no concurrency control is considered for sensor transactions. We assume that a sensor always samples the value of a temporal data at the beginning of its period, and all the first instances of sensor transactions are initiated at the same time. C_i , D_i , and P_i ($1 \leq i \leq m$) denote the execution time, relative deadline, and period of transaction τ_i , respectively. D_i of sensor transaction τ_i is defined as follows:

$$D_i = \text{Deadline of } \tau_i - \text{Sampling time of } \tau_i$$

Deadlines of sensor transactions are firm deadlines. The goal of an RTDB designer is to determine P_i and D_i , such that all the sensor transactions are schedulable and CPU workload resulting from sensor transactions is minimized. Let us assume a simple execution semantics for periodic transactions: a transaction must be executed once every period. However, there is no guarantee of when an instance of a periodic transaction is actually executed within a period. We also assume that these periodic transactions are preemptable.

Assume the period and relative deadline of a sensor transaction are set to be equal to the data validity interval. Because the separation interval of the execution of two consecutive instances of a transaction can exceed the validity interval, data can become invalid under this *One-One* approach. So this approach cannot guarantee the freshness of temporal data in RTDBs.

To guarantee the freshness of temporal data in RTDBs, the period and relative deadline of a sensor transaction are each typically set to be less than or equal to one-half of the data validity interval [39,78]. Unfortunately, even though data freshness is guaranteed, this design approach at least doubles CPU workload of the sensor transaction in the RTDBs compared to the *One-One* approach. In a *More-Less* approach, the period of a sensor transaction is assigned to be *more* than *half* of the validity interval of the temporal data updated by the transaction, while its corresponding relative deadline is assigned to be *less* than *half* of the validity interval of the same data. However, the sum of the period and relative deadline always equals the length of the validity interval of the data updated. Therefore, given a set of sensor transactions in RTDBs, we need to find periods and deadlines of update transactions based on the temporal validity intervals of data such that the CPU workload of sensor transactions is minimized and the schedulability of the resulting sensor transactions is guaranteed. The *More-Less* approach achieves this as shown in Ref. 110. Recent work [39] on similarity-based load adjustment also adopts this approach to adjust periods of sensor transactions based on similarity bounds. Recently, the deferrable scheduling algorithm (*DS-FP*) has been studied in Refs. 108 and 109. Compared to *More-Less*, *DS-FP* reduces update workload further by adaptively adjusting the separation of two consecutive jobs while guaranteeing *data freshness*. (Refer to Chapter 26 for a detailed comparison of one-one, half-half, and more-less approaches.)

In addition to maintaining absolute consistency, there is a need for *mutual consistency* (also known as *relative validity*), motivated by the observation that many objects may be related to one another and that the system should present a logically consistent view of the objects to any query. The need for mutual consistency is apparent in the web domain, for example, on newspaper websites carrying breaking news stories that consist of text objects accompanied by embedded images and audio/video clips. Since such stories are updated frequently (as additional information becomes available), the source serving the request has to ensure that any cached version of the text is consistent with the embedded objects. Similarly, while delivering up-to-the-minute sports information, it has to be ensured that the cached objects are mutually consistent with each other. In RTDBs, the values of the objects sampled by different sensors may be related to one another. Thus, it is important to preserve the relative freshness of these objects in addition to their individual freshness. This motivates the work reported here.

There are already various approaches proposed to maintain mutual consistency in the web domain, for example, see Ref. 100. Let us first review the mutual consistency semantics. For simplicity, the definitions are given for two objects, but they can be generalized for n objects. Consider two objects a and b that are related to each other and updated by transactions 1 and 2, respectively. The database versions of the objects a and b at time t , that is, D_t^a and D_t^b , are defined to be mutually consistent in the time domain if the following condition holds:

$$\text{if } D_t^a = A_{t_1}^a \text{ and } D_t^b = A_{t_2}^b, \text{ then } |t_1 - t_2| \leq \delta$$

where A_t^a refers to the correct version of the object a at time t . For example, say the object is the altitude of an aircraft and at time t_1 the aircraft was at an altitude x . It is possible that at time t ($t_1 < t$) the database version reads x . So $D_t^a = A_{t_1}^a$ means that the version of the object in the database at time t is actually the correct version at time t_1 . We refer to $|t_1 - t_2|$ as the mutual gap between the two transactions 1 and 2. Assuming that the transactions update the database with the current version, for each object a , $D_t^a = A_{t_1}^a$,

where t_1 is the time at which the value of a was last updated. So the mutual gap between the two objects a and b is just the difference in time when they were last updated by their respective update transactions.

The problem of checking whether, given an update transaction schedule, a periodic query would be able to read mutually consistent values is investigated in Ref. 45. Solutions for both single query and multiple queries for the case of nonpreemptable queries are proposed along with design approaches to decide the period and relative deadline of a query so that it would satisfy mutual consistency. Compared to Ref. 96, where it is assumed that earliest deadline first and rate-monotonic scheduling algorithms [60] are used for online scheduling of queries and transactions, Ref. 45 derives schedules of queries offline, which is achieved by choosing a right time to start queries such that they can preserve mutual consistency of accessed objects.

Concepts from active databases can be used to maintain the temporal validity of derived objects, where a derived object's value is dependent on the values of associated sensor data and it is updated sporadically. The technique described in Ref. 8 is one such approach and is designed to ensure that maintaining temporal consistency does not result in a significant deterioration of the real-time performance with respect to meeting transaction deadlines.

Before we conclude this section, we would like to point out that effective scheduling of real-time transactions is greatly hampered by the large variance that typically exists between average-case and worst-case execution times in a database system. This unpredictability arises owing to a variety of factors including data-value-dependent execution patterns, blocking or restarts owing to data contention, and disk latencies that are a function of the access history. Ref. 82 attempts to eliminate some of these sources of unpredictability and thereby facilitate better estimation of transaction running times. It employs main-memory database technology and presents new logical and physical versioning techniques that ensure that read-only transactions are completely isolated from update transactions with respect to data contention. An application of this technique to the adaptive cruise control subsystem of automobiles is described in Refs. 31, 32, 72, and 81.

24.3 Transaction Processing

In this section, issues related to the timely processing of transactions and queries such that they produce logically correct database states as well as outputs are discussed. Section 24.3.1 deals with scheduling and concurrency control issues, Section 24.3.2 summarizes results in the commit processing area, Section 24.3.3 presents work on recovery, and Section 24.3.4 outlines approaches developed to deal with resources such as I/O and buffers.

24.3.1 Scheduling and Concurrency Control

In this section, we discuss various aspects of transaction and query processing where the transactions and queries have time constraints attached to them and there are different consequences of not satisfying those constraints.

A key issue in transaction processing is *predictability*. In the context of an individual transaction, this relates to the question: "will the transaction meet its time-constraint?" Section 24.3.1.1 deals with the processing of transactions that have hard deadlines, that is, deadlines that must be met; and Section 24.3.1.2 deals with transactions that have soft deadlines, that is, deadlines which can be missed, but the larger the number of misses, the lower the value accrued to the system.

24.3.1.1 Dealing with Hard Deadlines

All transactions with hard deadlines must meet their time constraints. Since dynamically managed transactions cannot provide such a guarantee, the data and processing resources as well as the time needed by such transactions have to be guaranteed to be made available when necessary. There are several implications.

First, we have to know when the transactions are likely to be invoked. This information is readily available for periodic transactions, but for aperiodic transactions, by definition, it is not. The smallest

separation time between two incarnations of an aperiodic transaction can be viewed as its period. Thus, we can cast all hard real-time transactions as *periodic* transactions.

Second, to ensure *a priori* that their deadlines will be met, we have to determine their resource requirements and worst-case execution times. This requires that many restrictions be placed on the structure and characteristics of real-time transactions.

Once we have achieved the above, we can treat the transactions in a manner similar to the way real-time systems treat periodic tasks that require guarantees, that is, by using static *table-driven* schedulers or preemptive *priority-driven* approaches. Static *table-driven* schedulers reserve specific time slots for each transaction. If a transaction does not use all of the time reserved for it, the remaining time may be reclaimed to start other hard real-time transactions earlier than planned. Otherwise, it can be used for soft real-time transactions or left idle. The table-driven approach is obviously very inflexible. One priority-driven approach is the rate-monotonic priority assignment policy. One can apply the schedulability analysis tools associated with it to check if a set of transactions are schedulable given their execution times, periods, and data requirements. This is the approach discussed in Ref. 85, where periodic transactions that access main memory resident data via read and write locks are scheduled using rate-monotonic priority assignment.

We mentioned earlier that the variance between the worst-case computational needs and the actual needs must not be very large. We can see why. Since the schedulability analysis is done with respect to worst-case needs, if the variance is large, many transactions that may be schedulable in the average case will be considered infeasible in the worst case. Also, if the table-driven approach is used, a large variance will lead to large idle times.

In summary, while it is possible to deal with hard real-time transactions using approaches similar to those used in real-time systems, many restrictions have to be placed on these transactions so that their characteristics are known *a priori*. Even if one is willing to deal with these restrictions, poor resource utilization may result given the worst-case assumptions made about the activities.

24.3.1.2 Dealing with Soft Transactions

With soft real-time transactions, we have more leeway to process transactions since we are not required to meet the deadlines all the time. Of course, the larger the number of transactions that meet their deadlines the better. When transactions have different values, the value of transactions that finish by their deadlines should be maximized. The complexity involved in processing real-time transactions comes from these goals. That is to say, we cannot simply let a transaction run, as we would in a traditional database system, and abort it should its deadline expire before it commits. We must meet transaction deadlines by adopting priority-assignment policies and conflict resolution mechanisms that explicitly take time into account. Note that priority assignment governs CPU scheduling and conflict resolution determines, which of the many transactions contending for a data item will obtain access. As we will see, conflict resolution protocols make use of transaction priorities and because of this, the priority assignment policy plays a crucial role. We also discuss the performance implications of different deadline semantics.

Scheduling and Conflict Resolution. Rather than assigning priorities based on whether the transactions are CPU or I/O (or data) bound, RTDB systems must assign priorities based on transaction time constraints and the value they impart to the system. Possible policies are

- Earliest-deadline-first
- Highest-value-first
- Highest-value-per-unit-computation-time-first
- Longest-executed-transaction-first

For the purpose of conflict resolution in RTDB systems, various *time-cognizant* extensions of two-phase locking, optimistic, and timestamp-based protocols have been proposed in the literature. See, for example, Refs. 1, 19, 33, 40–42, 61, 92, and 98. Some of these are discussed below.

In the context of two-phase locking, when a transaction requests a lock that is currently held by another transaction, we must take into account the characteristics of the transactions involved in the conflict. Considerations involved in conflict resolution are the deadline and value (in general, the priority) of

transactions, how long the transactions have executed, and how close they are to completion. Consider the following set of protocols:

- If a transaction with a higher priority is forced to wait for a lower-priority transaction to release the lock, a situation known as *priority inversion* arises. This is because a lower-priority transaction makes a higher-priority transaction to wait. In one approach to resolving this problem, the lock holder *inherits* the lock requester's priority whereby it completes execution sooner than with its own priority.
- If the lock holding transaction has lower priority, abort it. Otherwise, let the lock requester wait.
- If the lock holding transaction is closer to its deadline, lock requester waits, independent of its priority.

Priority inheritance is shown to reduce transaction blocking times [40]. This is because the lock holder executes at a higher priority (than that of the waiting transaction) and hence finishes early, thereby blocking the waiting higher-priority transaction for a shorter duration. However, even with this policy, the higher-priority transaction is blocked, in the worst case, for the duration of a transaction under strict two-phase locking. As a result, the priority-inheritance protocol typically performs even worse than a protocol that makes a lock requester wait independent of its priority. If a higher-priority transaction always aborts a low-priority transaction, the resulting performance is sensitive to data contention. In contrast, if a lower-priority transaction that is closer to completion inherits priority rather than aborting, then a better performance results even when data contention is high. Such a protocol is a combination of the abort-based protocol proposed for traditional databases and the priority-inheritance protocol proposed for real-time systems [86]. In other words, the superior performance of this protocol [40] shows that even though techniques that work in real-time systems on the one hand and database systems on the other may not be applicable directly, they can often be tailored and adapted to suit the needs of RTDB systems. It should be noted that abort-based protocols (as opposed to wait-based) are especially appropriate for RTDB systems because of the time constraints associated with transactions.

Let us now consider optimistic protocols. In protocols that perform backward validation, the validating transaction either commits or aborts depending on whether it has conflicts with transactions that have already committed. The disadvantage of backward validation is that it does not allow us to take transaction characteristics into account. This disadvantage does not apply to forward validation. In forward validation, a committing transaction usually aborts ongoing transactions in case they conflict with the validating transaction. However, depending on the characteristics of the validating transaction and those with which it conflicts, we may prefer not to commit the validating transaction. Several policies have been studied in the literature [33,34,42]. In one, termed *wait-50*, a validating transaction is made to wait as long as more than half the transactions that conflict with it have earlier deadlines. This is shown to have superior performance.

Time-cognizant extensions to timestamp-based protocols have also been proposed. In these, when data accesses are out of timestamp order, the conflicts are resolved based on their priorities. In addition, several combinations of locking-based, optimistic, and timestamp-based protocols have been proposed but require quantitative evaluation [61].

Exploiting multiple versions of data for enhanced performance has been addressed in Ref. 50. Multiple versions can reduce conflicts over data. However, if data must have temporal validity, old versions, which are outdated must be discarded. Also, when choosing versions of related data, their relative consistency requirements must be taken into account: consider a transaction that uses multiversioned data to display aircraft positions on an air-traffic controller's screen. The data displayed must have both absolute validity as well as relative validity.

Different transaction semantics are possible with respect to discarding a transaction once its deadline is past. For example, with firm deadlines, a late transaction is aborted once its deadline expires [34]. In general, with soft deadlines, once a transaction's value drops to zero, it is aborted [41]. In contrast, in the transaction model assumed in Ref. 1, all transactions have to complete execution even if their deadlines have expired. In this model, delayed transactions may cause other transactions also to miss their deadlines and this can have a cascading effect.

Experimental studies reported in the literature cited at the end of this article are very comprehensive and cover most aspects of real-time transaction processing, but most have not considered time constraints associated with data.

Database systems in which time validity intervals are associated with the data are discussed in Refs. 51, 52, and 96. Such systems introduce the need to maintain data temporal consistency in addition to logical consistency. The performance of several concurrency control algorithms for maintaining temporal consistency are studied in Ref. 96. It is pointed out in Ref. 96 that it is difficult to maintain the data and transaction time constraints due to the following reasons:

- A transient overload may cause transactions to miss their deadlines.
- Data values may become out of date due to delayed updates.
- Priority-based scheduling can cause preemptions, which may cause the data read by the transactions to become temporally inconsistent by the time they are used.
- Traditional concurrency control ensures logical data consistency but may cause temporal data inconsistency.

Motivated by these problems, and taking into account the fact that transactions process data with validity constraints and such data will be refreshed with sensor transactions, the notion of *data-deadline* is introduced in Ref. 112. Informally, data-deadline is a deadline assigned to a transaction due to the temporal constraints of the data accessed by the transaction. Further, a *forced wait* policy, which improves performance significantly by forcing a transaction to delay further execution until a new version of sensor data becomes available is also proposed.

In Ref. 51, a class of real-time data access protocols called SSP (similarity stack protocols) is proposed. The correctness of SSP is based on the concept of similarity, which allows different but sufficient timely data to be used in a computation without adversely affecting the outcome. SSP schedules are deadlock free, subject to limited blocking and do not use locks. In Ref. 52, weaker consistency requirements based on the similarity notion are proposed to provide more flexibility in concurrency control for data-intensive real-time applications. While the notion of data similarity is exploited in their study to relax serializability (hence increase concurrency), in Ref. 112, it is coupled with data-deadline and used to improve the performance of transaction scheduling. The notion of similarity is used to adjust transaction workload by Ho et al. [39] and incorporated into embedded applications (e.g., process control) in Ref. 21.

24.3.2 Distributed Databases and Commit Protocols

Many RTDB applications are inherently distributed in nature. These include the intelligent network services database, telecom databases, mobile telecommunication systems, and the 1-800 telephone service in the United States. More recent applications include the directory, data feed, and electronic commerce services that have become available on the World Wide Web. The performance, reliability, and availability of such applications can be significantly enhanced through the replication of data on multiple sites of the distributed network.

An algorithm for maintaining consistency and improving the performance of replicated distributed real-time database systems (DRTDBS) is proposed in Ref. 93. In this algorithm, a multiversion technique is used to increase the degree of concurrency. Replication control algorithms that integrate real-time scheduling and replication control are proposed in Ref. 94. In contrast to the relaxed correctness assumed by the latter, conventional one-copy serializability supported by the algorithm called managing isolation in replicated real-time object repositories (MIRROR) is reported in Ref. 111. MIRROR is a concurrency control protocol specifically designed for firm deadline applications operating on replicated RTDBs. MIRROR augments the classical O2PL concurrency control protocol with a state-based real-time conflict resolution mechanism. In this scheme, the choice of conflict resolution method is a dynamic function of the states of the distributed transactions involved in the conflict. A feature of the design is that acquiring the state knowledge does not require intersite communication or synchronization, nor does it require modifications to the two-phase commit protocol.

The performance of the classical distributed 2PL locking protocol (augmented with the priority abort (PA) and priority inheritance (PI) conflict resolution mechanisms) and of OCC algorithms in replicated DRTDBS was studied in the literature for real-time applications with soft deadlines. The results indicate that 2PL-PA outperforms 2PL-PI only when the update transaction ratio and the level of data replication are both low. Similarly, the performance of OCC is good only under light transaction loads. Making clear-cut recommendations on the performance of protocols in the soft deadline environment is rendered difficult, however, by the following: (1) there are two metrics: missed deadlines and mean tardiness, and protocols, which improve one metric usually degrade the other; (2) the choice of the postdeadline value function has considerable impact on relative protocol performance; and (3) there is no inherent load control, so the system could enter an unstable state.

Temporal consistency guarantees are also studied in distributed real-time systems. In Ref. 114, *Distance constrained scheduling* is used to provide temporal consistency guarantees for real-time primary-backup replication service.

Let us now consider the transaction commitment process. Once a transaction reaches its commit point, it is better to let it commit quickly so that its locks can be released soon. If commit delays are not high, which will be the case in a centralized database, the committing transaction can be given a high enough priority so that it can complete quickly. The solution is not so easy in a distributed system because of the distribution of the commitment process. Furthermore, since a deadline typically refers to the deadline until the end of the two-phase commit, but since the decision on whether or not to commit is taken in the first phase, we can enter the second phase only if we know that it will complete before the deadline. This requires special handling of the commit process. An alternative is to associate the deadline with the beginning of the second phase, but this may delay subsequent transactions since locks are not released until the second phase. A few papers in the literature [95,113] have tried to address the issue of distributed commit processing but have required either relaxing the traditional notion of atomicity or strict resource allocation and resource performance guarantees from the system.

In Ref. 35, its authors propose and evaluate a commit protocol that is specifically designed for the real-time domain without these changes. PROMPT allows transactions to optimistically borrow in a controlled manner, the updated data of transactions currently in their commit phase. This controlled borrowing reduces the data inaccessibility and the priority inversion that is inherent in distributed real-time commit processing. A simulation-based evaluation shows PROMPT to be highly successful as compared to the classical commit protocols in minimizing the number of missed transaction deadlines. In fact, its performance is close to the best online performance that could be achieved using the optimistic lending approach.

24.3.3 Recovery Issues

Recovery is a complex issue even in traditional databases and is more so in RTDB systems for two reasons. First, the process of recovery can interfere with the processing of ongoing transactions. Specifically, suppose we are recovering from a transaction aborted owing to a deadline miss. If locks are used for concurrency control, it is important to release them as soon as possible so that waiting transactions can proceed without delay so as to meet their deadlines. However, it is also necessary to undo the changes done by the transaction to the data if in-place updates are done. But this consumes processing time that can affect the processing of transactions that are not waiting for locks to be released. Whereas optimistic concurrency control techniques or a shadow-pages-based recovery strategy can be used to minimize this time, they have several disadvantages including lack of commercial acceptance. Second, unlike traditional databases where permanent data should always reflect a consistent state, in RTDBs, the presence of temporal data, while providing some opportunities for quicker recovery [103], adds to the complexities of the recovery of transactions. Specifically, if a transaction's deadline expires before it completes the derivation of a data item, then rather than restoring the state of the data to its previous value, it could declare the data to be invalid thereby disallowing other transactions from using the value. The next instance of the transaction, in case the data is updated by a periodic transaction, may produce a valid state.

There is a need for designing new algorithms for logging and recovery in RTDBs because the sequential nature of logging and the lack of time and priority cognizance during recovery are not in tune with the priority-oriented and preemptive nature of activities in RTDBs. Motivated by this need, Ref. 77 presents SPLIT, a partitioned logging and recovery algorithm for RTDBs, and ARUN (algorithms for recovery using nonvolatile high speed store), a suite of logging and recovery algorithms for RTDBs. To improve performance, ARUN makes use of the solid state disk (SSD) technology that is referred to as nonvolatile high speed store (NVHSS).

The logging and recovery algorithms are based on dividing data into a set of equivalence classes derived from a taxonomy of data and transaction attributes. For example, data can be broadly classified into hot and cold depending on the type of the data (such as critical and temporal), and type of the transactions (such as high priority and low priority), that accesses the data. The logging and recovery algorithms assume that the classes of data are disjoint and that the recovery of one class can be done without having to recover the other classes, and if one class is recovered, then the transactions accessing that class can proceed without having to wait for the system to recover the other classes. For instance, network management systems consist of different kinds of data such as the performance, traffic, configuration, fault, and security management data. The fault management data, which could potentially get updated frequently, is very critical to the operation that recovering it immediately after a crash can improve the performance of the system. A recent work toward achieving bounded and predictable recovery using real-time logging based on these ideas is presented in Ref. 90.

24.3.4 Managing I/O and Buffers

While the scheduling of CPU and data resources has been studied fairly extensively in the RTDB literature, studies of scheduling approaches for dealing with other resources, such as disk I/O, and buffers has begun only recently. In this section, we review some recent work in this area and discuss some of the problems that remain.

I/O scheduling is an important area for real-time systems given the large difference in speeds between CPU and disks and the resultant impact of I/O devices' responsiveness on performance. Since the traditional disk scheduling algorithms attempt to minimize average I/O delays, just like traditional CPU scheduling algorithms aim to minimize average processing delays, time-cognizant I/O scheduling approaches are needed.

It must be recognized that what is important is the meeting of transaction deadlines and not the individual deadlines that may be attached to I/O requests. Assume that we model a transaction execution as a sequence of (disk I/O, computation) pairs culminating in a set of disk I/O's, the latter arising from writes to log and to the changed pages. Suppose we assign (intermediate) deadlines to the I/O requests of a transaction given the transaction's deadline. One of the interesting questions with regard to disk I/O scheduling is: How does one derive the deadline for an I/O request from that of the requesting transaction? First of all, it must be recognized that depending on how these I/O deadlines are set, deadlines associated with I/O requests may be *soft* since even if a particular I/O deadline is missed, the transaction may still complete by its deadline. This is the case if I/O deadlines are set such that the overall laxity (i.e., the difference between the time available before the deadline and the total computation time) of a transaction is uniformly divided among the computations and the I/O. In contrast, assume that an intermediate deadline is equal to the latest completion time (i.e., the time an I/O must complete assuming that subsequent computations and I/O are executed without delay). This is the less-preferred method since we now have a firm deadline associated with I/O requests—if an I/O deadline is missed, there is no way for the transaction to complete by its deadline and so the requesting transaction must be aborted.

Work on I/O scheduling includes [2,20,22]. The priority-driven algorithm described in Ref. 20 is a variant of the traditional SCAN algorithm, which works on the elevator principle to minimize disk arm movement. Without specifying how priorities are assigned to individual I/O requests, Ref. 20 proposes a variant in which the SCAN algorithm is applied to each priority level. Requests at lower priority are serviced only after those at higher priority are served. Thus, if after servicing a request, one or more

higher-priority requests are found waiting, the disk arm moves toward the highest-priority request that is closest to the current disk arm position. In the case of requests arising from transactions with deadlines, priority assignment could be based on the deadline assigned to the I/O request.

Another variant of SCAN, FDSCAN [2] directly takes I/O deadlines into account. In this algorithm, given the current position of the disk arm, the disk arm moves toward the request with the earliest deadline that can be serviced in time. Requests that lie in that direction are serviced and after each service it is checked whether (1) a request with an even earlier deadline has arrived and (2) the deadline of the original result cannot be met. In either case, the direction of disk arm movement may change.

Clearly, both these protocols involve checks after each request is served and so incur substantial runtime overheads. The protocols described in Ref. 22 are aimed at avoiding the impact of these checks on I/O performance. Specifically, the protocols perform the necessary computations while I/O is being performed. In the SSED algorithm (shortest-seek and earliest deadline by ordering), the need to give higher priority to requests, with earlier deadlines is met while reducing the overall seek times. The latter is accomplished by giving a high priority to requests, which may have large deadlines but are very close to the current position of the disk arm. A variant of SSED is SSEDV, which works with specific deadline values rather than deadline orderings. Ref. 22 shows how both the algorithms can be implemented so as to perform disk scheduling while service is in progress and shows that the algorithms have better performance than the other variants of the SCAN algorithms.

Another resource for which contention can arise is the database buffer. Studies discussed in Ref. 20 show that transaction priorities must be considered in buffer management.

24.4 Quality of Service in Real-Time Data Services

In a number of real-time and embedded applications including e-commerce, agile manufacturing, sensor networks, traffic control, target tracking, and network management, transactions should be processed within their deadlines, using fresh (temporally consistent) data that reflect the current real-world status. Meeting both requirements of timeliness and data freshness is challenging. Generally, transaction execution time and data access pattern are not known *a priori*, and they vary dynamically. For example, transactions in stock trading may read varying sets of stock prices, and perform different arithmetic/logical operations to maximize the profit considering the current market status. Further, transaction timeliness and data freshness can often pose conflicting requirements. By preferring application transactions to sensor updates, the deadline miss ratio is improved; however, the data freshness might be reduced. Alternatively, the freshness increases if sensor data updates receive a higher priority. Recently, emerging applications need to operate on continuous unbounded data streams coming from sensors. For real-time stream data management, it is critical to understand the effects of trade-offs between the quality of query results and the QoS to achieve both the timeliness and the required level of quality. In this section, we discuss QoS management in RTDBs, discuss performance metrics for QoS management, and survey several approaches to QoS management and stream data management.

24.4.1 QoS Management

Despite the abundance of the QoS research, QoS-related work is relatively scarce in database systems. Priority adaptation query resource scheduling (PAQRS) is one of the first significant work dealing with QoS in terms of timeliness. It provides timeliness differentiation of query processing in a memory-constrained environment [73]. From the observation that the performance of queries can vary significantly depending on the available memory, per-class query response time was differentiated by an appropriate memory management and scheduling. Given enough memory, queries can read the operand relations at once to produce the result immediately. If less memory is allocated, they have to use temporary files to save the intermediate results, therefore, the query processing may slow down. In this way, query deadline miss ratios were differentiated between the classes. However, no data freshness issues were considered, and the performance could easily fluctuate under the workload changes.

Stanford real-time information processor (STRIP) addressed the problem of balancing between the freshness and transaction timeliness [5]. To study the trade-off between freshness and timeliness, four scheduling algorithms were introduced to schedule updates and transactions, and the performance was compared. In their later work, a similar trade-off problem was studied for derived data [6]. Ahmed and Vrbsky [7] proposed a new approach to maintain the temporal consistency of derived data. Different from STRIP, an update of a derived data object is explicitly associated with a certain timing constraint, and is triggered by the database system only if the timing constraint could be met. By a simulation study, the relative performance improvement was shown compared to the forced delay scheme of STRIP.

The correctness of answers to database queries can be traded off to enhance the timeliness. A query processor, called APPROXIMATE [102], can provide approximate answers depending on the availability of data or time. An imprecise computation technique (milestone approach [62]) is applied by APPROXIMATE. In the milestone approach, the accuracy of the intermediate result increases monotonically as the computation progresses. Therefore, the correctness of answers to the query could monotonically increase as the query processing progresses.

To address QoS management in real-time data services, Kang et al. [46,47] developed a QoS management architecture called QMF. In QMF, a formal control theoretic approach is applied to compute the required workload adjustment considering the miss-ratio error, that is, the difference between the desired miss ratio and the currently measured miss ratio. Feedback control is used since it is very effective to support the desired performance when the system model includes uncertainties [66,74]. At each sampling instant, the feedback-based miss-ratio controller measures the miss-ratio and computes the miss ratio error, and the controller computes the control signal, that is, the required workload adjustment to react to the error.

According to the required workload adjustment computed in the feedback loop, flexible freshness management and admission control are applied to support the desired miss ratio without affecting the freshness. Using the notion of perceived freshness, an *adaptive update policy* was developed to maintain the freshness in a cost-effective manner. This flexible approach contrasts to the existing database update policy, commonly accepted in the RTDB research such as in Refs. 5 and 49, which is fixed and not adaptable regardless of the current system status.

To prevent potential deadline misses or stale data accesses owing to the delay for on-demand updates, Kang et al. [46] developed an alternative approach in which all data are updated immediately. In that approach, the notions of *QoD* and *flexible validity intervals* are used to manage the freshness. When overloaded, update periods of some sensor data can be relaxed within the specified range of QoD to reduce the update workload, if necessary. However, sensor data are always maintained fresh in terms of flexible validity intervals. Therefore, the age of sensor data is always bounded. According to the performance study, QMF shows satisfying stringent QoS requirements for a wide range of workloads and access patterns. QMF achieves a near-zero miss ratio and perfect freshness even given dynamic workloads and data access patterns.

Imprecise computation techniques [63] provide means for achieving graceful degradation during transient overloads by trading off resource needs for the quality of a requested service. Imprecise computation and feedback control scheduling have been used for QoS management of RTDBs [9–11]. In this approach, the notion of imprecise computation is applied on transactions as well as data, that is, data objects are allowed to deviate, to a certain degree, from their corresponding values in the external environment. Although on RTDB models an external environment that changes continuously, the values of data objects that are slightly different in age or in precision are often interchangeable as consistent read data for user transactions. The time it takes to update a data object alone introduces a time delay, which means that the value of the data object cannot be the same as the corresponding real-world value at all times. This means that during transient overloads the system skips update transactions with values similar to the values stored in the database. To measure data imprecision the notion of data error, denoted de_i , is used, which gives an indication of how much the value of a data object d_i stored in the RTDB deviates from the corresponding real-world value given by the latest arrived transaction updating d_i . A user transaction T_j is discarded if the data error of the data object d_i to be updated by T_j is less than or equal to the maximum data error mde (i.e., $de_i \leq mde$). If mde increases, more update transactions are discarded, degrading the precision of

the data. Imprecision at transaction level can be expressed in terms of certainty, accuracy, and specificity [115]. Certainty refers to the probability of a result to be correct, accuracy refers to the degree of accuracy of a value returned by an algorithm (typically through a bound on the difference from the exact solution), and specificity reflects the level of detail of the result. The imprecision of the result of a user transaction T_i , denoted the transaction error te_i , increases as the resource available for the transaction decreases.

24.4.2 QoS Metrics

It is important to note that almost all the work on QoS in RTDBs is not aiming to provide hard guarantees. Instead, the main focus is on soft real-time applications in which infrequent deadline misses/temporal consistency violations can be tolerated, if neither of them exceeds the limits specified in the QoS requirements.

Two major performance metrics usually considered for QoS specification are the following:

- *User transaction deadline miss ratio*: In a QoS specification, a database administrator can specify the target deadline miss ratio that can be tolerated for a specific real-time application.
- *Data freshness*: We categorize data freshness into *database freshness* and *perceived freshness*. Database freshness is the ratio of fresh data to the entire temporal data in a database. Perceived freshness is the ratio of fresh data accessed to the total data accessed by timely transactions. To measure the current perceived freshness, we exclusively consider timely transactions. This is because tardy transactions, which missed their deadlines, add no value to the system. Note that only the perceived freshness can be specified in the QoS requirement. A QoS-sensitive approach provides the perceived freshness guarantee while managing the database freshness internally (hidden to the users) [47].

Long-term performance metrics such as the ones listed above are not sufficient for performance specification of dynamic systems in which the system performance can be widely time-varying. For that reason, transient performance metrics such as overshoot and settling time, adopted from control theory, have been considered as performance metrics for real-time data services [29,65,74]. Similar transient performance metrics are proposed in Ref. 80 to capture the responsiveness of adaptive resource allocation in real-time systems.

- *Overshoot* is the worst-case system performance in the transient system state (e.g., the highest miss ratio in the transient state). Overshoot is an important metric because a high transient miss ratio can cause serious implications in several applications such as robots and e-commerce.
- *Settling time* is the time for the transient overshoot to decay and reach the steady-state performance. The settling time represents how fast the system can recover from overload. This metric has also been called reaction time or recovery time.

24.4.3 QoS Management in Distributed Data Services

Providing QoS guarantees for data services in a distributed environment is a challenging task. The presence of multiple sites in distributed environments raises issues that are not present in centralized systems. The transaction workloads in distributed RTDBs may not be balanced and the transaction access patterns may be time-varying and skewed. The work in Ref. 104 describes an algorithm that provides QoS guarantees for data services in distributed RTDBs with full replication of temporal data. The algorithm consists of feedback-based local controllers and heuristic global load balancers (GLBs) working at each site. The local controller controls the admission process of incoming transactions. The GLBs collect the performance data from different nodes and balance the system-wide workload.

The system architecture of one node is shown in Figure 24.1. The RTDB system consists of an admission controller, a scheduler, a transaction manager, and blocked transaction queues. A local system performance monitor, a local controller, and a GLB are added to the system for QoS management purpose. In the figure, the solid arrows represent the transaction flows in the system and the dotted arrows represent the performance and control information flows in the system.

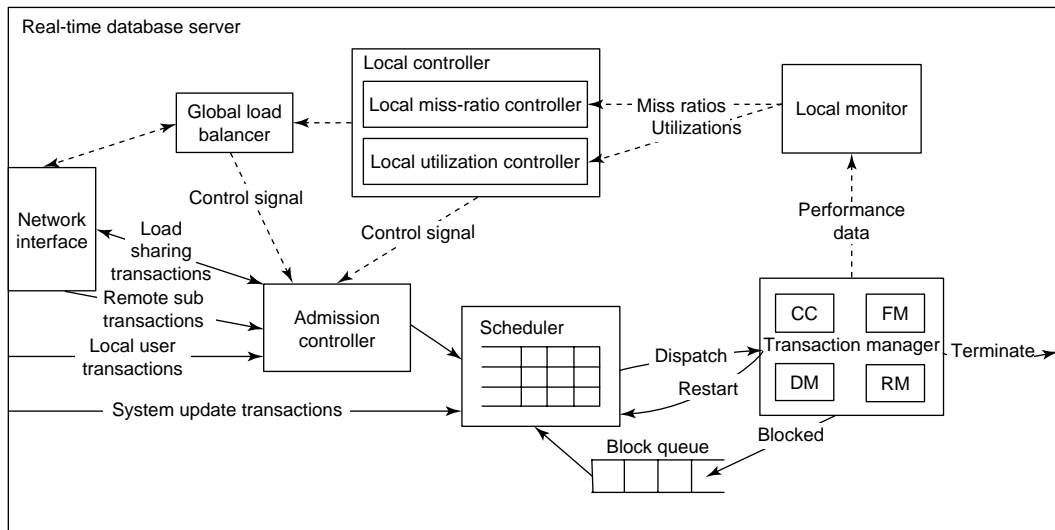


FIGURE 24.1 Real-time database system architecture for QoS management.

The system performance statistics are collected periodically by the transaction manager. At each sampling period, the local monitor samples the system performance data from the transaction manager and sends it to the local controller. The local miss-ratio controller and utilization controller generate local control signals based on the sampled local miss ratio and utilization data.

At each node, there is a local miss-ratio controller and a local utilization controller. The local miss-ratio controller takes the miss ratios from the latest sampling period, compares them with the QoS specification and computes the local miss-ratio control signal, which is used to adjust the target utilization at the next sampling period. To prevent underutilization, a utilization feedback loop is added. It is used to avoid a trivial solution in which all the miss-ratio requirements are satisfied due to underutilization. At each sampling period, the local utilization controller compares the utilization of the last period with the preset utilization threshold and generates the local utilization control signal. The local miss-ratio controller reduces the incoming transaction workload when the QoS specification is violated; the utilization increases the incoming transaction workload when the system is underutilized.

To balance the workload between the nodes and thus provide distributed QoS management, decentralized GLBs are used. GLBs sit at each node in the system, collaborating with each other for load balancing. At each sampling period, nodes in the system exchange their system performance data. The GLB at each node compares the system condition of different nodes. If a node is overloaded while some other nodes in the system are not, in the next period, the GLB at the overloaded node will send some transactions to the nodes that are not overloaded.

A more recent work extends the algorithm so that it scales to large distributed systems [105]. In that work, partial data replication and efficient replica management algorithms are studied because full replication is inefficient or impossible in large distributed systems.

Feedback-based QoS management in RTDBs is a new research direction with several key issues remaining, including fine-grained RTDB control, dynamic modeling, and adaptive control, integration with e-commerce/web applications. Considering an array of applications that can benefit from the work, the significance of QoS management in RTDBs will increase as the demand for real-time data services increases.

24.4.4 QoS Management for Data Streams

With the emergence of large wired and wireless sensor networks, many real-time applications need to operate on continuous unbounded data streams. At the same time, many of these systems have inherent

timing constraints. Real-time query processing on data streams is challenging for several reasons. First, the incoming data streams can be irregular with unpredictable peaks, which may overload the system. When the data arrival rate is too high, the system may become *CPU-constrained* and many queries will miss their deadlines. Second, the execution time of the queries depends not only on the data volume but also on the contents of the data streams. The queries registered in the databases (especially those with *join* operations) may have execution time that varies dramatically with the contents of their inputs. Third, in addition to being CPU-constrained, the system can also be *memory-constrained*. The total storage space required by all registered queries may exceed the size of the physical memory, and the system performance degrades drastically when the intermediate query results are forced to be swapped out of the physical memory.

Several data stream management systems support the *continuous query* model [12,14]. In the continuous query model, the query instances are triggered by the incoming streaming data. When a new tuple arrives, a new query instance is initiated and processed. The query results are updated with the results from the newest input. The continuous model performs well when the system workloads are moderate and the system has resources to finish all query instances triggered. However, because the number of query instances and system workloads depend directly on the unpredictable input, it is not appropriate for real-time applications that need predictable responses. Another drawback of the continuous query model is that the application cannot control how often each query is executed. In some applications, some queries are more interesting to the application and thus need to be executed more often. With continuous query model, this cannot be easily achieved. Statically allocating system resource to different queries does not work because the query execution time changes with the system input.

The *periodic query* (PQuery) model for real-time applications that need predictable query response has been proposed in Ref. 106. In the PQuery model, once a query is registered with the data stream management systems (DSMS) system, its query instances are initiated periodically by the system. Upon initialization, a query instance takes the snapshot from data streams as its input. The input does not change throughout the course of the instance execution even when there are newer data tuples in the data streams. Instead, the newly arrived data tuples are processed by the next query instance. Using this semantics, the query execution is not interrupted or aborted by the new incoming data. When an application gets the results of a periodic query instance, it knows that the results reflect the state of the system when the query instance is initiated. In the periodic query model, the query frequencies and deadlines are specified by applications and enforced by the system. Compared to the continuous query model, the workloads of the periodic queries are easier to estimate since at any given time, there are a fixed number of query instances in the system.

The quality of stream data processing can be specified using the following QoS metrics:

- *Data completeness*: The data completeness measures the percentage of incoming stream data that are used to compute the query results. Owing to the bursty nature of stream data, techniques such as *sampling* and *load shedding* [13,99] may be used to reduce the system workload in case of overload. The data completeness metric quantitatively measures the effects of these operations. The data completeness notion applies to both the *raw data* (i.e., the incoming sensor data streams) and the *intermediate results* (e.g., the output of a *join* operator).
- *Miss ratios*: Miss ratios measure the percentage of queries that are not finished within their deadlines.

Providing deadline guarantees for queries over dynamic data streams is a challenging problem due to bursty stream rates and time-varying contents. A prediction-based QoS management scheme for periodic queries over dynamic data streams is proposed in Ref. 107. It uses online *profiling* and *sampling* to estimate the cost of the queries on dynamic streams. The profiling process is used to calculate the average cost (CPU time) for each operator to process one data tuple. With all the input data and intermediate results in main memory, the execution time per data tuple tends to be predictable when certain conditions are met (e.g., low contention rates for the hash index). The sampling process is used to estimate the selectivity of the operators in a query. It is shown that for high-rate data streams, sampling is a viable and cost-effective way to estimate query selectivity. It is one of the first attempts that use online sampling to estimate and control query QoS. It is demonstrated by the experiments using real-time stream processing testbed that such

dynamic QoS management algorithms can handle the workload fluctuations very well and yield better overall system utility than existing approaches.

24.5 Data Services in Sensor Networks

Sensor networks are large-scale wireless networks that consist of numerous sensor and actuator nodes used to monitor and interact with physical environments [26,38]. From one perspective, sensor networks are similar to distributed database systems. They store environmental data on distributed nodes and respond to aperiodic and long-lived periodic queries [17,43,69]. Data interest can be preregistered to the sensor network so that the corresponding data are collected and transmitted only when needed. These specified interests are similar to views in traditional databases because they filter the data according to the application's data semantics and shield the overwhelming volume of raw data from applications [18,89].

Despite their similarity to conventional distributed RTDBs, sensor networks differ in the following important ways. First, individual sensors are small in size and have limited computing resources, while they also must operate for long periods of time in an unattended fashion. This makes power conservation an important concern in prolonging the lifetime of the system. In current sensor networks, the major source of power consumption is communication. To reduce unnecessary data transmission from each node, data collection and transmission in sensor networks are always initiated by subscriptions or queries. Second, any individual sensor is not reliable. Sensors can be damaged or die after consuming the energy in the battery. The wireless communication medium is also unreliable. Packets can collide or be lost. Because of these issues we must build trust on a group of sensor nodes instead of any single node. Previous research emphasizes reliable transmission of important data or control packets at the lower levels, but less emphasis is on the reliability of data semantics at the higher level [79]. Third, the large amount of sensed data produced in sensor networks necessitates in-network processing. If all raw data are sent to base stations for further processing, the volume and burstiness of the traffic may cause many collisions and contribute to significant power loss. To minimize unnecessary data transmission, intermediate nodes or nearby nodes work together to filter and aggregate data before the data arrives at the destination.

There have been several data service middleware research projects for the sensor network applications, including Cougar, Rutgers Dataman, SINA, SCADDS, and some virtual-machine-like designs [18,23,24,28,70,84,89]. In COUGER, sensor data are viewed as tables and query execution plans are developed and possibly optimized in the middleware. DSWare project [59] is more tailored to sensor networks, including supporting group-based decision, reliable data-centric storage, and implementing other approaches to improve the performance of real-time execution, reliability of aggregated results, and reduction of communication. SINA is a cluster-based middleware design, which focuses on the cooperation among sensors to conduct a task. Its extensive SQL-like primitives can be used to issue queries in sensor networks. However, it does not provide schemes to hide the faulty nature of both sensor operations and wireless communication. In SINA, it is the application layer that must provide robustness and reliability for data services. The real-time scheduling component and built-in real-time features of other service components make DSWare more suitable than SINA for real-time applications in wireless sensor networks.

Multisensor data fusion research focuses on solutions that fuse data from multiple sensors to provide more accurate estimation of the environment [44,76]. In mobile-agent-based data fusion approaches, software that aggregates sensor information are packed and dispatched as *mobile agents* to "hot" areas (e.g., the area where an event occurred) and work independently there. The software migrates among sensors in a cluster, collects observations, then infers the real situation [76]. This group-based approach makes use of consensus among a number of nearby sensors of the same type to increase the reliability of a single observation. The mobile-agent-based approach, however, leverages on the migration traffic of mobile agents and their appropriate processing at each sensor node in its routes. For instance, if a node in the route inserts wrong data or refuses to forward the mobile agents, the aggregation and subsequent analysis are untrustful.

Data aggregation is an important service provided by a sensor network because it can reduce the amount of data that is transmitted throughout the network. Many data aggregation techniques have been proposed in the research [25,37,54,75,83]. A key to good aggregation is to minimize data loss while also reducing

power usage throughout the network. Synopsis diffusion [71] is a framework for data aggregation that combines energy-efficient multipath routing with techniques that avoid counting the same data more than once. In Ref. 68, the authors make a case that data fusion does not always save energy, and that the cost of fusion should also be taken into account when deciding whether to perform data fusion in-network. They present an algorithm called Adaptive Fusion Steiner Tree that optimizes over the costs of data transmission and data fusion.

24.5.1 Real-Time Data Services in Sensor Networks

Sensor networks have inherent real-time properties. The environment that sensor networks interact with is usually dynamic and volatile. The sensor data usually has an absolute validity interval of time after which the data values may not be consistent with the real environment. Transmitting and processing “stale” data wastes communication resources and can result in wrong decisions based on the reported out-of-date data. Besides data freshness, often the data must also be sent to the destination by a deadline. To date, not much research has been performed on real-time data services in sensor networks.

Recently, there has been work toward developing infrastructure and protocols for real-time sensor networks [3,27,36,53,64,67,97]. A survey of the current work, and the research challenges involved with developing real-time sensor network communication protocols is presented in Ref. 97. Much of this work is devoted to providing some level of predictability to the lower-level protocols in the sensor network.

SPEED [36] is a communication protocol that provides real-time unicast, real-time area-multicast, and real-time area-anycastr. The protocol uses feedback control and nondeterministic geographic forwarding to maintain an acceptable overall delivery speed. In Ref. 67, a sensor network scheduling policy is presented, called Velocity Monotonic Scheduling, that assigns priority to data packets based on their deadline and their distance from their destination. The work described in Ref. 3 provides a mechanism for measuring the capacity of a sensor network to meet specified deadlines. This measure can be used to schedule network traffic as well as provide predictable data delivery in the sensor network.

There has been some work toward providing real-time data services in sensor networks. In Ref. 4, the degree of data aggregation is manipulated using a feedback control mechanism to limit the amount of latency in the delivery of the data. This work utilizes the real-time capacity measure described in Ref. 3 to provide a limit for acceptable network traffic.

JITS [64] is a Just-In-Time scheduling algorithm for efficiently scheduling the dissemination of real-time data packets across a sensor network. The algorithm introduces nonuniform delays at intermediate nodes across an end-to-end path to account for the higher expected traffic on nodes that are closer to the destination, or sink nodes. This has been shown to improve deadline miss ratio in the sensor network over similar techniques.

An interesting twist on scheduling data collection in wireless sensor networks is taken in Ref. 91. In this work, mobile nodes in the sensor network do the collection of data from stationary nodes. The mobile node visits the stationary nodes to upload the data before the buffer is full. The challenge presented in the paper is how to schedule these mobile nodes in such a way as to minimize data loss due to buffer overflow. While the problem itself is NP-complete, the authors present several heuristic algorithms for potential solutions. The minimum weighted sum first heuristic was shown to be a promising solution.

24.6 Mobile Real-Time Databases

Recent advances in wireless communications technology have made mobile information services a valuable mechanism for applications to share and distribute data. Applications like real-time traffic information systems, and mobile internet stock trading systems have requirements for timely transactions and valid data. However, the resource constraints of mobile systems make it difficult to meet timing constraints. These constraints include low bandwidth, unreliable wireless networks, and frequent disconnections. The research described in this section attempts to apply RTDB solutions and solve the resource constraint issues for mobile RTDB applications.

The work in Ref. 48 describes a mobile RTDB system model that consists of a number of fixed and mobile hosts. Each mobile host has a fixed host that acts as a mobile support station (MSS) and coordinates the operations of the transactions submitted by that mobile host. A mobile transaction in this model can execute by the submitting mobile host, and/or at one or more fixed hosts, and has a deadline by which it must be completed. The mobile transaction can be executed either in a single request message to the fixed network, or as a sequence of messages for Read and Write operations to the fixed network, thus placing majority of the execution of the transaction either on the fixed host (former case) or on the mobile host (latter case). The authors performed various simulations to test these execution strategies, as well as other chosen techniques. The results of these simulations indicate that as the number of hosts increases, the fraction of transactions that satisfy their deadlines decreases. Further, transactions that are executed on the fixed host, consistently perform better than those that are executed on the mobile host.

The work in Ref. 57 extends an RTDB concurrency control technique to handle the specific characteristics of a mobile RTDB. The protocols in this work address two main issues: the cost of resolving data conflicts, and the impact of the mobile network. The system model consists of both a mobile network and a reliable wired network. The concurrency control mechanisms described in this chapter are based on the HP-2PL protocol [1]. The Distributed High-Priority Two-Phase Locking protocol (DHP-2PL) uses a transaction restart mechanism to resolve data conflicts. However, it attempts to minimize the restarting of transactions that access data on mobile hosts by allowing a committing transaction to hold a lock until it has finished the commit procedure. Priority inversion is minimized by increasing the priority of the committing transaction higher than all executing transactions. To handle the problem of disconnection of a mobile transaction, a cautious waiting scheme is used in which a higher-priority transaction is restarted by a lower-priority transaction if the higher-priority transaction is suspected of having been disconnected.

The idea of transaction shipping is presented in Ref. 57 to reduce communication costs of executing mobile transactions. Recall that in Ref. 48, it was shown that transactions that are executed on the fixed host perform better than those executed on the mobile host. Transaction shipping sends the entire transaction to the fixed database server. The main issue of concern is how to identify the execution path and the required data objects of a transaction ahead of time. A preanalysis approach is suggested in which the set of operations of the transaction is identified, and the precedence relationships of the operations are determined. Given this information, the database server can execute the transaction without the need to communicate with the mobile host on which it originated.

In Ref. 56, a multiversion data model is adopted to allow for more flexibility in resolving data conflicts. This provides a looser correctness criterion based on relative consistency, and therefore increases data availability in the mobile system. In this model, new updates of data items are generated by mobile hosts, and a specific fixed host maintains a backup copy of, not only the current version, but also of previous copies of the data item. An image transaction is generated at a fixed host, called the anchor host, when a mobile transaction is received. Its job is to get its required data items to the anchor host in time for the mobile transaction to access it. Techniques similar to those in Ref. 57 are used for preanalysis. When the mobile transaction is executed, the relative consistency of the data items is checked to ensure that they all reflect the same temporal state of the data. After the mobile transaction accesses a version of a particular data item, all other data items are checked for relative consistency. If at any point, such a version cannot be found, the transaction is restarted, and the next latest version of its first accessed data item will be selected.

An enhanced multiversion data model is presented in Ref. 58 for data broadcasts in time-constrained mobile systems. The broadcast cycle is made smaller by relaxing the requirement to broadcast every data item in every cycle. For each version of a data item in the broadcast cycle, this technique keeps track of the cycle number when the version was created, and the cycle number of the latest version of the data item. This allows for the distance between versions to be recorded. This work also proposes an on-demand broadcast for minimizing missing deadlines. After the broadcast of all data items that are scheduled, the system examines the on-demand buffer, containing any request that occurred during the broadcast cycle. These requests are sorted by deadline, and processed in this order. The amount of time spent processing broadcast data versus on-demand data is a tunable parameter in the system.

As mentioned above, mobile RTDB/systems must contend with power constraints on the mobile devices that generate and access data. The work in Ref. 30 presents transaction management that reduces missed deadlines while balancing the amount of energy consumed by mobile hosts in the network. The system model in this work is a mobile *ad hoc* network (MANET) in which only mobile hosts exist, not fixed hosts. Each mobile host can be in one of three energy modes: active (executing as usual), doze (CPU runs at lower rate and can receive messages), and sleep (CPU and communication are suspended). Small mobile hosts (SMHs) have highly constrained memory, storage, and power capabilities. Large mobile hosts (LMHs) have more resources than SMHs. Each LMH maintains a global schema that describes the status of the other LMHs. Each LMH broadcasts its ID, position, and energy level to all other hosts. When an SMH starts a transaction, it is sent in its entirety to an LMH to be processed. To schedule the transactions, an energy-efficient modification of Least Slack Time scheduling is proposed that considers the problem of disconnection. Highest priority is given to the transaction with the smallest slack time. In case of equal slack time, higher priority is given to the transaction whose requestor has the lower energy level.

RTMonitor [55] is a real-time data monitoring system that uses mobile agents to adaptively monitor real-time data based on the urgency of the requests and on-system workload. The target application for this system is traffic navigation. The agents use Adaptive PUSH OR PULL (APoP) to maintain the temporal consistency of the data. To manage the traffic system, local traffic graphs connect to neighboring local graphs through external nodes. The system consists of two types of agents: stationary and mobile. Stationary agents manage the traffic graphs. The mobile agents serve the client navigation requests. In the traffic system, traffic sensors collect traffic data for road segments and send the data whenever there is a significant change in a value. In the APoP approach used here, the monitoring period is defined based on the urgency of the request, the system status, and the dynamic properties of the traffic data. When a request is received, a GraphAgent is created to find the shortest path. This is accomplished by computing a path on the local graph of the originating region, a virtual path to the destination region, and a local path from an external node to the destination in the destination region. To keep the calculated path up to date, the system initially uses a PUSH mode for data updates. When the workload at a server becomes heavy, some less-critical requests will be switched to a PULL mode. The QueryAgent for a client generates a pull based on the slack time and the dynamic properties of the traffic data.

24.7 Dissemination of Dynamic Web Data

Recent studies have shown that an increasing fraction of the data on the World Wide Web is time-varying (i.e., changes frequently). Examples of such data include sports information, news, and financial information such as stock prices. The coherency requirements associated with a data item depends on the nature of the item and user tolerances. To illustrate, a user may be willing to receive sports and news information that may be out-of-sync by a few minutes with respect to the server, but may desire to have stronger coherency requirements for data items such as stock prices. A user who is interested in changes of more than a dollar for a particular stock price need not be notified of smaller intermediate changes.

An important issue in the dissemination of such time-varying web data is the maintenance of temporal coherency. In the case of servers adhering to the HTTP protocol, clients need to frequently pull the data based on the dynamics of the data and a user's coherency requirements. In contrast, servers that possess push capability maintain state information pertaining to clients and push only those changes that are of interest to a user. These two canonical techniques have complementary properties with respect to the level of temporal coherency maintained, communication overheads, state space overheads, and loss of coherency owing to (server) failures.

In Ref. 16, combinations of push- and pull-based techniques are proposed to achieve the best features of both approaches. The combined techniques tailor the dissemination of data from servers to clients based on (i) the capabilities and load at servers and proxies and (ii) clients' coherency requirements.

Time-varying web data are frequently used for online decision making, and in many scenarios, such decision making involves multiple time-varying data items, possibly from multiple independent sources. Examples include a user tracking a portfolio of stocks and making decisions based on the net value of the portfolio. Observe that computing the overall value of the portfolio at any instant requires up-to-date values of each stock in the portfolio. In some scenarios, users might be holding these stocks in different (brokerage) accounts and might be using a third-party aggregator, such as yodlee.com, to provide them a unified view of all their investments. The third-party aggregator assembles a unified view of the portfolio by periodically gathering and refreshing information from each individual account (source), all of which may be independent of one another. If the user is interested in certain events (such as a notification every time the value of the portfolio increases or decreases by \$1000), then the aggregator needs to periodically and intelligently refresh the value of each stock price to determine when these user-specified thresholds are reached. More formally, the aggregator runs a continuous query over the portfolio by intelligently refreshing the components of the portfolio from their sources to determine when user-specified thresholds are exceeded. We refer to such queries as Continuous Threshold Queries (CTQs).

A technique presented in Ref. 15 allows time-varying data items, like stock prices, to be cached either at a proxy or by financial websites. The approach provides mechanisms to maintain coherency of cached data with the original servers and tracks the total value of a portfolio with a specified accuracy. The technique is pull-based so it can be used with the HTTP protocol. The coherency of an individual data item is maintained by a proxy through a Time-To-Refresh (TTR) value. This specifies the next time the proxy should get a fresh value for the data from the server. This TTR value can change dynamically if the update rate of the data item on the server varies. The key feature of this research is the Coherency requirement Balancing Algorithm (CBA). This algorithm updates the coherency requirement for each data item in the portfolio based on how the data changes, as well as the relative contribution of that data item to the entire portfolio. The results of this work will allow a user to specify a continuous query that will inform her when the net worth of her stock portfolio changes more than a specified threshold.

The use of proxies to cache data from a portfolio can be useful to deliver coherent, time-constrained data to a user. However, failure of the proxy could result in a loss of coherency and could cause the user to miss vital information. The work presented in Ref. 87 describes a system in which a set of repositories cooperate with each other as well as the data sources. The resulting peer-to-peer network pushes important data changes to users in a coherency-preserving, and resilient manner. To maintain consistency among the cooperating repositories, an efficient dissemination tree is created to propagate changes from data sources to the repositories. Two dissemination algorithms are developed to maintain the coherency of the data throughout the tree. The first is a Repository-Based, distributed approach in which repositories decide when to distribute updates to its descendents in the tree based on specified coherency requirements. In the second, source-based (centralized), approach, the data source maintains a list of the coherency requirements for the data items. When a new update occurs, the source determines if any coherency requirements have been violated, and disseminates the update to interested repositories through the tree.

When multiple users request data from a system such as is described above, a decision must be made regarding, which repository will best serve the request of the user. In Ref. 88, this client assignment problem is addressed. The solution that is presented is based on available solutions to the matching problem in combinatorial optimization. Experimental results indicate that in update intensive situations, better fidelity can be achieved when some clients receive data at a coherence lower than their desired requirement. Based on this result, the authors have developed techniques that quickly adapt delivered coherence with respect to the changes in the system.

24.8 Conclusion

Research in the field of real-time data services has evolved a great deal over the relatively short time of its existence. In the early days of the 1980s, much of the research concentrated on RTDBs and examining how

to add real-time support to traditional databases. Much of this work involved developing new scheduling and concurrency control techniques that extended existing techniques to allow RTDBs to provide timely transactions and temporally valid data. This has included techniques to relax traditional atomicity, consistency, isolation, durability (ACID) properties of databases, and managing QoS provided by the database to allow for timing constraints to be met.

More recently, the focus of this research has been on applying the lessons learned from RTDB research toward providing real-time functionality to emerging applications that have data delivery and data management needs, such as sensor networks, dissemination of dynamic web data, and mobile databases. While some of the RTDB results can apply to these newer applications, there are issues such as power conservation, mobility, and the very large scale of the Internet, that pose new challenges.

The future of research in the field of real-time data services will depend on the continuing evolution of ideas. Research in this field should continue to follow cutting-edge applications and apply existing techniques to them, as well as develop new ones when needed.

References

1. R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3): 513–560, 1992.
2. R. Abbott and H. Garcia-Molina. Scheduling I/O Requests with Deadlines: A Performance Evaluation. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.
3. T. Abdelzaher, S. Prabh, and R. Kiran. On Real-Time Capacity Limits of Multihop Wireless Sensor Networks. In *Proceedings of the 2004 Real-Time Systems Symposium*, Dec. 2004.
4. T. Abdelzaher, T. He, and J. Stankovic. Feedback Control of Data Aggregation in Sensor Networks. In *Proceedings of the 2004 Conference on Decision Control*, Dec. 2004.
5. B. Adelberg, H. Garcia-Molina, and B. Kao, Applying Update Streams in a Soft Real-Time Database System. In *Proceedings of the 1995 ACM SIGMOD*, pp. 245–256, 1995.
6. B. Adelberg, B. Kao, and H. Garcia-Molina. Database Support for Efficiently Maintaining Derived Data. In *ETDB*, 1996.
7. Q. Ahmed and S. Vrbsky. Triggered Updates for Temporal Consistency in Real-Time Databases. *Real-Time Systems Journal*, 19: 209–243, 2000.
8. Q. Ahmed and S. Vrbsky. Triggered Updates for Temporal Consistency in Real-Time Databases. *Real-Time Systems Journal*, 19(3): 205–208, 2000.
9. M. Amirijoo, J. Hansson, and S. H. Son. Algorithms for Managing QoS for Real-Time Data Services Using Imprecise Computation. In *Proceedings of the Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, 2003.
10. M. Amirijoo, J. Hansson, and S. H. Son. Error-Driven QoS Management in Imprecise Real-Time Databases. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.
11. M. Amirijoo, J. Hansson, and S. H. Son. Specification and Management of QoS in Imprecise Real-Time Databases. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, 2003.
12. A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In *Stanford University Tech. Report*, 2003.
13. B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE Conference on Data Engineering (ICDE 2004)*, Mar. 2004.
14. H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal Special Issue on Data Stream Processing*, 2004.
15. M. Bhide, K. Ramamritham, and P. Shenoy. Efficiently Maintaining Stock Portfolios Up-To-Date on the Web. In *Proceedings of the 12th International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE'02)*, 2002.

16. M. Bhide, P. Deolasse, A. Katker, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers Special Issue on Quality of Service*, 51: 652–668, June 2002.
17. P. Bonnet, J. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Personal Communication Magazine*, 7(5): 10–15, 2000.
18. P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the 2nd International Conference on Mobile Data Management*, Hong Kong, 2001.
19. A. P. Buchmann, D. R. McCarthy, M. Chu, and U. Dayal. Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control. In *Proceedings of the Conference on Data Engineering*, 1989.
20. M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS Resource Scheduling. In *Proceedings of the 15th VLDB Conference*, pp. 397–410, Aug. 1989.
21. D. Chen and A. K. Mok. SRDE-Application of Data Similarity to Process Control. In *The 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, Dec. 1999.
22. S. Chen, J. Stankovic, J. Kurose, and D. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *Real-Time Systems*, Sept. 1991.
23. Cougar Project. www.cs.cornell.edu/database/cougar.
24. Dataman Project. www.cs.rutgers.edu/dataman.
25. K. Dasgupta, K. Kalpakis, and P. Namjoshi. An Efficient Clustering-Based Heuristic for Data Gathering and Aggregation in Sensor Networks. *Wireless Communications and Networking*, 3, 2003.
26. D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th Annual International Conference on Mobile Computing and Networks*, Seattle, WA, 1999.
27. A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, Dec. 2005.
28. J. Feng, F. Koushanfar, and M. Potkonjak. System-Architectures for Sensor Networks: Issues, Alternatives, and Directions. In *Proceedings of the 20th International Conference on Computer Design*, Freiburg, Germany, 2002.
29. G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems (3rd edition)*. Addison-Wesley, 1994.
30. L. Gruenwald and S. Banik. A Power-Aware Technique to Manage Real-Time Database Transactions in Mobile Ad-Hoc Networks. In *Proceedings of the 12th International Conference on Database and Expert Systems Applications (DEXA'01)*, Munich, Germany, Sept. 2001.
31. T. Gustafsson and J. Hansson. Dynamic On-Demand Updating of Data in Real-Time Database Systems. In *Proceedings of the ACM Symposium on Applied Computing*, New York, pp. 846–853, 2004.
32. T. Gustafsson and J. Hansson. Data Management in Real-Time Systems: A Case of On-Demand Updates in Vehicle Control Systems, In *Proceedings of the 10th IEEE RTAS*, pp. 182–191, 2004.
33. J. R. Haritsa, M. J. Carey, and M. Livny. On Being Optimistic about Real-Time Constraints. In *Proceedings of ACM PODS*, 1990.
34. J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.
35. J. Haritsa, K. Ramamritham, and R. Gupta. The PROMPT Real-Time Commit Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2): 160–181, 2000.
36. T. He, J. A. Stankovic, C. Lu, and T. Abdelzaher. SPEED: A Stateless Protocol for Real-Time Communication in Ad Hoc Sensor Networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, Providence, RI, 2003.
37. T. He, B. M. Blum, J. A. Stankovic, and T. Abdelzaher. AIDA: Adaptive Application-independent data aggregation in wireless sensor networks. *ACM Transactions on Embedded Computing Systems*, 3(2), 2004.

38. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.
39. S. Ho, T. Kuo, and A. K. Mok. Similarity-Based Load Adjustment for Static Real-Time Transaction Systems. In *18th Real-Time Systems Symposium*, 1997.
40. J. Huang, J. A. Stankovic, K. Ramamritham, D. Towsley, and B. Purimetla. On Using Priority Inheritance in Real-Time Databases. *Special Issue of Real-Time Systems Journal*, 4(3), 1992.
41. J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental Evaluation of Real-Time Transaction Processing. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1989.
42. J. Huang, J. A. Stankovic, K. Ramamritham and D. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of the Conference on Very Large Data Bases*, Sept. 1991.
43. C. Jaikaeo, C. Srisathapornphat, and C.-C. Shen. Querying and Tasking in Sensor Networks. In *Proceedings of SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V)*, Orlando, FL, 2000.
44. D. Jayasimha, S. Iengar, and R. Kashyap. Information Integration and Synchronization in Distributed Sensor Networks. *IEEE Transactions on Systems, Man and Cybernetics*, 21(5): 1032–1043, 1991.
45. A. K. Jha, M. Xiong, and K. Ramamritham, Mutual Consistency in Real-Time Databases. In *RTSS2006: The 27th IEEE International Real-Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.
46. K. D. Kang, S. H. Son, and J. A. Stankovic. Managing Deadline Miss Ratio and Sensor Data Freshness in Real Databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(7), 2004.
47. K. D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. In *The 14th Euromicro Conference on Real-Time Systems*, June 2002.
48. E. Kayan and O. Ulusoy. An Evaluation of Real-Time Transaction Management Issues in Mobile Database Systems. *The Computer Journal* (Special Issue on Mobile Computing), 42(6), 1999.
49. S. Kim, S. Son, and J. Stankovic. Performance Evaluation on a Real-Time Database. In *IEEE Real-Time Technology and Applications Symposium*, 2002.
50. W. Kim and J. Srivastava. Enhancing Real-Time DBMS Performance with Multi-Version Data and Priority-Based Disk Scheduling, In *Proceedings of the Real-Time Systems Symposium*, pp. 222–231, Dec. 1991.
51. T. Kuo and A. K. Mok, SSP: A Semantics-Based Protocol for Real-Time Data Access, In *IEEE 14th Real-Time Systems Symposium*, Dec. 1993.
52. T. Kuo and A. K. Mok. Real-Time Data Semantics and Similarity-Based Concurrency Control. *IEEE Transactions on Computers*, 49(11): 1241–1254, 2000.
53. A. Koubaa and M. Alves. A Two-Tiered Architecture for Real-Time Communications in Large-Scale Wireless Sensor Networks: Research Challenges. In *Proceedings of the 17th Euromicro Conference On Real-Time Systems (ECRTS 05)*, July 2005.
54. A. Kulakov and D. Davcev. Intelligent Data Aggregation in Sensor Networks Using Artificial Neural-Networks Algorithms. In *Technical Proceedings of the 2005 NSTI Nanotechnology Conference and Trade Show*, Vol. 3, 2005.
55. K.-Y. Lam, A. Kwan, and K. Ramamritham. RTMonitor: Real-Time Data Monitoring Using Mobile Agent Technologies. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.
56. K.-Y. Lam, G. Li, and T.-W. Kuo. A Multi-Version Data Model for Executing Real-Time Transactions in a Mobile Environment. In *Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 2001.
57. K.-Y. Lam, T.-W. Kuo, W.-H. Tsang, and G. Law. Concurrency Control in Mobile Distributed Real-Time Database Systems. *Information Systems*, 25(4): 261–322, 2000.

58. H.-W. Leung, J. Yuen, K.-Y. Lam, and E. Chan. Enhanced Multi-Version Data Broadcast Schemes for Time-Constrained Mobile Computing Systems. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA'02)*, Sept. 2002.
59. S. Li, Y. Lin, S. H. Son, J. Stankovic, and Y. Wei. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. *Telecommunication Systems*, 26: 351–368, 2004.
60. C. L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1), 1973.
61. Y. Lin and S. H. Son. Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order. In *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.
62. K. J. Lin, S. Natarajan, and J. W. S. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Real-Time System Symposium*, Dec. 1987.
63. J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise Computations. In *Proceedings of the IEEE*, Vol. 82, Jan. 1994.
64. K. Liu, N. Abu-Ghazaleh, and K. Kang. JiTS: Just-in-Time Scheduling for Real-Time Sensor Data Dissemination. *Proceedings of the Fourth International Conference on Pervasive Computing and Communications*, Mar. 2006.
65. C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Real-Time Systems Symposium*, Orlando, FL, Nov. 2000.
66. C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23, 2002.
67. C. Lu, B. Blum, T. Abdelzaher, J. A. Stankovic, and T. He. RAP: A Real-Time Communication Architecture for Large-Scale Wireless Sensor Networks. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, 2002.
68. H. Luo, J. Luo, Y. Liu, and S. Das. Energy Efficient Routing with Adaptive Data Fusion in Sensor Networks. In *Proceedings of the 3rd ACM/SIGMOBILE International Workshop on Foundations of Mobile Computing*, 2005.
69. S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, 2002.
70. F. Mattern, K. Römer, and O. Kasten. Middleware Challenges for Wireless Sensor Networks. In *ACM SIGMOBILE Mobile Computing and Communication Review (MC2R)*, 2002.
71. S. Nath and P. Gibbons. Synopsis Diffusion for Robust Aggregation in Sensor Networks. *Technical Report IRP-TR-03-08, Intel Research*, 2003.
72. D. Nyström, A. Tesanovic, C. Norström, J. Hansson, and N.-E. Bänkestad. Data Management Issues in Vehicle Control Systems: A Case Study. In *Proceedings of the 16th ECRTS*, pp. 249–256, 2002.
73. H. Pang, M. J. Carey, and M. Livny. Multiclass Query Scheduling in Real-Time Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(4), 1995.
74. C. L. Phillips and H. T. Nagle. *Digital Control System Analysis and Design (3rd edition)*. Prentice-Hall, 1995.
75. B. Przydatek, D. Song, and A. Perrig. SIA: Secure Information Aggregation in Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, 2003.
76. H. Qi, X. Wang, S. S. Iyengar, and K. Chakrabarty. Multisensor Data Fusion in Distributed Sensor Networks Using Mobile Agents. In *Proceedings of 5th International Conference on Information Fusion*, Annapolis, MD, 2001.
77. R. Sivasankaran, K. Ramamritham, and J. A. Stankovic. System Failure and Recovery. In *Real-Time Database Systems: Architecture and Techniques*, K.-Y. Lam and T.-W. Kuo (eds.), Kluwer Academic Publishers, pp. 109–124, 2000.
78. K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases*, 1: 199–226, 1993.

79. S. Ratnasamy, D. Estrin, R. Govindan, B. Karp, S. Shenker, L. Yin, and F. Yu. Data-Centric Storage in Sensor networks. In *Proceedings of the 1st Workshop on Sensor Networks and Applications*, Atlantic, GA, 2002.
80. D. Rosu, K. Schwan, S. Yalmanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. In *IEEE Real-Time Systems Symposium*, Dec. 1997.
81. R. Gurulingesh, N. Sharma, K. Ramamritham, and S. Malewar. Efficient Real-Time Support for Automotive Applications: A Case Study. *RTCSA 2005*, Sydney, Australia, Aug. 2006.
82. R. Rastogi, S. Seshadri, J. Bohannon, D. Leinbaugh, A. Silberschatz, and S. Sudarshan. Improving Predictability of Transaction Execution Times in Real-Time Databases. *Real-Time Systems Journal* 19(3): 205–208, 2000.
83. O. Savas, M. Alanyali, and V. Saligrama. Randomized Sequential Algorithms for Data Aggregation in Sensor Networks. In *Proceedings of the 40th Annual Conference on Information Sciences and Systems*, Mar. 2006.
84. SCADDS: Scalable Coordination Architectures for Deeply Distributed Systems. www.isi.edu/scadds.
85. L. Sha, R. Rajkumar, and J. P. Lehoczky. Concurrency Control for Distributed Real-Time Databases. *ACM SIGMOD Record*, Mar. 1988.
86. L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In *IEEE Transactions on Computers*, 39(9): 1175–1185, 1990.
87. S. Shah and K. Ramamritham. Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers. *IEEE Transactions on Knowledge and Data Engineering*, 16(7), 2004.
88. S. Shah, K. Ramamritham, and C. Ravishankar. Client Assignment in Content Dissemination Networks for Dynamic Data. In *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, 2005.
89. C.-C. Shen, C. Srisathapornphat, and C. Jaikao. Sensor Information Networking Architecture and Applications. *IEEE Personal Communication Magazine*, 8(4): 52–59, 2001.
90. L. Shu, S. H. Son, and J. Stankovic. Achieving Bounded and Predictable Recovery Using Real-Time Logging. *The Computer Journal*, 47(4): 373–394, 2004.
91. A. Somasundara, A. Ramamoorthy, and M. Srivastava. Mobile Element Scheduling for Efficient Data Collection in Wireless Sensor Networks with Dynamic Deadlines. In *IEEE Real-Time Systems Symposium 2004*, Lisbon, Portugal, Dec. 2004.
92. S. H. Son, Y. Lin, and R. P. Cook. Concurrency Control in Real-Time Database Systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. van Tilborg and G. Koobeds (eds.), Kluwer Academic Publishers, pp. 185–202, 1991.
93. S. H. Son. Using Replication for High Performance Database Support in Distributed Real-Time Systems. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, 1987.
94. S. H. Son and S. Kouloumbis. A Real-Time Synchronization Scheme for Replicated Data in Distributed Database Systems. *Information Systems*, 18(6), 1993.
95. N. Soparkar et al. Adaptive Commitment for Real-Time Distributed Transactions. *Technical Report TR 92–15*, Department of Computer Science, University of Texas, Austin, 1992.
96. X. Song and J. W. S. Liu. Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5): 786–796, 1995.
97. J. Stankovic, T. Abdelzaher, C. Lu, L. Sha, and J. Hou. Real-Time Communication and Coordination in Embedded Sensor Networks. *Proceedings of the IEEE*, 91(7): 1002–1022, 2003.
98. J. A. Stankovic, K. Ramamritham, and D. Towsley. Scheduling in Real-Time Transaction Systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. van Tilborg and G. Koob (eds.), Kluwer Academic Publishers, pp. 157–184, 1991.
99. N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *International Conference on Very Large Data Bases*, Berlin, Germany, 2003.
100. B. Urgaonkar, A. G. Ninan, M. S. Raunak, P. Shenoy, and K. Ramamritham. Maintaining Mutual Consistency for Cached Web Objects. In *International Conference on Distributed Computing Systems*, pp. 371–380, Apr. 16–19, 2001.

101. V. Ranadive. *The Power to Predict: How Real-Time Businesses Anticipate Customer Needs, Create Opportunities and Beat the Competition*, McGraw-Hill, 2006.
102. S. Vrbsky. *APPROXIMATE: A Query Processor that Produces Monotonically Improving Approximate Answers*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1993.
103. S. V. Vrbsky and K. J. Lin. Recovering Imprecise Computations with Real-Time Constraints. *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pp. 185–193, Oct. 1988.
104. Y. Wei, S. Son, J. Stankovic, and K. D. Kang. QoS Management in Replicated Real-Time Databases. In *24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003.
105. Y. Wei, A. Aslinger, S. Son, and J. Stankovic. ORDER: A Dynamic Replication Algorithm for Periodic Transactions in Distributed Real-Time Databases. In *10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Aug. 2004.
106. Y. Wei, S. Son, and J. Stankovic. RTStream: Real-Time Query Processing for Data Streams. In *IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, Apr. 2006.
107. Y. Wei, V. Prasad, S. Son, and J. Stankovic. Prediction-Based QoS Management for Real-Time Data Streams. In *IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, Dec. 2006.
108. M. Xiong, S. Han, and K. Lam. A Deferrable Scheduling Algorithm for Real-Time Transactions Maintaining Data Freshness. In *IEEE Real-Time Systems Symposium*, 2005.
109. M. Xiong, S. Han, and D. Chen. Deferrable Scheduling for Temporal Consistency: Schedulability Analysis and Overhead Reduction. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.
110. M. Xiong and K. Ramamritham. Deriving Deadlines and Periods for Real-Time Update Transactions. *IEEE Transactions on Computers*, 53(5): 567–583, 2004.
111. M. Xiong, K. Ramamritham, J. Haritsa, and J. A. Stankovic. Mirror: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases. *Information Systems: An International Journal*, 27(4): 277–297, 2002.
112. M. Xiong, K. Ramamritham, J. A. Stankovic, D. Towsley, and R. Sivasankaran. Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics. *IEEE Transactions on Knowledge and Data Engineering*, 14(5): 1155–1166, 2002.
113. Y. Yoon. *Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems*. Ph.D. thesis, Korea Advanced Institute of Science and Technology, May 1994.
114. H. Zhou and F. Jahanian. Real-Time Primary-Backup (RTPB) Replication with Temporal Consistency Guarantees. In *Proceedings of ICDCS*, May 1998.
115. S. Zilberstein and S. J. Russell. Optimal Composition of Real-Time Systems. *Artificial Intelligence*, 82(1–2): 181–213, 1996.

25

Real-Time Data Distribution

Angela Uvarov Frolov
University of Rhode Island

Lisa Cingiser DiPippo
University of Rhode Island

Victor Fay-Wolfe
University of Rhode Island

25.1	Introduction to Real-Time Data Distribution	25-1
25.2	Real-Time Data Distribution Problem Space	25-1
	System Characteristics • Real-Time Characteristics • Data Characteristics	
25.3	Approaches to Real-Time Data Distribution	25-5
	Point-to-Point RTDD • Real-Time Databases • Real-Time CORBA • OMG Data Distribution Service • Current Research in Real-Time Data Distribution	
25.4	Conclusion	25-16

25.1 Introduction to Real-Time Data Distribution

Many different applications, such as electronic stock trading, industrial automation, telecommunications, and military command and control, require shared data among their various distributed components. Further, these applications also require that data be available within a timely manner and be temporally consistent. To satisfy these and many other specific application requirements, real-time data distribution (RTDD) is necessary. RTDD is the transfer of data from one source to one or more destinations within a deterministic time frame, regardless of the method and the timescale. Currently, there are various approaches to RTDD in practice and in research. This chapter describes the range of problems associated with this type of system by providing an RTDD problem space. The chapter goes on to discuss some existing solutions, and shows which areas of the problem space each of the solution addresses.

25.2 Real-Time Data Distribution Problem Space

In systems that require RTDD, there are some common characteristics, such as data must be at the right place at the right time and it must be temporally consistent. There are also other system-specific characteristics that vary from one system to another. This section identifies these system-specific characteristics and groups them into three types:

1. System characteristics
2. Real-time characteristics; and
3. Data characteristics [1]

These characteristic categories are further classified into specific characteristics, each of which can take on one or more values. Figure 25.1 illustrates this concept in an RTDD problem space taxonomy. This section describes each of the characteristics of an RTDD problem, and discusses the values that it may take.

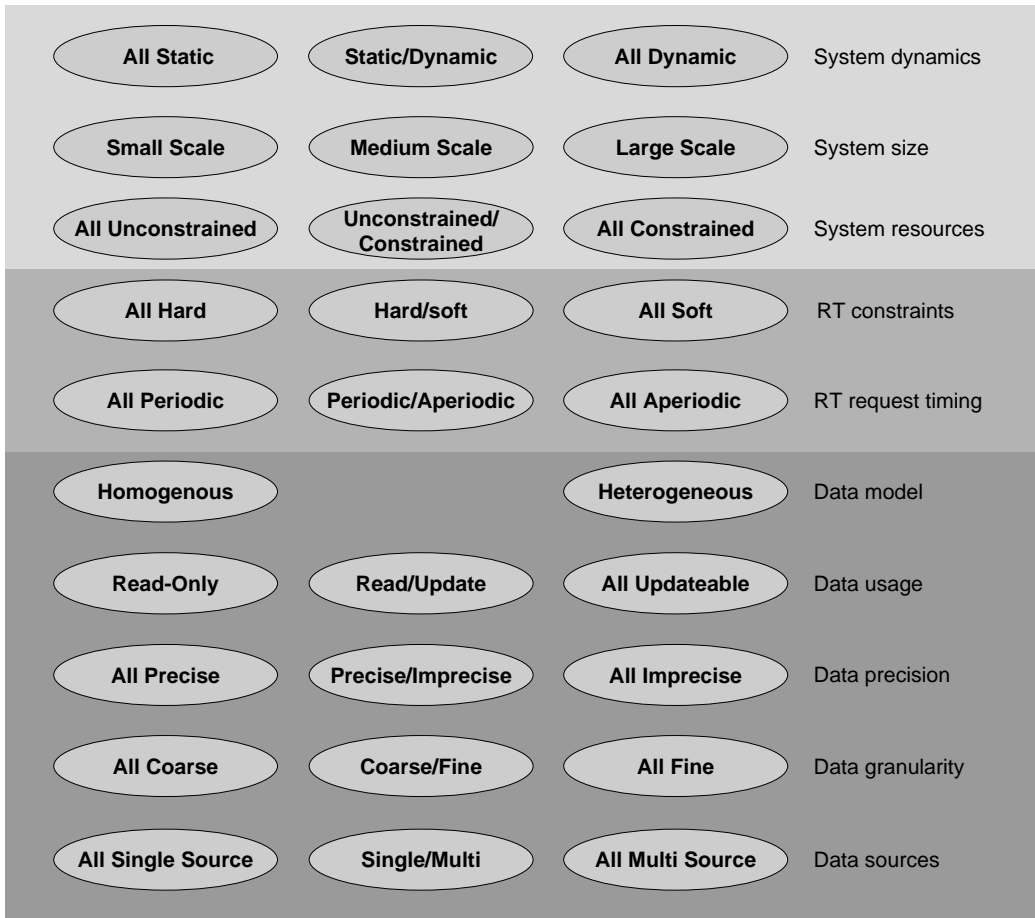


FIGURE 25.1 RTDD problem space.

25.2.1 System Characteristics

The first layer in the RTDD problem space taxonomy represents system characteristics. These are overall characteristics of the system that define the general problem.

System dynamics. Some systems that require RTDD are *static*, that is the system requirements are fully known in advance and do not change. Therefore, the needs for data distribution can be specified and analyzed prior to system execution to ensure that data needed at any particular time and location are delivered on time. For example, an industrial automated system may be static if all of its parts are known at the design stage and do not change during the system's lifetime.

A *dynamic* system is one in which the system specification cannot be predicted before execution time. Requests for data can be made at any time during execution, and the system must be able to either estimate the data needs or react to dynamic requests to meet the timing requirements. An example of this type of system is an electronic stock trading system in which a client's request for a particular stock price can come at any time during the system's execution.

There are also some systems with a combination of static and dynamic elements. That is, there may be some requirements that remain the same throughout the execution of the system, while others change or are unpredictable. For instance, in an air traffic control system the requirement for how often to provide wind-speed information may remain the same, while the requirement to receive aircraft information may change based on environmental conditions.

System size. The size of a system can vary from a single node to thousands of nodes. The size can also affect how much data are being stored, how many suppliers of data there are, and how many consumers there are in the system. An example of a small system that requires an RTDD is a patient-monitoring system in a hospital. Data about the vital condition of a patient can be sent to several doctors or other hospital systems. A much larger system might involve thousands of cell phone users requesting stock prices or sports scores from a bank of servers that have the information.

System resources. The resources of an RTDD system may have various constraints on their operation. For example, a system of small, battery-operated wireless sensors that collect and distribute data about certain environmental conditions has power constraints on each of the nodes, as well as communication constraints based on the strength of the wireless radios. Other systems, such as an embedded network of wired computers aboard a submarine, have fewer physical constraints on the system.

25.2.2 Real-Time Characteristics

The next layer in the taxonomy illustrated in Figure 25.1 represents real-time characteristics that involve the timing of the system (periodic versus aperiodic), as well as the consequences of missing a specified constraint (hard versus soft).

Real-time constraints. Real-time constraints define the system's behavior in case of missing specified deadlines. In a *hard* real-time system, if a deadline is missed the system fails. For example, in an industrial automated system, if data are not delivered on time the system cannot proceed leading to further failures down the line. Data itself can have hard deadlines as well. In a submarine contact tracking system, the tracks have to be updated from the sensors within a specified time or they will be considered old, or temporally inconsistent.

A system has *soft* real-time constraints if missing the deadlines causes a degradation of value to the system, but not a failure. For example, a high-availability telecom system may specify that it will deliver data on time a certain percentage of the time. In a soft real-time system, some temporal inconsistency in the data may be tolerated as long as it is corrected in a timely manner.

There are systems with combination of soft and hard real-time constraints. For instance, in a submarine the contact tracking will have hard deadlines, while showing video to the crew will have soft deadlines. Obviously, the crew could tolerate some frozen video frames while the tracking system is following a potential enemy ship.

Real-time request timing. Requests for data in a real-time distributed system can be made *periodically* or *sporadically* (aperiodically). When a periodic request is made, the data are expected to be delivered at the specified frequency, or else the delivery deadline is considered to be missed. Periodic requests usually occur once requesting the delivery of the data regularly for many periods. The requests can be halted or the period can change, but while a request is intact the data should be delivered every period. An example of a system that may require periodic data delivery is a submarine contact tracking system. To ensure that the system is representing the real-world contact sufficiently, the system requires that the new real-time data be updated frequently enough to represent a smooth transition from one contact data point to the next.

Sporadic requests for RTDD occur when a client requires data on a one-time basis or based on events rather than time periods. For example, in the stock trading system described above, a client may specify that it requires a stock price whenever its value changes by 5% or more.

25.2.3 Data Characteristics

The last layer in the taxonomy represents characteristics that involve the kind of data being shared in a real-time system, and how it is used within the system.

Data model. The data model used within an RTDD system can be *homogeneous*, where each participant is expected to use the same data model, or *heterogeneous*, where such an expectation is not required. A homogeneous data model makes the sharing of data across the distributed system simpler because no conversion is necessary. However, it may be too restrictive in a large-scale system to expect that various applications that share data will use the same data model. A heterogeneous data model is more flexible

that various applications that are developed at different times, with different requirements can share data without restricting the way in which their own data are stored. However, this type of system may require conversions from one data model to another, or the use of an agreed-upon intermediary representation. For example, in a system that provides data sharing among a coalition of forces from various nations, it is unreasonable to expect the data to be stored in a homogeneous model. For such a system the various data models are stored in their own formats, and a data transfer language, like XML, is used to interpret the data that are shared among the various components.

Data usage. Many RTDD systems only require that data be disseminated to various clients within timing constraints, but do not expect the data to be updated and written back to the source. These types of systems, which we call *read-only*, do not necessarily require any concurrency control among the distributed clients because they treat the data as their own copies. As long as each client receives data that are temporally consistent, and the data are received within specified timing constraints, the distribution of the data is successful. For example, in an electronic stock trading system, the stock prices are distributed to requesting clients, but the clients do not update them.

However, there are applications in which distributed consumers of the data also update the data and *write it back* to the source or to other copies of the data. For example, in a submarine contact tracking system, the track data, synthesized from sensor information, may be distributed to various locations so that it can be used and viewed by other applications and human users. Some of these applications may receive data from other sources that would allow it to make refinements to the track data. In this case, the track data may need to be updated, not only at the source but possibly also at any other copies of the data. This kind of data usage is much more complicated than read-only data usage because more than one application may wish to update the original data, and therefore concurrency control among these updates is required. If copies of the data also have to be updated then the system is even more complex. The fact that all of the data must be kept both logically and temporally consistent with each other adds to the complexity of the problem.

Data precision. Some real-time systems require that the data they receive be absolutely *precise* and consistent with the real-world entities that are being modeled. In such systems, the concurrency control mechanism that maintains the integrity of the data will not allow multiple updates, even if the locking that might be required will cause deadlines to be missed. Further, the data must be temporally consistent at all times—never becoming older than a specified age. For instance, a command and control system that is closely tracking a target will want to be sure that the data it receives are precise.

However, some applications allow for the possibility of some *imprecision* in the value as well as the age of the data to allow for more flexibility in meeting other constraints, such as temporal consistency constraints. For example, a client of an electronic stock trading system may be willing to receive data that are slightly old or slightly imprecise, if it means paying a lower fee. As long as the amount of imprecision is bounded, the client can analyze the data with the imprecision in mind.

Data granularity. The amount or granularity of data that is distributed to clients can vary from entire tree structures to single atomic elements. In the case of an object-oriented system, entire objects can be distributed to various locations for use by clients. In fact, groupings or hierarchies of objects can be distributed all together. These are *coarse-grained* distributions of data. On the contrary, a *finer grain* of data can be distributed, such as individual attribute values, or return values of object methods. The granularity of the data being distributed depends largely on the applications that are using the data as well as how the data are being used. For example, in a system in which the distributed data are being updated and written back, it might make sense to employ the smallest granularity possible so that large portions of data are not locked owing to concurrency control.

However, when groups of objects are closely related, it may make sense to distribute them together as a group. This way the values of the related data are more likely to be relatively temporally consistent with each other and therefore more valuable to the requesting client.

Data source. In many real-time systems, real-time data comes from sensors that provide the most recent version of the data. In many cases, the sensor transaction is the *single source* of update for the data. However, it is also possible for the data to be updated by *multiple sources*. For example, in a target detection system,

various sensors may be used to update the data depending upon which is the closest or most reliable. In this case, it may be possible that both sensors try to update the data simultaneously, requiring concurrency control to ensure the integrity of the data.

All the characteristics described above form the definition of a problem space for RTDD. The better understanding of the problem involved in RTDD will eventually lead to a better solution considering these characteristics will help to provide good, useful solutions in current and future applications.

25.3 Approaches to Real-Time Data Distribution

This section discusses different approaches to RTDD. For each approach, there is a description of the areas within the problem space illustrated in Figure 25.1 that the solution addresses.

25.3.1 Point-to-Point RTDD

Various types of point-to-point data distribution are commonly found in existing systems. While these are widely used methods, they may not address RTDD very well for many applications because of the requirement to know the exact receiver of the data. These solutions may not scale well in large systems. This section briefly discusses several of these types of solutions.

Client-server. The client-server communication model is a central idea in network computing. Many business applications existing today use this model. In this model, a server awaits requests from clients, who access data via queries. In some of these applications, clients can read and update information on the server. For example, in an online banking application, the customer accesses his account and may make payments, changing the value in the database. In others, clients are only allowed to read the information. For example, a web browser is a client that can fetch information from a web server and display web page content for the user.

The client-server approach to RTDD is a very broad approach. Therefore, the area within the RTDD problem space that can be addressed depends greatly on the application that is being served. A client-server model can address both static and dynamic systems. Most applications that use this approach are dynamic, but in a system in which all requests for data are known *a priori*, a client-server approach can work. The client-server model can work in a system of any size. However, to provide real-time support for data distribution a larger size can become unwieldy. Further, if there are a lot of requests for the same data, it becomes difficult for a single server to respond in a timely fashion. Thus, multiple servers might be necessary, which makes the system more complex.

In the client-server model, clients can access data both to read and to update it. The typical client-server model does not specify any allowance for imprecise data. However, a specialized implementation can build imprecision into a particular application. The granularity of the data depends upon the service provided by a server. For example, in an object-oriented server, the data that is provided are at the level of the return type for the methods that are available on the server object(s). Typically, in a client-server model, there is a single source for any data that are available. If more than one server provides the data it usually originates at the same source (single sensor in the environment).

Broadcast and multicast. Broadcast is a communication model in which data or signal is transmitted to anyone and everyone in particular service area or network. For instance, in the wireless network of the portable devices (e.g., cell phones, PDA, and palmtops) information, such as electronic newspapers, weather and traffic information, stock and sports tickers, entertainment delivery, is broadcast to all devices in the network. The difference between broadcast and multicast is that in a multicast communication model data is transmitted to a select list of recipients and not to everyone in the network. One of the approaches to transmit data in such a way is through broadcast disks [2–4]. Broadcast disks provide an abstraction for asymmetric communication that treats a stream of data as a storage device.

The target systems for broadcast or multicast RTDD are dynamic because it is difficult to have knowledge of all the requirements of the recipients *a priori*. Thus, the constraints that a broadcast or multicast system

can have are usually soft. For the supplier of broadcast or multicast data to efficiently serve all requestors the data model must be homogeneous.

This is a read-only approach because data is sent out from the supplier and received by any receiver in the network. There is no data sent back upstream to the original sender. Broadcast data can be precise or imprecise depending upon the requirements of the receivers. As long as the receiver is aware of the level of imprecision, it can be factored into how the data is used. Broadcast data can be at any level of granularity. However, owing to the widespread use of the network in a broadcast, smaller, more fine-grained data may be more efficient to send. Typically, in a broadcast model there is a single source for any data that is available.

Streaming. Streaming is a technology in which data are transferred from a server to a client and are processed in a steady and continuous stream, without the need to be stored in a client's space. Typical applications that use streaming for RTDD are video, and continuous backup copying to a storage medium.

Systems that use streaming for RTDD are usually dynamic because clients can connect at any time to receive the streaming data. Also, it is difficult to know *a priori* what the data needs of an application will be. The size of the system can be quite large. In an high definition television (HDTV) application, thousands of users view the stream from a source. Since clients do not need to store data, they can operate with some limited resources. For example, a receiver of a video stream does not require much storage space. All it requires is enough resources to display the streaming video frames as they arrive. Streaming systems typically have soft real-time constraints, such as minimum frame rate on a video stream.

Data transfer can be periodic as well as sporadic. In a video streaming application, the frames are transferred periodically so that they can be displayed on the receiving node with a constant frame rate. For an application in which data are streamed for continuous backup, the rate of the stream is not as important and can be more sporadic. The data model of a streaming application is typically homogeneous. This way, the sender can stream data, such as video frames, and the receiver knows how to process it.

Similar to broadcast, streaming RTDD is a read-only approach because data are sent out from the sender and processed immediately by the receiver. There is no data sent back upstream to the original sender. For best quality streaming data should be precise. However, in a real-time scenario, in which processing time is highly constrained, it is sometimes necessary to reduce the quality of the streaming data. For example, it may be necessary to reduce the frequency of the data stream to meet other timing constraints. This will degrade the quality of a video stream, but it may still be understandable by a human eye.

The granularity of the data in a stream depends upon the application. The receiving node has to process the data upon receipt, so it would make sense to use the smallest granularity possible. Typically, in a streaming model there is a single data source.

25.3.2 Real-Time Databases

A real-time database (RTDB) can be considered an extension to a traditional database. It has all traditional database features but must also be able to express and maintain timing constraints, such as deadlines, earliest and latest start time on transactions and timing constraints, such as temporal consistency on data itself. An RTDB consists of real-time objects representing the entities of real world and updated by sensor transactions. To be coherent with the state of environment the real-time object must be refreshed by a transaction before it becomes invalid, that is within its temporal validity interval, whose length is usually application dependent. There are many applications that require real-time data, and with advances in networking all of them are not necessarily located on the same node as the RTDB; therefore, they require the real-time data to be distributed to them.

Since there is an entire chapter in this handbook dedicated to different aspects of RTDBs, this section concentrates on the areas of the RTDD problem space that can be addressed by an RTDB.

An RTDB can handle both static and dynamic systems. In a static system, all data and transactions are known and scheduled *a priori*. A dynamic RTDB system has the ability to adapt to transactions and changes in the environment. Small- to medium-scale systems can be serviced with a central database.

For larger-scale systems, a distributed database is usually used. Computational resources are usually constrained by the timing constraints imposed by the applications that use an RTDB. Further resource constraints exist in an RTDB that involves mobile, wireless nodes.

Transactions in an RTDB can be hard or soft and can be periodic or sporadic. The data model is typically homogeneous. Although, in larger systems that combine various RTDBs into a single virtual RTDB, it may be possible to have a heterogeneous data model. In this case, middleware is typically used to synthesize the various models.

Most RTDB applications expect precise data. However, imprecise data may be acceptable if the other timing constraints can be met. For example, it may be necessary to allow two transactions to access a real-time object simultaneously to meet the timing constraints of the transactions. As long as the consequences of this concurrent access are accounted for, the application can still use the data.

25.3.3 Real-Time CORBA

The Common Object Request Broker Architecture (CORBA), developed by The Object Management Group (OMG), is a standard for object-oriented middleware for distributed systems [5]. The goal of middleware is to facilitate seamless client-server interactions in a distributed system.

CORBA is designed to allow a programmer to construct object-oriented programs without regard to traditional object boundaries, such as address spaces or location of the object, in a distributed system. This means that a client program should be able to invoke a method on a server object whether the object is in the client's address space or located on a remote node in a distributed system. The CORBA standard defines a framework to allow communication among applications in a distributed system regardless of platform or programming language differences.

To address the needs of broad real-time applications, the OMG Real-Time Special Interest Group (RT SIG) defined the standards for the Real-Time CORBA (RT CORBA) [6,7]. The goal of RT CORBA is to provide a standard for a CORBA ORB to deal with the expression and enforcement of real-time constraints on executions to support end-to-end predictability in a system. To provide the special capabilities to special applications without restricting non-real-time development, RT CORBA is positioned as a separate extension to CORBA 2.2 and constitutes an optional, additional compliance point.

RT CORBA provides several services that can distribute real-time data. This section describes these services, points out the distinctions among them, and discusses the areas of the RTDD problem space that is addressed by each.

Real-time event service. A standard CORBA request results in the synchronous execution of an operation by an object, during which data defined by the operation is communicated between client and server (client-server communication model). Though it is widely used, this model has some limitations. First, for the request to be successful, both the client and the server must be available. Second, a client has to wait until the server finishes processing and returns the result, and third, a client invokes a single-target object on a particular server. Although, there were attempts to relax these limitations with time-independent invocations (TIIs) [8] and asynchronous methods invocations (AMIs) [9], there are some scenarios where even more decoupled communication between objects is required.

To address this type of communication, the OMG issued a specification for CORBA Object Service (COS) Event Service [10]. The Event Service decouples the communication between objects by providing for them two roles: the supplier and the consumer. Event data are communicated between the supplier and the consumer by a standard CORBA call.

The specification describes two approaches to initiate communication between supplier and consumer: push model and pull model (see Figure 25.2). In the push model, the supplier is an initiator of communication. It pushes data to the Event Channel and then the Event Channel pushes data to the consumer. In the pull model, the consumer initiates the connection. It requests data from Event Channel, and the Event Channel in turns pulls data from the supplier. At the heart of the Event Service is the Event Channel, which plays the role of intermediary between the objects producing data or being changed (suppliers) and the objects interested in data or in knowing about changes (consumers).

The Event Channel appears to suppliers as a proxy consumer and appears to consumers as a proxy supplier. It is the Event Channel that frees suppliers and consumers from limitations of standard synchronous CORBA calls and provides flexible communication among multiple suppliers and consumers.

While the CORBA Event Service provides a flexible model for asynchronous communication between objects, its specification lacks important features required by various real-time applications. The work in Ref. 11 describes the design and performance of a Real-Time Event Service (RT ES) that was developed as a part of the TAO project (The ACE ORB) at Washington University [12]. This extension is based on enhancements to the push model of the CORBA Event Service and supports real-time event scheduling and dispatching, periodic rate-based event processing, and efficient event filtering and correlation. Figure 25.3 presents the TAO's RT ES architecture and collaborations in it.

While in this architecture the Event Channel plays the same role as it does in CORBA Event Service, it consists of several processing modules, each of which encapsulates independent tasks of the channel. The TAO's RT ES Consumer and Supplier Proxy interfaces extend the standard COS ProxyPushConsumer and ProxyPushSupplier so that suppliers can specify the types of events they provide, and consumers can register with the Event Channel their execution dependencies. The Subscription and Filtering module allows consumers to subscribe for particular subset of events, and then the channel uses this subscription to filter supplier events to forward them only to interested consumers. In COS Events Service, all events from suppliers are delivered to all consumers. The RT Event Channel provides three types of filtering: (1) supplier-based filtering, where consumers register for and receive events only from particular suppliers; (2) type-based filtering, where consumers register for and receive events only of a particular type; and (3) combined supplier/type-based filtering. The Event Correlation module allows consumers to specify what kind of events to occur before Event Channel can proceed. The dispatching module determines when events should be delivered to consumers and pushes them accordingly. The architecture of RT ES allows

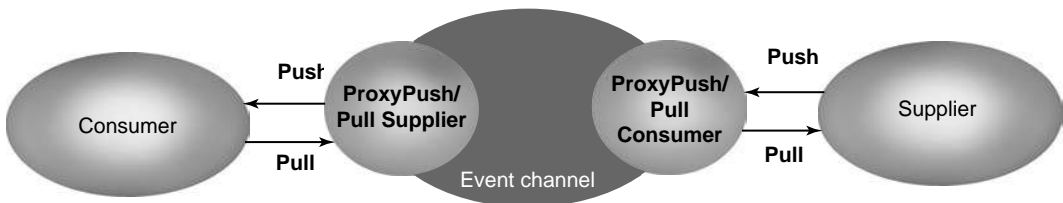


FIGURE 25.2 Event Channel communication models.

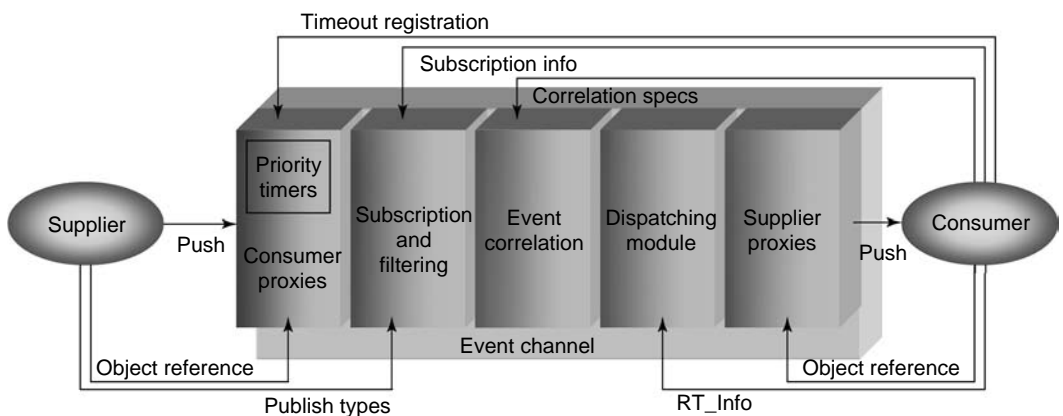


FIGURE 25.3 Collaborations in the RT ES Architecture.

the service to be configured in many ways, since its modules can be added, removed, or modified without changes to other modules to address requirements of different applications.

TAO's RT ES can handle static and dynamic systems of various sizes. The computational resources in the system are constrained by the timing constraints imposed by the application. The service can provide support for both hard and soft real-time applications. The publish-subscribe nature of the RT ES allows processing of both periodic and aperiodic types of requests.

The data model for TAO's RT ES is homogeneous, since the consumers use the same data model as the suppliers. Only the suppliers can change their data and the consumers are just readers; therefore, the data usage is read-only. Since the service allows the suppliers to register for the "whole" event, and not a part of it, only coarse granularity is supported. On the contrary, if we consider an event as a single piece of information it can be considered fine. Then, if a subscriber wants to impose some event dependencies and get a combination of several events, that can be considered coarse. The RT ES allows supplier/type based filtering, therefore it can address the multiple sources of data.

Real-time notification service. Although there are some implementations of the Event Service that address the limitations of a standard OMG Event Service, such as a lack of quality of service (QoS) policies and event filtering capabilities [11,13], these tied users to these specific products, thus preventing them from using other middleware providers. Therefore, to address these limitations in a standardized way OMG issued a specification for CORBA Notification Service that extends the OMG Event Service capabilities [14].

According to the specification, the Notification Service preserves all the Event Service's interfaces and functionality making it backward compatible with the Event Service. In addition, the Notification Service has new, extended interfaces that allow definition of structured events. These events consist of header and body, which in turn are defined by name/value pairs holding the information associated with events. This enables consumers to use filters to specify which events they would like to receive and to attach different QoS properties to the event. The specification also defines standard interfaces to control QoS properties on a per-channel and per-proxy (supplier and consumer) basis. One more important feature of the Notification Service is the ability to share subscription information between event channel and consumers. This allows consumers to find out about and subscribe for newly available event types, and provides the suppliers with information regarding the event types for which there is an interest.

Despite providing QoS properties that are useful for many distributed applications, such as timeliness, reliability, priority, and ordering, the original Notification Service lacks features important for wide range of real-time applications. Among these features are end-to-end predictability of event delivery and use of RT CORBA scheduling and priority assignment. To overcome these issues, OMG issued a request for proposal (RFP) for RT Notification Service [15]. The mandatory requirements for the new service were defined as follows:

- Limitation of the complexity of filters. For instance, by restricting the number of filters or specifying the message length.
- Identification of the subset of RT Notification functionality used for achieving predictable resource management and timeliness predictability.
- Definition of schedulable entities for both Client-propagated and Server-declared models.
- Support of the end-to-end priority propagation as defined in RT CORBA.
- Provision of means to set RT QoS parameters end-to-end, and at channel, connection, proxy, and message levels.
- Definition of interfaces for resource management.
- Definition of interfaces between RT Notification and RT CORBA Scheduling services.

The TAO RT Notification Service [16] is built with the standard CORBA Notification Service as its basis. By using RT CORBA features, the service maintains events' priorities along the whole pass from suppliers to consumers, thus providing end-to-end priority-aware event propagation. Through extension to the proxy interfaces, so that ThreadPool/Lanes can be applied per event channel, TAO's RT Notification

Service can allow administration of concurrency within the service. Providing these extensions have solved the following problems:

- To limit complexity of filters and to make sure that filters do not run too long, timeouts were used.
- To ensure end-to-end priority preservation, RT CORBA CLIENT_PROPAGATED priority model was used.
- To support RT thread pools, QoS properties can be applied to POA at multiple levels (event channel, admin, and proxy).
- To optimize the processing of an event, the Reader/Writer lock is used to the Subscription LookupTable.
- To minimize the context switching between Consumer and Supplier Proxies, the ORB's collocation mechanism was employed.

Since the RT Notification Service can be considered as an extended, more sophisticated version of RT ES, the problem space addressed by it is quite similar to that of the RT ES. Although the assumed RT Notification is supposed to work in both static and dynamic environments, the current version of TAO RT Notification can support only static systems (integration of dynamic features is the part of the future work). The RT Notification Service uses more sophisticated filters, so the level of data granularity can be finer than that of the RT ES.

CORBA audio/video streaming service. A stream can be defined as a continuous sequence of digitally encoded signals presenting information flowing from one object to another. Although streams may represent any type of data flowing between objects, with the advances in networking and computer processing power, a special type of streaming and one more aspect of data distribution has emerged—video streaming. To prevent this technology from going into customized and isolated multimedia systems and to develop applications in a more standardized way using reusable components and general interfaces, OMG issued CORBA audio/video (A/V) Stream specification [17].

This specification focuses on A/V applications with QoS constraints. It defines an architecture for building distributed multimedia applications through well-defined modules and interfaces that provide stream initialization and control, so that data suppliers and consumers that are built independently are still able to communicate. The specification provides a means for different data transfer protocols, so that developers are not restricted and can choose the protocol more suitable for the specific application needs. It also addresses the issues of QoS, interoperability, and security.

The CORBA A/V Stream Service architecture is presented in Figure 25.4. The stream here is represented as a flow within two flow endpoints. Streams may consist of multiple flows (data, audio, and video) that can be simultaneously or independently started or terminated by flow endpoints. Each stream that may have several flow endpoints is terminated by a stream endpoint. Each stream end-point consists of three logical entities: a data source or sink, a stream adaptor responsible for sending and receiving frames over network, and a stream interface control object providing an interface definition language (IDL) interface. Control and management objects are responsible for stream establishment and control, through defined interfaces with stream interface control objects.

TAO's CORBA A/V Stream Service [18] is built based on the OMG specification. With compliance to the specification, the TAO's A/V Stream Service provides flexibility:

- In stream endpoint creation strategies by decoupling the behavior of stream components from strategies governing their creation.
- In data transfer protocol by providing a flexible mechanism within the stream establishment components that permit applications to use various APIs (sockets and TLI) and Internet protocols (TCP, UDP, RTP, or ATM).
- In stream control interfaces, and in managing states of supplier and consumer.

TAO's A/V Stream Service also provides a uniform API interface for different flow protocols, and uniform interface for full (where flow interfaces are exposed) and light (where only stream and stream

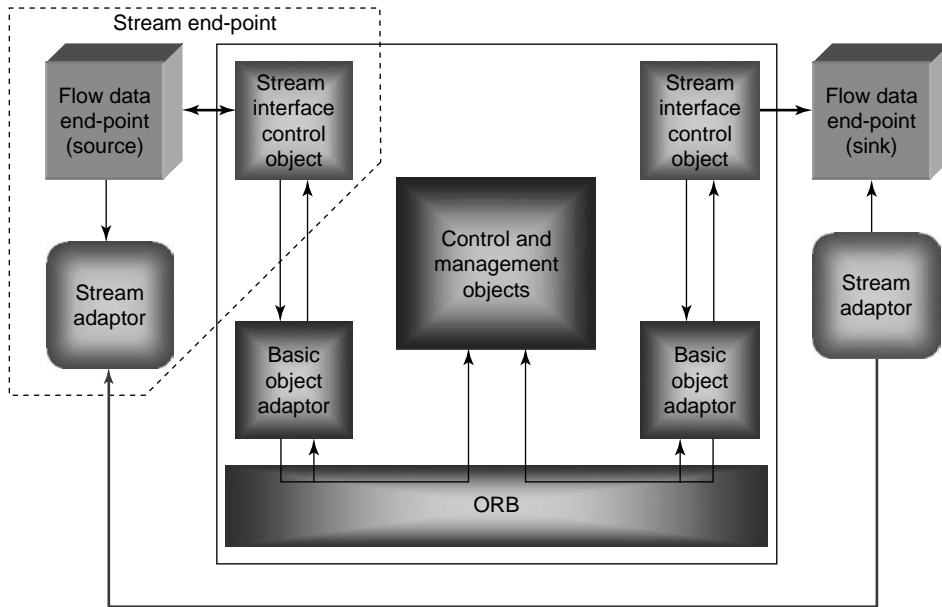


FIGURE 25.4 CORBA A/V Streaming Service architecture.

endpoint interfaces are exposed) profiles. In addition, TAO's service provides multipoint-to-multipoint binding, even though the specification leaves it as the responsibility of application developers.

The areas of the RTDD problem space addressed by this mechanism are the same as the general streaming solution that was described in Section 25.3.1.

25.3.4 OMG Data Distribution Service

Many distributed applications use a data-centric message exchange as their communication style. Some of them may consist of hundreds or even thousands of interacting components. For coordination of interactions in such large-scale distributed systems, the data-centric publish-subscribe (DCPS) model has become very popular. This model is built on the concept of a "global data space" that is accessible to all interacting components. Some components are publishers; they contribute information to the global data space. Other components are subscribers that desire access to parts of the global space. Each time a publisher provides new information to the data space, the data distribution service (DDS) notifies all interested subscribers.

To provide a common application interface that clearly defines DDS for such DCPS systems, OMG issued a specification for a DDS [19]. This specification describes two levels of interfaces:

- **DCPS Layer**—A lower DCPS layer is responsible for efficient delivery of the proper information to the proper recipients. That is, it allows publishers to identify the data objects they want to publish and then provide their values. It allows subscribers to identify data object they are interested in and then access them. And, finally it allows applications to define topics and to attach type information to them, to create publishers and subscribers entities, to attach QoS to all these entities and make all of them operate.
- **Data local reconstruction layer (DLRL)**—A higher (optional) layer may be built on top of DCPS. Its purpose is to provide more direct access to exchanged data, thus allowing easier integration into the application level.

Figure 25.5 presents the conceptual view of DCPS layer of DDS. The data flows with the aid of a Publisher and a DataWriter on the sending side, and a Subscriber and a DataReader on the receiving side.

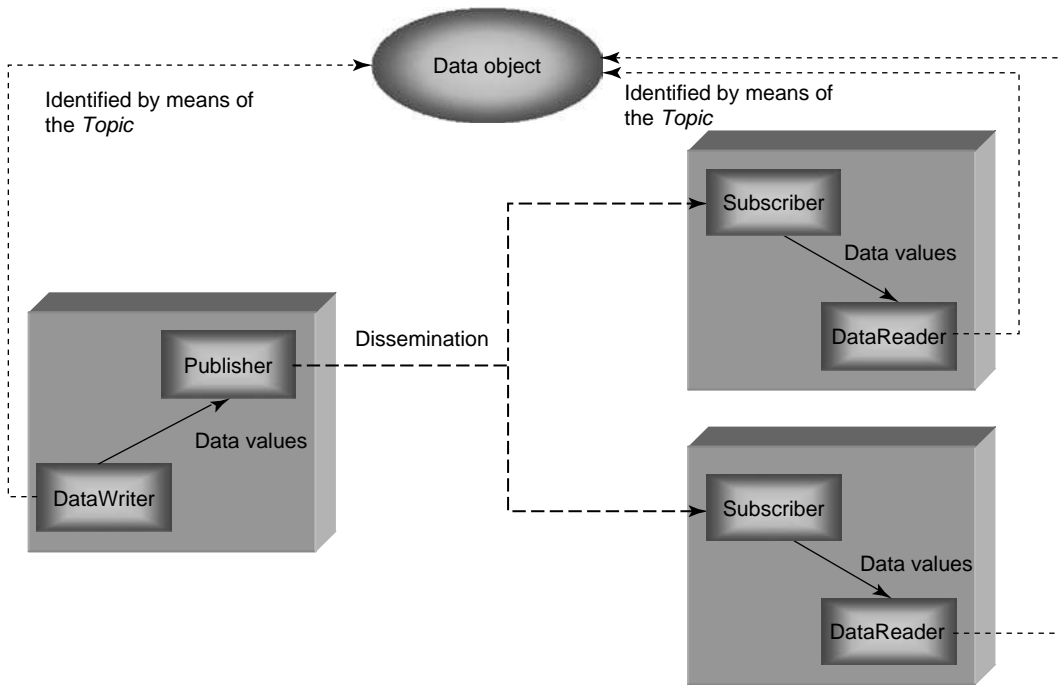


FIGURE 25.5 OMG DDS overview.

The DataWriter object is the application's communication interface to Publisher, used to communicate the existence of data and its values. When a DataWriter provides new data object values to the Publisher, it is the responsibility of the Publisher to distribute data according to its own QoS or the QoS attached to the corresponding DataWriter. The Publisher can act on behalf of one or more DataWriters, while each DataWriter is associated with a single data object.

A Subscriber is the object responsible for receiving data and making it in accordance with its QoS available to the application. To access the data, the application uses DataReader object.

A Topic is an object that conceptually fits between publications and subscriptions. It is uniquely identified by name, data type, and QoS related to the data. To reduce the number of topics, the combination of a Topic and a key is used. That is, if the Topic is a unique data object (for instance, pressure or temperature sensors) no key is necessary, but if the Topic represents a set of related data objects (e.g., a number of drifting buoys monitored by GPS to measure velocity and direction of a current), each of them can be distinguished by a key. The key may be represented by a single value within a data object or by the combination of fields.

Another major functionality of the DCPS along with the topic's definition, the creation of publishers and subscribers is attaching QoS policies to all of the objects it creates. In many cases, for communication to occur properly QoS policies on the publisher and subscriber sides must be compatible. For instance, if a Publisher can do only best-effort data transfer, and a Subscriber wants its data reliably, the communication cannot occur. To address this issue while maintaining the desirable decoupling of subscribers and publishers as much as possible, the specification for QoS policies follows the subscriber-requested and publisher-offered pattern. In this pattern, the Subscriber side can specify a "requested" value for a particular QoS policy. The Publisher side specifies an "offered" value for this policy. The DDS then determines if the requested and offered values are compatible. If so, the communication will be established. For instance, the policy that is responsible for periodic updates is the DEADLINEQoS policy. The deadline on the publishing site is the contract the application must meet, the deadline on the subscriber side is a minimum requirement for the remote publisher supplying the data. To match a DataWriter and a

DataReader, the DDS checks the compatibility of settings (offered deadline \leq requested deadline). If they do not match, communication will not occur and both sides are informed of incompatibilities via the listeners or condition mechanisms. If matching occurs, the DDS monitors the fulfillment of the service agreement and informs the application of any violations by means of the proper listener or condition.

The DDS can be used for both static and dynamic types of systems of various sizes. The DDS can address both soft and hard real-time systems. However, it does not enforce any constraints. For this, an underlying real-time scheduling mechanism must be used. Both periodic and aperiodic requests can be specified. For specifying periodic requests, DEADLINEQoS and TIME_BASED_FILTERQoS policies are used. The filter allows a DataReader to specify that it wants to have no more than one value each minimum_separation regardless of how often changes occur. It is inconsistent for a DataReader to have minimum_separation longer than its DEADLINE period.

The data model assumed by the DDS is homogeneous. However, implementations of DLRL can provide transition among application data formats to the DDS data model, making the service suitable for heterogeneous applications. Since there is a decoupling between publishers writing to the data and subscribers accessing data, the data usage can be defined as read-only.

Both precise data and imprecise data (by the mean of TIME_BASED_FILTERQoS and HISTORY policies) can be used by DDS. Various levels of granularity can also be supported. By using MultiTopic Class, a subscriber can select and combine data from multiple topics into a single resulting type.

The OWNERSHIPQoS policy allows multiple DataWriters to update the same instance of data object. There are two settings for this policy: SHARED indicates that the service does not enforce unique ownership so multiple writers can update the same data instance simultaneously, and subscribers can access modifications from all DataWriters. EXCLUSIVE indicates that each data instance can be updated by one DataWriter that “owns” this instance, though the owner of data can be changed. Thus, the service provides both multiple and single data source solutions.

There are several implementations of DDS in both commercial and open source. RTI DDS (former NDDS) from Real-Time Innovations [20] and (until recently) SPLICE from THALES Naval Nederland [21,22] are commercially available products that fully implement DCPS and partially implement DLRL. OpenSPLICE, the product of a strategic alliance of THALES and PrizmTech [23], can be regarded as the most feature-full implementation of DDS with DLRL really integrated into DCPS.

TAO DDS is an open-source CORBA-based implementation of OMG DDS by Object Computing Inc. (OCI) [24]. Currently, it implements a subset of the DCPS layer and none of the DLRL functionality. There is also a Java-based implementation of DDS [25], and an open source DDS project called OpenDDS [26], both are in early stages of development.

25.3.5 Current Research in Real-Time Data Distribution

Along with the standard efforts toward RTDD described above, there has recently been active research into the development of systems that attempt to solve specific parts of the RTDD problem space. This section presents a brief discussion on some of the current research in this area.

OMG DDS-based implementations cannot guarantee that all the data will be delivered in time. In the case of a missed deadline, the service will notify the subscriber so that application can take some actions to preserve its determinism. However, the work in Ref. 27 describes a data distribution mechanism for static systems that can guarantee in-time delivery of all data and its temporal consistency. Since in a static system all system characteristics are known *a priori* and system analysis can be done ahead of time, the implementation of data distribution is divided into two parts: an offline analysis and online event-based delivery of data (see Figure 25.6).

The implementation begins with specification of the system containing description of all data sources, data sinks, nodes they are on, and the data itself. Using this information, the Just-In-Time Data Distribution (JITDD) [28] algorithm computes the scheduling parameters for each data distribution, then the complete system is analyzed using a schedulability tool such as RapidRMA [29] or OpenSTARS [30]. If the system is found to be schedulable, then a configuration file is created and used in the runtime implementation.

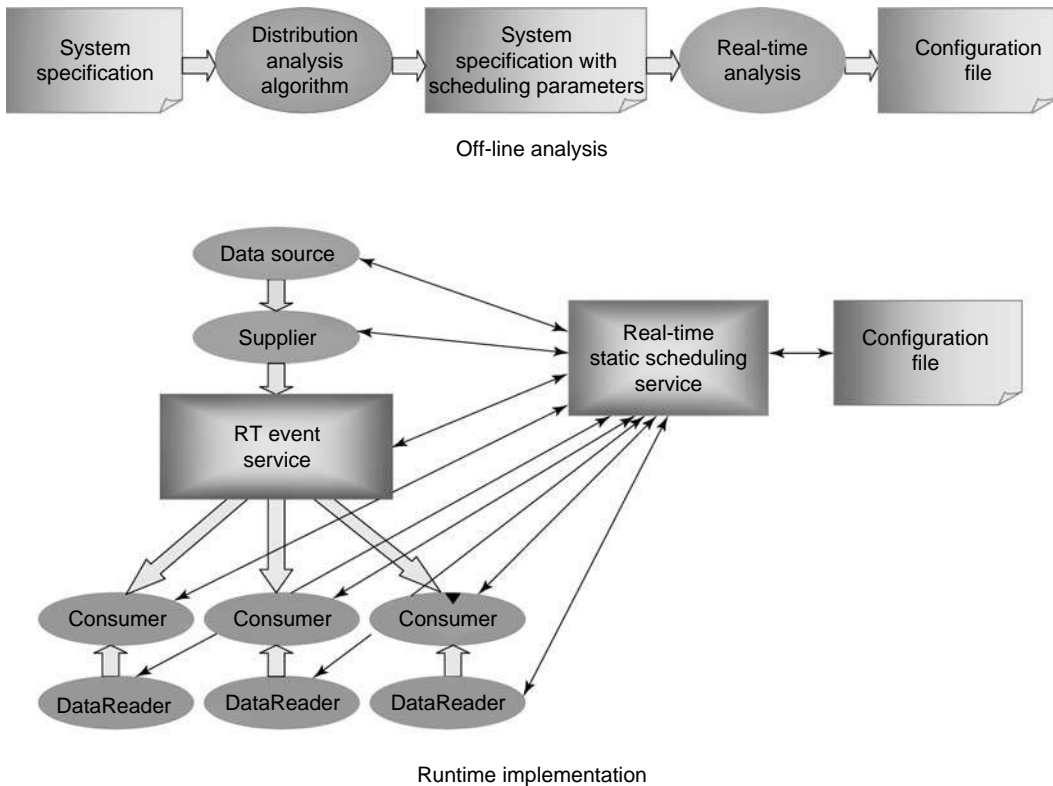


FIGURE 25.6 Static RTDD system design.

If the system is not schedulable, then its specification must be reworked by adding, for instance, more nodes or use more powerful nodes.

The runtime implementation runs on Linux Kernel 2.4.21 with TAO's v1.3.5 CORBA software and uses two of TAO's common services: the RT ES (described earlier) and the Real-Time Static Scheduling Service (RTSSS) [31]. The RT ES is used as a data distribution mechanism, and the RTSSS provides priority-based scheduling to ensure that all deadlines are met. The RTSSS is implemented as a set of library codes that creates a mapping of task to priority using the information from the configuration file and is compiled into programs that use it. When the system starts up, each of the executing entities (sources, suppliers, consumers, targets, RTES) requests its priority from the RTSSS, which looks them up in the task/priority mapping table, and sets the priority accordingly. Then each of these tasks executes at its specified priority, ensuring that all deadlines are preserved.

This solution assumes static applications and static infrastructure and it works well with small- and medium-scale applications. Computational resources are constrained by the hard deadlines that are specified by the application, but other resources are not necessarily constrained. The static nature of the applications allows this solution to support hard real-time constraints on periodic requests.

Follow-on work to the static RTDD research described above involves addressing dynamic, soft real-time systems. This work is still in development, however, the infrastructure for it is in place [32]. Figure 25.7 displays the overall structure of the system.

In this system, there is a Global Scheduling Service that is responsible for scheduling all data distribution to meet specified timing constraints. There is also a Global Data Distribution Service (Global DDS) that serves as a mechanism for sources and readers to find each other in the system. On each node, there is a Local DDS for data sources on the node, and a Local DDS for data readers on the node. Each data source

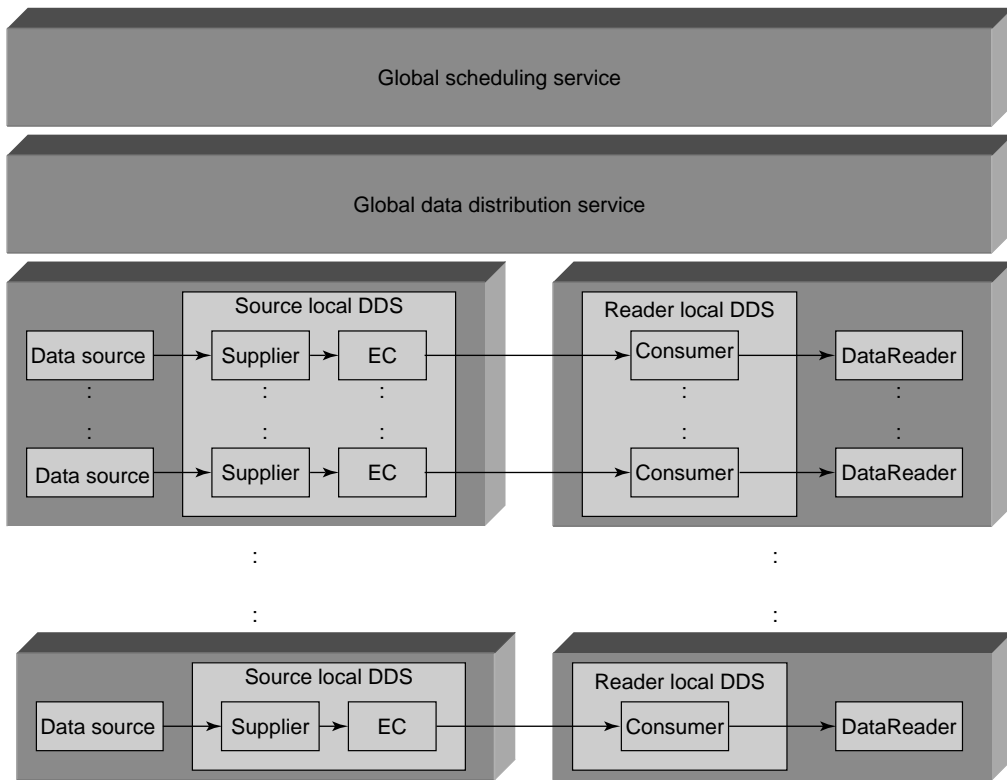


FIGURE 25.7 Dynamic RTDD system design.

and data reader must register its Local DDS to indicate the data and computational requirements. The Local DDS then contacts the Global DDS to indicate these services and requirements. The Global DDS then communicates with the Global Scheduling Service to ensure that the specified deadlines can be met.

When an appropriate data source is found for a requesting data reader, the Local DDS of the data source communicates directly with the Local DDS of the reader to pass along any data that is required by readers on that node. The JITDD algorithm [28] is implemented to ensure that data that is delivered is temporally consistent.

This solution extends the previous static solution to allow for support of dynamic applications. It works well with small- and medium-scale applications. In larger implementations, the use of a Global DDS and Global Scheduling Service can become bottlenecks. However, a more federated model of these services can be used. This solution is designed for applications with periodic requests that have soft real-time constraints.

Multiprocessor systems with shared data objects that have the read–write data model bring their own challenges. When data accessed by readers are maintained by more than one writer, both the integrity and coherency of data must be guaranteed. That is, the data must not only be transferred as a whole but also must be no older than any previously received version. It also should be free of unbounded blocking and should address the priority inversion problem. The work in Ref. 33 describes a loop-free asynchronous multiple writers/readers data sharing mechanism built on timing-based approach. This approach is based on circular-buffer sharing mechanism for one writer and multiple readers, where feasibility conditions and buffer configurations (minimum number of buffer slots) are derived from time properties of relevant reader/writer tasks. The multiple writers/readers system is built by replicating the above writer/multiple readers approach. To overcome violation of data coherency brought by the above data sharing mechanism a timestamp-based protocol is used.

Since all reader/writer task requirements must be known *a priori*, the target applications are static. This approach works in small- and medium-scale systems. The targeted applications of this approach allow for multiple writers and multiple readers of real-time data. Each data item in the applications targeted by this approach can have one or more original sources. This is where the requirement for multiple writers comes into play.

25.4 Conclusion

Many real-time systems have requirements to have timely data delivered within tight timing constraints. As this chapter has illustrated, the problem space involved with these types of applications is quite large. There are many approaches to solving the various problems in the space, and not all of the problems in the space have complete solutions.

There has been a strong movement to standardize solutions to various RTDD problems through the OMG. The standard specifications are still evolving into final requirements, and commercial implementations of these standards are beginning to become available. Research in the field, beyond the standards work, has also been active recently.

There are other emerging solutions to more areas of the RTDD problem space that are not described in this chapter but may be discussed in other chapters in this handbook. Real-time wireless sensor networks can provide a solution for RTDD, where data are collected from the environment by small, wireless devices that have tight memory, computational and power constraints. Given these constraints, the solutions provided by such systems must consider mechanisms for delivering data that use multiple hops from data source to data destination. These solutions also need to consider the unreliable communications that exist in wireless devices.

Other wireless systems can also be involved in RTDD solutions. For example, in a disaster recovery scenario, rescue workers can carry wireless PDAs to receive time-critical data that will help them to find survivors and to be updated on other environmental conditions, like weather. Solutions for systems like these can build upon the other RTDD approaches described previously in this chapter. The additional resource constraints along with the unreliability of the communications increases the complexity of the problem and thus requires more complex solutions.

References

1. A. Uvarova and V. Fay Wolfe, Towards a Definition of the Real-Time Data Distribution Problem Space, In *Proceedings of the 1st International Workshop on Data Distribution for Real-Time Systems*, Providence, RI, May 2003.
2. D. Aksoy, M. Altinel, R. Bose, U. Cetintemel, M. Franklin, J. Wang, and S. Zdonik, Research in Data Broadcast and Dissemination, *International Conference on Advanced Multimedia Content Processing (AMCP)*, Osaka, Japan, November 1998, pp. 196–211.
3. A. Bestavros, AIDA-Based Real-Time Fault-Tolerant Broadcast Disks, In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium. (RTAS'96)*, Boston, MA, June 1996, pp. 49–58.
4. W. Lam and H. Garcia-Molina, *Slicing Broadcast Disks*, Stanford University Technical Report, 2003.
5. OMG, Common Object Request Broker Architecture—Version 2.2, OMG Inc., 1998 (formal/98-12-01).
6. OMG RT CORBA (Static Scheduling)—Specification—Version 1.2, OMG Inc., 2005 (formal/2005-01-04).
7. OMG RT CORBA (Dynamic Scheduling)—Specification—Version 2.0, OMG Inc., 2005 (formal/2003-11-01).
8. OMG CORBA Messaging Specification, OMG Inc., 1998 (orbos/98-05-05).
9. A.B. Arulanthu, C. O’Ryan, D.C. Schmidt, M. Kircher, and J. Parsons, The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging, In *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, April 2000.

10. OMG Event Service—Specification—Version 1.2, OMG Inc., 2004 (formal/04-10-02).
11. T.H. Harrison, D.L. Levine, and D.C. Schmidt, The Design and Performance of a Real-Time CORBA Event Service, In *Proceedings of the OOPSLA'97 Conference*, Atlanta, GA, October 1997.
12. TAO, <http://www.cs.wustl.edu/~schmidt/TAO.html>.
13. C. Ma and J. Bacon, COBEA: A CORBA Based Event Architecture, In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, NM, 1998.
14. OMG Notification Service—Specification—Version 1.1, OMG Inc., 2004 (formal/04-10-11).
15. Object Mgmt. Group, *Real-Time Notification: Request for Proposals*, OMG Doc. orbos/00-06-10, June 2000.
16. P. Gore, I. Pyarali, C.D. Gill, and D.C. Schmidt, The Design and Performance of a Real-Time Notification Service, In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS'2004)*, Toronto, Canada, May 2004.
17. OMG CORBA Audio/Visual (A/V) Streams Specification, OMG Inc., 2000 (formal/2000-01-03).
18. S. Mungee, M. Surendran, Y. Krishnamurthy, and D.C. Schmidt, The Design and Performance of CORBA Audio/Video Streamin Service, In *Design and Management of Multimedia Information Systems: Opportunities and Challenges*, edited by M. Syed, Idea Group Publishing, Hershey, USA, 2001.
19. OMG Data Distribution Service for Real-Time Systems Specification, OMG Inc., 2005 (formal/05-12-04).
20. RTI DDS, http://www.rti.com/products/data_distribution/index.html.
21. J.H. van't Hag, Data-Centric to the Max, The SPLICE Architecture Experience, In *Proceedings of the First International Workshop on Data Distribution for Real-Time Systems*, Providence, RI, May 2003.
22. THALES, <http://www.thales.com>.
23. OpenSPLICE, <http://www.prisms technologies.com>.
24. TAO DDS, <http://www.ociweb.com/products/dds>.
25. JavaDDS, <http://www-adele.imag.fr/users/Didier.Donsez/dev/dds/readme.html>.
26. OpenDDS, <http://opendds.sourceforge.net/>.
27. A. Uvarov, L. DiPippo, V. Fay-Wolfe, K. Bryan, P. Gadrow, T. Henry, M. Murphy, P.R. Work, and L.P. DiPalma, Static Real-Time Data Distribution, In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS'2004)*, Toronto, Canada, May 2004.
28. P. Peddi and L. DiPippo, A Replication Strategy for Distributed Real-Time Object-Oriented Databases, *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, Washington, DC, April 2002.
29. TriPacific Software, www.tripac.com.
30. K. Bryan, T. Ren, J. Zhang, L. DiPippo, and V. Fay-Wolfe, The Design of the OpenSTARS Adaptive Analyzer for Real-Time Distributed Systems, In *Proceedings of the 2005 Workshop on Parallel and Distributed Real-Time Systems*, Interlocken, CO, April 2005.
31. M. Murphy and K. Bryan, *CORBA 1.0 Compliant Static Scheduling Service for Periodic Tasks Technical Documentation*, URI Technical Report TR04-297, January 2004.
32. J. Mao, *Implementation of a Dynamic Real-Time Data Distribution Service for Middleware System*, University of Rhode Island, Computer Science Dept, MS Thesis, 2005.
33. J. Chen, A Loop-Free Asynchronous Data Sharing Mechanism in Multiprocessor Real-Time System based on Timing Properties, In *Proceedings of the 1st International Workshop on Data Distribution for Real-Time Systems*, Providence, RI, May 2003.

26

Temporal Consistency Maintenance for Real-Time Update Transactions

Ming Xiong

Bell Labs

Krithi Ramamritham

Indian Institute of Technology

26.1	Introduction	26-1
26.2	More-Less Using EDF	26-3
	Restricted Optimization Problem for ML Using EDF •	
	Designing \mathcal{ML}_{EDF} Using a Sufficient Feasibility Condition	
26.3	More-Less Using <i>Deadline Monotonic</i>	26-7
26.4	Deferrable Scheduling	26-8
	Intuition of DS-FP • <i>Deferrable Scheduling Algorithm</i> •	
	Comparison of DS-FP and \mathcal{ML}_{DM} • Schedulability Analysis	
	for DS-FP	
26.5	Conclusions	26-17

26.1 Introduction

A real-time data object, for example, the speed or position of a vehicle, or the temperature in an engine, is temporally consistent if its value reflects the current status of the corresponding entity in the environment. This is usually achieved by associating the value with a *temporal validity interval* [3,4,6–8,11–14,17–19]. For example, if position or velocity changes that can occur in 10 s do not affect any decisions made about the navigation of a vehicle, then the temporal consistency interval associated with these data elements is 10 s.

One important design goal of real-time and embedded database systems is to always keep the real-time data temporally consistent. Otherwise, the systems cannot detect and respond to environmental changes in a timely fashion. Thus, sensor transactions that sample the latest status of the entities need to periodically refresh the values of real-time data objects before their old values expire. Given *temporal consistency* requirements for a set of sensor transactions, the problem of designing such sensor transactions comprises two issues if transactions are scheduled periodically [18]: (1) the determination of the updating period and deadline for each transaction from their *temporal consistency* requirements; and (2) the scheduling of periodic sensor transactions. Minimizing the update workload for maintaining temporal consistency is an important design issue because (1) it allows a system to accommodate more sensor transactions, (2) it allows a real-time embedded system to consume less energy, and (3) it leaves more processor capacity to other workload (e.g., transactions triggered owing to detected environmental changes).

To monitor the states of objects faithfully, a *real-time object must be refreshed by a sensor transaction before it becomes invalid, that is, before its temporal validity interval expires*. The actual length of the temporal validity interval of a real-time object is application dependent. Sensor transactions have periodic jobs, which are generated by intelligent sensors that sample the value of real-time objects. One important

design goal of real-time data bases (RTDBs) is to guarantee that real-time data remain fresh, that is, they are always valid. Temporal consistency (a.k.a. absolute consistency [12]) is proposed to determine the temporal validity of a real-time data object based on its update time by assuming that the highest rate of change in a real-time data object is known.

Definition 26.1

A real-time data object (X_i) at time t is temporally consistent (or temporally valid) if the sampling time (r_{ij}) of its most recent version plus the validity interval (\mathcal{V}_i) of the data object is not less than t , that is, $r_{ij} + \mathcal{V}_i \geq t$.

From here on, $\mathcal{T} = \{\tau_i\}_{i=1}^m$ refers to a set of periodic sensor transactions and $\mathcal{X} = \{X_i\}_{i=1}^m$ to a set of real-time data. All real-time data are kept in main memory. Associated with X_i ($1 \leq i \leq m$) is a validity interval of length \mathcal{V}_i ; transaction τ_i ($1 \leq i \leq m$) updates the corresponding data X_i with validity length \mathcal{V}_i . Because each sensor transaction updates different data, no concurrency control is considered for sensor transactions. We assume that a sensor always samples the value of a real-time data at the beginning of its period, and the system is *synchronous*, that is, all the first jobs of sensor transactions are initiated at the same time. C_i , D_i , and P_i ($1 \leq i \leq m$) denote the execution time, relative deadline, and period of transaction τ_i , respectively. The relative deadline D_i of the j th job J_{ij} of sensor transaction τ_i is defined as $D_i = d_{ij} - r_{ij}$, where d_{ij} is the absolute deadline of J_{ij} , and r_{ij} is the sampling (or release) time of J_{ij} . Formal definitions of the frequently used symbols are given in Table 26.1. Deadlines of sensor transactions are firm deadlines. The design goal for temporal consistency maintenance is to determine transaction schedules such that all the sensor transactions are schedulable and processor workload resulting from sensor transactions is minimized. For example, transaction schedules can be determined once P_i and D_i are fixed for periodic transactions. For convenience, we use the terms transaction and task interchangeably in this chapter. Throughout this chapter, we assume that scheduling algorithms are preemptive and ignore all preemption overhead.

To guarantee the validity of real-time data in RTDBs, the period and relative deadline of a sensor transaction are each typically set to be one-half of the data validity interval in *Half-Half* (HH) approach [6,12]. In Figure 26.1, the farthest distance (based on the sampling time of a periodic transaction job and the finishing time of its next job) of two consecutive jobs of transaction τ_i is $2P_i$. If $2P_i = \mathcal{V}_i$, then the validity of X_i is guaranteed as long as jobs of τ_i meet their deadlines. HH assumes a simple execution semantics for periodic transactions: a transaction must be executed once every period. However, there is no guarantee on when a job of a periodic transaction is actually executed within a period. Unfortunately, even though data validity is guaranteed, this design approach incurs unnecessarily high processor workload for the sensor update transactions. The design goal is to minimize the CPU workload of sensor transactions while guaranteeing the freshness of temporal data in RTDBs. For simplicity of discussion, we have assumed

TABLE 26.1 Symbols and Definitions

Symbol	Definition
X_i	Real-time data object i
τ_i	Update transaction updating X_i ($i = 1, \dots, m$)
J_{ij}	The j th job of τ_i ($j = 0, 1, 2, \dots$)
R_{ij}	Response time of J_{ij}
C_i	Computation time of transaction τ_i
\mathcal{V}_i	Validity (interval) length of X_i
f_{ij}	Finishing time of J_{ij}
r_{ij}	Release (sampling) time of J_{ij}
d_{ij}	Absolute deadline of J_{ij}
P_i	Period of transaction τ_i in <i>ML</i>
D_i	Relative deadline of transaction τ_i in <i>ML</i>
$\Theta_i(a, b)$	Total cumulative processor demands from higher-priority transactions received by τ_i in interval $[a, b]$

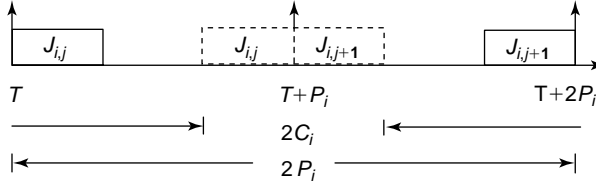
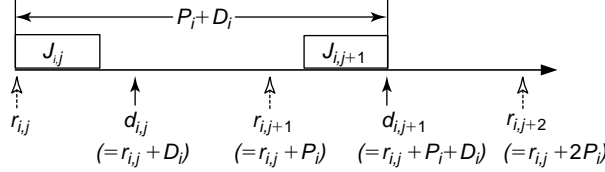


FIGURE 26.1 Extreme execution cases of periodic sensor transactions.

FIGURE 26.2 Illustration of the *More-Less* scheme.

that a sensor transaction is responsible for updating a single temporal data item in the system. In periodic scheduling based approaches discussed in Sections 26.2 and 26.3, namely *More-Less* (ML) approaches, the period of a sensor transaction is assigned to be *more* than *half* of the validity interval of the temporal data updated by the transaction, while its corresponding relative deadline is assigned to be *less* than *half* of the validity interval of the same data. However, the sum of the period and relative deadline always equals the length of the validity interval of the data updated. Consider Figure 26.2. Let $P_i > \frac{\mathcal{V}_i}{2}$, $C_i \leq D_i < P_i$, where $P_i + D_i = \mathcal{V}_i$. The farthest distance (based on the arrival time of a periodic transaction instance and the finishing time of its next instance) of two consecutive sensor transactions J_{ij} and J_{ij+1} is $P_i + D_i$. In this case, the freshness of X_i can always be maintained if sensor transactions make their deadlines. Obviously, the load incurred by sensor transaction τ_i can be reduced if P_i is enlarged (which implies that D_i is shrunk). Therefore, we have the constraints $C_i \leq D_i < P_i$ and $P_i + D_i = \mathcal{V}_i$ which aim at minimizing the CPU workload of periodic transaction τ_i .

Finally, this chapter presents *deferrable scheduling for fixed priority* transactions (DS-FP) with the objective to minimize the update workload even further. Distinct from the ML approaches, DS-FP adopts a *sporadic* task model, which dynamically assigns relative deadlines to transaction jobs by deferring the sampling time of a transaction job as much as possible. The deferral of a job's sampling time results in a relative deadline that is less than its job's worst-case response time, thus increases the separation of two consecutive jobs.

This chapter is organized as follows. Section 26.2 presents the ML algorithm using *earliest deadline first* (EDF) [10]. Section 26.3 presents the ML algorithm using *deadline monotonic* (DM) [9]. Section 26.4 presents the DS-FP algorithm and its schedulability analysis. Section 26.5 concludes the chapter and presents some open issues.

26.2 More-Less Using EDF

In this section, we present design of ML using EDF, while $D_i \leq P_i$ holds [20]. Section 26.2.1 formulates an EDF optimization problem for $D_i \leq P_i$. Section 26.2.2 gives the design detail of ML using EDF, namely $\mathcal{ML}_{\text{EDF}}$, by solving the optimization problem under a sufficient (but not necessary) feasibility condition.

26.2.1 Restricted Optimization Problem for ML Using EDF

The optimized solution has to minimize processor workload \mathcal{U} while maintaining the temporal validity of real-time data in RTDBs. This essentially formalizes the following optimization problem that minimizes \mathcal{U} with variables \vec{P} and \vec{D} . Note that \vec{P} and \vec{D} are vectors.

Problem 26.1

Restricted EDF optimization problem:

$$\min_{\vec{P}, \vec{D}} \mathcal{U}, \text{ where } \mathcal{U} = \sum_{i=1}^m \frac{C_i}{P_i} \quad (1 \leq i \leq m)$$

subject to

- *Validity constraint:* the sum of the period and relative deadline of transaction τ_i is always less than or equal to \mathcal{V}_i , the validity length of the object is updated, that is, $P_i + D_i \leq \mathcal{V}_i$, as shown in Figure 26.2.
- *Deadline constraint:* the period of a sensor transaction is assigned to be more than half of the validity length of the object to be updated by the transaction, while its corresponding relative deadline is assigned to be less than half of the validity length. For τ_i to be schedulable, D_i must be greater than or equal to C_i , the worst-case execution time of τ_i , that is, $C_i \leq D_i \leq P_i$.
- *Feasibility constraint:* \mathcal{T} with derived deadlines and periods is feasible by using EDF scheduling.

We next consider a sufficient feasibility condition for designing ML using EDF, namely $\mathcal{ML}_{\text{EDF}}$. We first present a technical result that is a sufficient condition for a set of periodic transactions \mathcal{T} to be feasible using EDF scheduler (see Ref. 15 for proof).

Lemma 26.1

Given $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$, suppose for all $\tau_i \in \mathcal{T}_1$, $D_i > P_i$, and for all $\tau_i \in \mathcal{T}_2$, $D_i \leq P_i$. If $\sum_{\tau_i \in \mathcal{T}_1} \frac{C_i}{P_i} + \sum_{\tau_i \in \mathcal{T}_2} \frac{C_i}{D_i} \leq 1$, then \mathcal{T} is feasible.

26.2.2 Designing $\mathcal{ML}_{\text{EDF}}$ Using a Sufficient Feasibility Condition

In this subsection, we study designing $\mathcal{ML}_{\text{EDF}}$ using Lemma 26.1 for all τ_i with $D_i \leq P_i$. If Lemma 26.1 is used to derive deadlines and periods for \mathcal{T} , the optimization problem for EDF scheduling is essentially transformed to the following problem.

Problem 26.2

EDF optimization problem with sufficient feasibility condition:

$$\min_{\vec{P}} \mathcal{U}, \text{ where } \mathcal{U} = \sum_{i=1}^m \frac{C_i}{P_i} \quad (1 \leq i \leq m)$$

subject to

- *Validity and deadline constraints* in Problem 26.1 and
- *Feasibility constraint:* $\sum_{i=1}^m \frac{C_i}{D_i} \leq 1$.

Without loss of generality, we assume that

$$D_i = \frac{1}{N_i} \mathcal{V}_i \quad (26.1)$$

$$P_i = \frac{N_i - 1}{N_i} \mathcal{V}_i \quad (26.2)$$

where N_i is a variable, $N_i > 0$ and $P_i + D_i = \mathcal{V}_i$. It should be obvious that \mathcal{U} is minimized only if $P_i + D_i = \mathcal{V}_i$. Otherwise, P_i can always be increased and processor utilization can be decreased.

Definition 26.2

Given a set of transactions \mathcal{T} , the density factor of \mathcal{T} , denoted as γ , is $\sum_{i=1}^m \frac{C_i}{V_i}$.

The following theorem provides a minimal solution for Problem 26.2.

Theorem 26.1

Given the minimization problem in Problem 26.2, there exists a unique minimal solution given by $N_k = \mathcal{N}_{ml}^{opt} = \frac{1}{\gamma} (1 \leq k \leq m \text{ and } 0 < \gamma \leq 0.5)$, which has minimum utilization $\mathcal{U}_{ml}^{opt}(\gamma) = \frac{\gamma}{1-\gamma}$.

Proof

From the deadline constraint in Problem 26.2, we have

$$\begin{aligned} P_i \geq D_i &\implies \frac{N_i - 1}{N_i} V_i \geq \frac{1}{N_i} V_i \implies N_i \geq 2 \\ D_i \geq C_i &\implies \frac{V_i}{C_i} \geq N_i \end{aligned}$$

Following Equations 26.1 and 26.2, Problem 26.2 is reduced to the following nonlinear programming problem with variable \vec{N} :

$$\min_{\vec{N}} \mathcal{U}, \text{ where } \mathcal{U} = \sum_{i=1}^m \frac{N_i C_i}{(N_i - 1) V_i} \quad (1 \leq i \leq m)$$

subject to

$$N_i \geq 2 \tag{26.3}$$

$$\frac{V_i}{C_i} \geq N_i \tag{26.4}$$

$$\sum_{i=1}^m N_i \cdot \frac{C_i}{V_i} \leq 1 \tag{26.5}$$

It can be proved that the objective function and all three constraints are *convex* functions. Thus, this is a convex programming problem, and a local minimum is a global minimum for this problem [5]. Considering Equations 26.3 and 26.5 together, we have $2 \cdot \sum_{i=1}^m \frac{C_i}{V_i} \leq \sum_{i=1}^m N_i \cdot \frac{C_i}{V_i} \leq 1$. That is

$$\sum_{i=1}^m \frac{C_i}{V_i} \leq \frac{1}{2} \tag{26.6}$$

Equation 26.6 implies that $\gamma \leq \frac{1}{2}$.

For convenience, let $w_i = \frac{C_i}{V_i}$, $\gamma = \sum_{i=1}^m \frac{C_i}{V_i} = \sum_{i=1}^m w_i$, and $x_i = N_i - 1$. By definition of \mathcal{U} and Equation 26.2, $\mathcal{U} = \sum_{i=1}^m \frac{N_i C_i}{(N_i - 1) V_i} = \sum_{i=1}^m \frac{x_i + 1}{x_i} w_i$, i.e., $\mathcal{U} = \sum_{i=1}^m w_i + \sum_{i=1}^m \frac{w_i}{x_i}$.

Following Equations 26.3 through 26.5, the problem is transformed to

$$\min_{\vec{x}} \sum_{i=1}^m \frac{w_i}{x_i}$$

subject to

$$x_i - 1 \geq 0 \quad (26.7)$$

$$\frac{1}{w_i} - x_i - 1 \geq 0 \quad (26.8)$$

$$1 - \gamma - \sum_{i=1}^m w_i \cdot x_i \geq 0 \quad (26.9)$$

We now solve the above nonlinear optimization problem by using Kuhn–Tucker condition [5]. Introducing Lagrangian multipliers $\lambda_{1,i}$, $\lambda_{2,i}$, and λ_3 , we write Kuhn–Tucker condition [5] as the following ($1 \leq i \leq m$):

$$-\frac{w_i}{x_i^2} + \lambda_3 w_i - \lambda_{2,i} + \lambda_{1,i} = 0 \quad (26.10)$$

$$\lambda_3 \left(1 - \gamma - \sum_{i=1}^m w_i x_i \right) = 0 \quad (26.11)$$

$$\lambda_3 \geq 0 \quad (26.12)$$

$$\lambda_{2,i}(x_i - 1) = 0 \quad (26.13)$$

$$\lambda_{2,i} \geq 0 \quad (26.14)$$

$$\lambda_{1,i} \left(\frac{1}{w_i} - x_i - 1 \right) = 0 \quad (26.15)$$

$$\lambda_{1,i} \geq 0 \quad (26.16)$$

We use the above conditions to construct an optimal solution. Suppose $\lambda_{1,i} = \lambda_{2,i} = 0$ and $\lambda_3 > 0$. Following Equation 26.10, $-\frac{w_i}{x_i^2} + \lambda_3 w_i = 0$. Therefore, $x_i = \frac{1}{\sqrt{\lambda_3}}$ ($\lambda_3 > 0$ and $1 \leq i \leq m$). Following $\lambda_3 > 0$ and Equation 26.11,

$$1 - \gamma - \sum_{i=1}^m w_i x_i = 0$$

Replacing x_i with $\frac{1}{\sqrt{\lambda_3}}$,

$$1 - \gamma - \frac{1}{\sqrt{\lambda_3}} \sum_{i=1}^m w_i = 0$$

Replacing $\sum_{i=1}^m w_i$ with γ ,

$$1 - \gamma - \frac{1}{\sqrt{\lambda_3}} \gamma = 0$$

Solving the above equation, we have $\lambda_3 = \left(\frac{\gamma}{1-\gamma} \right)^2$. It is easy to check that

$$\lambda_{1,i} = \lambda_{2,i} = 0, \quad \lambda_3 = \left(\frac{\gamma}{1-\gamma} \right)^2, \quad \text{and } x_i = \frac{1}{\sqrt{\lambda_3}} = \frac{1-\gamma}{\gamma}$$

satisfy Equations 26.7 through 26.16, which means that \vec{x} reaches a local minimum. Because the objective function is convex and constraints are all linear, \vec{x} is also a global optimal solution. Since $N_i = x_i + 1$, \mathcal{U} is

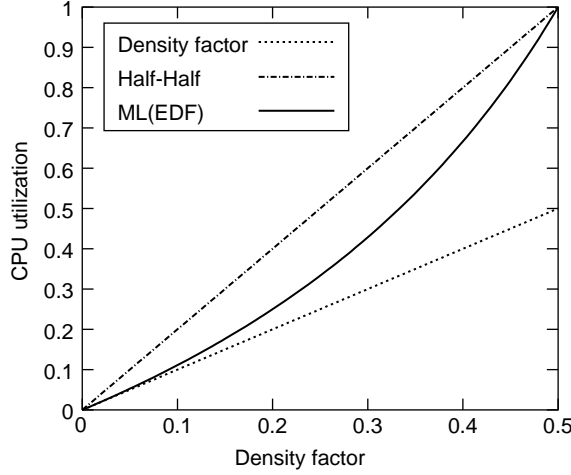


FIGURE 26.3 Utilization comparison: \mathcal{ML}_{EDF} versus \mathcal{HH} .

minimized when $N_i = \mathcal{N}_{ml}^{opt} = \frac{1}{\gamma}$, and the minimum utilization is $\mathcal{U}_{ml}^{opt}(\gamma) = \mathcal{N}_{ml}^{opt} / (\mathcal{N}_{ml}^{opt} - 1) \sum_{i=1}^m \frac{C_i}{V_i} = (\frac{1}{\gamma} / (\frac{1}{\gamma} - 1))\gamma = \gamma / (1 - \gamma)$.

Given a set \mathcal{T} of N sensor transactions, the optimization problem defined in Problem 26.2 can be solved in $O(N)$. Given the discussion on HH, the utilization of HH is $\mathcal{U}_{hh}(\gamma) = 2\gamma$.

Theorem 26.2

Given a set \mathcal{T} of sensor transactions ($0 < \gamma \leq 0.5$), $\mathcal{U}_{hh}(\gamma) - \mathcal{U}_{ml}^{opt}(\gamma) \leq 3 - 2\sqrt{2} \approx 0.172$.

Proof

Let $\mathcal{D}(\gamma) = \mathcal{U}_{hh}(\gamma) - \mathcal{U}_{ml}^{opt}(\gamma)$. From definitions of $\mathcal{U}_{ml}^{opt}(\gamma)$ and $\mathcal{U}_{hh}(\gamma)$, it follows that $\mathcal{D}(\gamma) = 2\gamma - \frac{\gamma}{1-\gamma}$. To obtain the maximum of $\mathcal{D}(\gamma)$, we differentiate $\mathcal{D}(\gamma)$ with respect to γ , and set the result to 0:

$$\frac{d\mathcal{D}(\gamma)}{d\gamma} = \frac{2\gamma^2 - 4\gamma + 1}{(1 - \gamma)^2} = 0$$

Thus the maximum of $\mathcal{D}(\gamma)$ is $3 - 2\sqrt{2}$, when $\gamma = 1 - \sqrt{2}/2$.

The function curves of γ , $\mathcal{U}_{hh}(\gamma)$ and $\mathcal{U}_{ml}^{opt}(\gamma)$ ($0 < \gamma \leq 0.5$), are depicted in Figure 26.3. \mathcal{ML}_{EDF} improves utilization by solving Problem 26.2 in linear time. However, it does not necessarily produce an optimal solution for Problem 26.1 in that the feasibility condition in \mathcal{ML}_{EDF} (i.e., Lemma 26.1) is sufficient but not necessary. We next consider a more accurate schedulability condition for ML under DM scheduling.

26.3 More–Less Using *Deadline Monotonic*

Suppose that a set of update transactions is scheduled by the DM scheduling algorithm [9]. Note that DM is a fixed priority scheduling algorithm, whereas EDF is a dynamic priority scheduling algorithm. Consider the longest response time for any job of a periodic transaction τ_i in DM scheduling. The response time is the difference between the transaction initiation time ($I_i + KP_i$) and the transaction completion time, where I_i is the offset within the period.

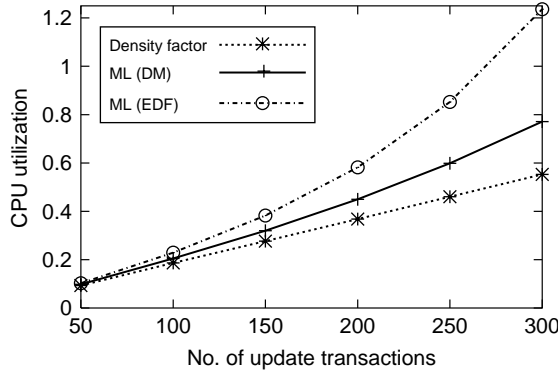


FIGURE 26.4 CPU Utilization comparison: \mathcal{ML}_{DM} versus \mathcal{ML}_{EDF} .

Lemma 26.2

For a set of periodic transactions, $\mathcal{T} = \{\tau_i\}_{i=1}^m$ ($D_i \leq P_i$) with transaction initiation time $(I_i + KP_i)$ ($K = 0, 1, 2, \dots$), the longest response time for any job of τ_i occurs for the first job of τ_i when $I_1 = I_2 = \dots = I_m = 0$ [10].

A time instant after which a transaction has the longest response time is called a *critical instant*, for example, time 0 is a critical instant for all transactions if those transactions are *synchronous* (i.e., starting at the same time together) and $(D_i \leq P_i)$ ($1 \leq i \leq m$) [9,10]. To minimize the update workload, ML using DM, namely \mathcal{ML}_{DM} , is used to guarantee temporal consistency with much less processor workload than HH [2,18]. In \mathcal{ML}_{DM} , DM is used to schedule periodic sensor transactions. There are three constraints to follow:

- *Validity constraint:* $P_i + D_i \leq \mathcal{V}_i$.
- *Deadline constraint:* $C_i \leq D_i \leq P_i$.
- *Feasibility constraint:* for a given set of sensor transactions, DM scheduling algorithm [9] is used to schedule the transactions. Consequently, $\sum_{j=1}^i (\lceil \frac{D_i}{P_j} \rceil \cdot C_j) \leq D_i$ ($1 \leq i \leq m$).

\mathcal{ML}_{DM} assigns priorities to transactions according to *shortest validity first*, that is, in the *inverse* order of validity length and resolves ties in favor of transactions with larger computation times. It assigns deadlines and periods to τ_i as follows:

$$D_i = f_{i,0} - r_{i,0}, \quad P_i = \mathcal{V}_i - D_i$$

where $f_{i,0}$ and $r_{i,0}$ are finishing and sampling times of $J_{i,0}$, respectively.

In \mathcal{ML}_{DM} , the first job's response time is the longest response time. This assumption holds for DM algorithm when $D_i \leq P_i$. A transaction set is not schedulable if there exists $\tau_i \in \mathcal{T}$, $f_{i,0} - r_{i,0} > \frac{\mathcal{V}_i}{2}$.

An example of the utilization comparison of \mathcal{ML}_{DM} and \mathcal{ML}_{EDF} for sensor update transactions is depicted in Figure 26.4 [20]. Note that \mathcal{ML}_{EDF} becomes infeasible when CPU utilization is greater than 1. It clearly demonstrates that \mathcal{ML}_{DM} outperforms \mathcal{ML}_{EDF} in terms of minimizing CPU utilization for update transactions. This is because \mathcal{ML}_{DM} uses a sufficient and necessary condition, which is more accurate than the sufficient condition in \mathcal{ML}_{EDF} , for assigning deadlines and periods of update transactions.

26.4 Deferrable Scheduling

In this subsection, we present *DS-FP*, and compare it with \mathcal{ML}_{DM} [16,17].

26.4.1 Intuition of DS-FP

In \mathcal{ML}_{DM} , D_i is determined by the first job's response time, which is the *worst-case* response time of all jobs of τ_i . Thus, \mathcal{ML}_{DM} is pessimistic on the deadline and period assignment in the sense that it uses periodic task model that has a fixed period and deadline for each task, and the deadline is equivalent to the worst-case response time. It should be noted that the *validity constraint* can always be satisfied as long as $P_i + D_i \leq \mathcal{V}_i$. But processor workload is minimized only if $P_i + D_i = \mathcal{V}_i$. Otherwise, P_i can always be increased to reduce processor workload as long as $P_i + D_i < \mathcal{V}_i$. Given release time $r_{i,j}$ of job $J_{i,j}$ and deadline $d_{i,j+1}$ of job $J_{i,j+1}$ ($j \geq 0$),

$$d_{i,j+1} = r_{i,j} + \mathcal{V}_i \quad (26.17)$$

guarantees that the *validity constraint* can be satisfied, as depicted in Figure 26.5. Correspondingly, the following equation follows directly from Equation 26.17:

$$(r_{i,j+1} - r_{i,j}) + (d_{i,j+1} - r_{i,j+1}) = \mathcal{V}_i \quad (26.18)$$

If $r_{i,j+1}$ is shifted onward to $r'_{i,j+1}$ along the time line in Figure 26.5, it does not violate Equation 26.18. This shift can be achieved, for example, in the \mathcal{ML}_{DM} schedule, if preemption to $J_{i,j+1}$ from higher-priority transactions in $[r_{i,j+1}, d_{i,j+1}]$ is less than the worst-case preemption to the first job of τ_i . Thus, temporal validity can still be guaranteed as long as $J_{i,j+1}$ is completed by its deadline $d_{i,j+1}$.

The intuition of DS-FP is to defer the sampling time, $r_{i,j+1}$, of $J_{i,j}$'s subsequent job as late as possible while still guaranteeing the *validity constraint*. Note that the sampling time of a job is also its release time, that is, the time that the job is ready to execute as we assume zero cost for sampling and no arrival jitter for a job for convenience of presentation.

The deferral of job $J_{i,j+1}$'s release time reduces the relative deadline of the job if its absolute deadline is fixed as in Equation 26.17. For example, $r_{i,j+1}$ is deferred to $r'_{i,j+1}$ in Figure 26.5, but it still has to complete by its deadline $d_{i,j+1}$ to satisfy the *validity constraint* (Equation 26.17). Thus its relative deadline, $D_{i,j+1}$, becomes $d_{i,j+1} - r'_{i,j+1}$, which is less than $d_{i,j+1} - r_{i,j+1}$. The deadline of $J_{i,j+1}$'s subsequent job, $J_{i,j+2}$, can be further deferred to $(r'_{i,j+1} + \mathcal{V}_i)$ to satisfy the *validity constraint*. Consequently, the processor utilization for completion of three jobs, $J_{i,j}$, $J_{i,j+1}$, and $J_{i,j+2}$ then becomes $3C_i/(2\mathcal{V}_i - (d_{i,j+1} - r'_{i,j+1}))$. It is less than the utilization $3C_i/(2\mathcal{V}_i - (d_{i,j+1} - r_{i,j+1}))$ required for the completion of the same amount of work in \mathcal{ML}_{DM} .

Definition 26.2

Let $\Theta_i(a, b)$ denote τ_i 's total cumulative processor demands made by all jobs of higher-priority transaction τ_j for $\forall j (1 \leq j \leq i - 1)$ during time interval $[a, b]$ from a schedule S produced by a fixed priority scheduling

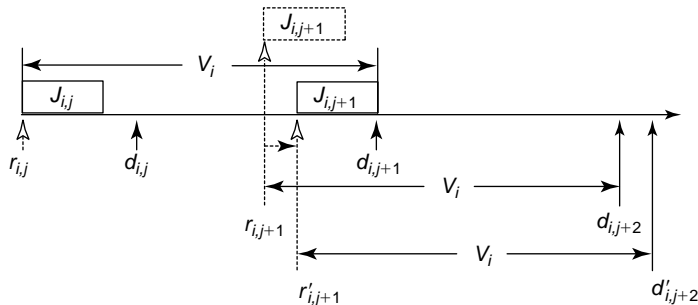


FIGURE 26.5 Illustration of DS-FP scheduling ($r_{i,j+1}$ is shifted to $r'_{i,j+1}$).

algorithm. Then,

$$\Theta_i(a, b) = \sum_{j=1}^{i-1} \theta_j(a, b)$$

where $\theta_j(a, b)$ is the total processor demands made by all jobs of single transaction τ_j during $[a, b)$.

Now we discuss how much a job's release time can be deferred. According to fixed priority scheduling theory, $r'_{i,j+1}$ can be derived backwards from its deadline $d_{i,j+1}$ as follows:

$$r'_{i,j+1} = d_{i,j+1} - \Theta_i(r'_{i,j+1}, d_{i,j+1}) - C_i \quad (26.19)$$

Note that the schedule of all higher-priority jobs that are released prior to $d_{i,j+1}$ needs to be computed before $\Theta_i(r'_{i,j+1}, d_{i,j+1})$ is computed. This computation can be invoked in a recursive process from jobs of lower-priority transactions to higher-priority transactions. Nevertheless, it does not require that schedule of all jobs should be constructed offline before the task set is executed. Indeed, the computation of job deadlines and their corresponding release times is performed online while the transactions are being scheduled. We only need to compute the first job's response times when system starts. Upon completion of job $J_{i,j}$, deadline of its next job, $d_{i,j+1}$, is first derived from Equation 26.17, then the corresponding release time $r'_{i,j+1}$ is derived from Equation 26.19. If $\Theta_i(r'_{i,j+1}, d_{i,j+1})$ cannot be computed due to incomplete schedule information of release times and absolute deadlines from higher-priority transactions, DS-FP computes their complete schedule information online until it can gather enough information to derive $r'_{i,j+1}$. Job $J_{i,j}$'s DS-FP scheduling information (e.g., release time, deadline, and bookkeeping information) can be discarded after it completes and it is not needed by jobs of lower-priority transactions. This process is called *garbage collection* in DS-FP.

Let $S_j(t)$ denote the set of jobs of all transactions whose deadlines have been computed by time t . Also let $LSD_i(t)$ denote the latest scheduled deadline of τ_i at t , that is, maximum of all $d_{i,j}$ for jobs $J_{i,j}$ of τ_i whose deadlines have been computed by t ,

$$LSD_i(t) = \max_{J_{i,j} \in S_j(t)} \{d_{i,j}\} \quad (j \geq 0) \quad (26.20)$$

Given a job $J_{k,j}$ whose scheduling information has been computed at time t , and $\forall i \ (i > k)$, if

$$LSD_i(t) \geq d_{k,j} \quad (26.21)$$

then the information of $J_{k,j}$ can be garbage collected.

Example 26.1

Suppose that there are three update transactions whose parameters are shown in Table 26.2. The resulting periods and deadlines in HH and \mathcal{ML}_{DM} are shown in the same table. Utilizations of HH and \mathcal{ML}_{DM} are $\mathcal{U}_{ml} \approx 0.68$ and $\mathcal{U}_{hh} = 1.00$, respectively.

TABLE 26.2 Parameters and Results for Example 26.1

i	C_i	\mathcal{V}_i	$\frac{C_i}{\mathcal{V}_i}$	\mathcal{ML}_{DM}		Half-Half
				P_i	D_i	$P_i(D_i)$
1	1	5	0.2	4	1	2.5
2	2	10	0.2	7	3	5
3	2	20	0.1	14	6	10

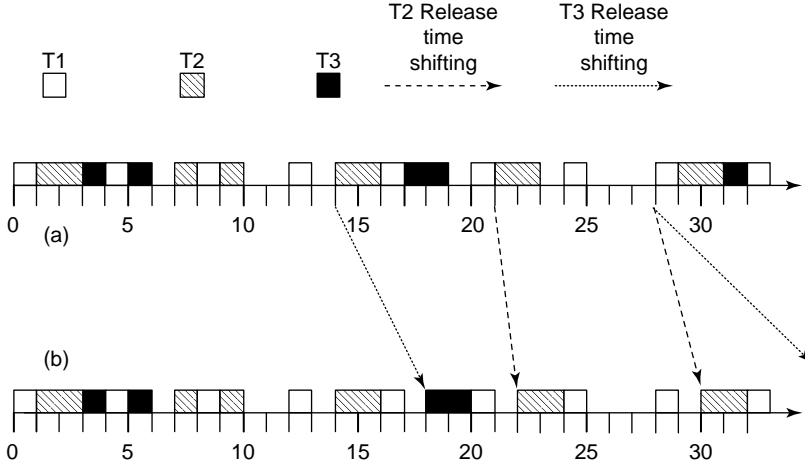


FIGURE 26.6 Comparing \mathcal{ML}_{DM} and DS-FP schedules. (a) Schedule with *More-Less*. (b) Schedule with DS-FP.

Figures 26.6a and 26.6b depict the schedules produced by \mathcal{ML}_{DM} and DS-FP, respectively. It can be observed from both schedules that the release times of transaction jobs $J_{3,1}$, $J_{2,3}$, $J_{2,4}$, and $J_{3,2}$ are shifted from times 14, 21, 28, and 28 in \mathcal{ML}_{DM} to 18, 22, 30, and 35 in DS-FP, respectively.

The DS-FP algorithm is described in Section 26.4.2.

26.4.2 Deferrable Scheduling Algorithm

This subsection presents the DS-FP algorithm. It is a *fixed priority* scheduling algorithm. Given an update transaction set \mathcal{T} , it is assumed that τ_i has higher priority than τ_j if $i < j$. Transaction priority assignment policy in DS-FP is the same as in \mathcal{ML}_{DM} , that is, *shortest validity first*. Algorithm 26.1 presents the DS-FP algorithm. For convenience of presentation, garbage collection is omitted in the algorithm. There are two cases for the DS-FP algorithm: (1) At system initialization time, Lines 13–20 iteratively calculate the first job's response time for τ_i . The first job's deadline is set as its response time (Line 21). (2) Upon completion of τ_i 's job $J_{i,k}$ ($1 \leq i \leq m, k \geq 0$), the deadline of its next job ($J_{i,k+1}$), $d_{i,k+1}$, is derived at Line 27 so that the farthest distance of $J_{i,k}$'s sampling time and $J_{i,k+1}$'s finishing time is bounded by the validity length \mathcal{V}_i (Equation 26.17). Then the sampling time of $J_{i,k+1}$, $r_{i,k+1}$, is derived backwards from its deadline by accounting for the interference from higher-priority transactions (Line 29).

Function $ScheduleRT(i, k, C_i, d_{i,k})$ (Algorithm 26.2) calculates the release time $r_{i,k}$ with known computation time C_i and deadline $d_{i,k}$. It starts with release time $r_{i,k} = d_{i,k} - C_i$, then iteratively calculates $\Theta_i(r_{i,k}, d_{i,k})$, the total cumulative processor demands made by all higher-priority jobs of $J_{i,k}$ during interval $[r_{i,k}, d_{i,k})$, and adjusts $r_{i,k}$ by accounting for interference from higher-priority transactions (Lines 5–12). The computation of $r_{i,k}$ continues until interference from higher-priority transactions does not change in an iteration. In particular, Line 9 detects an infeasible schedule. A schedule becomes infeasible under DS-FP if $r_{i,k} < d_{i,k-1}$ ($k > 0$), that is, release time of $J_{i,k}$ becomes earlier than the deadline of its preceding job $J_{i,k-1}$. Function $GetHPPreempt(i, k, t_1, t_2)$ scans interval $[t_1, t_2)$, adds up total preemptions from τ_j ($\forall j, 1 \leq j \leq i-1$), and returns $\Theta_i(t_1, t_2)$, the cumulative processor demands of τ_j during $[t_1, t_2)$ from schedule \mathcal{S} that has been built.

Function $CalcHPPreempt(i, k, t_1, t_2)$ (Algorithm 26.3) calculates $\Theta_i(t_1, t_2)$, the total cumulative processor demands made by all higher-priority jobs of $J_{i,k}$ during the interval $[t_1, t_2)$. Line 7 indicates that ($\forall j, 1 \leq j < i$), τ_j 's schedule is completely built before time t_2 . This is because τ_i 's schedule cannot be completely built before t_2 unless schedules of its higher-priority transactions are complete before t_2 . In this case, the function simply returns the amount of higher-priority preemptions for τ_i during $[t_1, t_2)$ by

Algorithm 26.1 DS-FP algorithm:

Input: A set of update transactions $\mathcal{T} = \{\tau_i\}_{i=1}^m$ ($m \geq 1$) with known $\{C_i\}_{i=1}^m$ and $\{\mathcal{V}_i\}_{i=1}^m$.
Output: Construct a partial schedule *Sif* \mathcal{T} is feasible; otherwise, reject.

```

1 case (system initialization time):
2    $t \leftarrow 0$ ; // Initialization
3   //  $LSD_i$  – Latest Scheduled Deadline of  $\tau_i$ 's jobs.
4    $LSD_i \leftarrow 0, \forall i (1 \leq i \leq m)$ ;
5    $\ell_i \leftarrow 0, \forall i (1 \leq i \leq m)$ ;
6   //  $\ell_i$  is the latest scheduled job of  $\tau_i$ 
7   for  $i = 1$  to  $m$  do
8     // Schedule finish time for  $\tau_{i,0}$ .
9      $r_{i,0} \leftarrow 0$ ;
10     $f_{i,0} \leftarrow C_i$ ;
11    // Calculate higher-priority (HP) preemptions.
12     $oldHPPreempt \leftarrow 0$ ; // initial HP preemptions
13     $hpPreempt \leftarrow CalcHPPreempt(i, 0, 0, f_{i,0})$ ;
14    while ( $hpPreempt > oldHPPreempt$ ) do
15      // Accounting for the interference of HP tasks
16       $f_{i,0} \leftarrow r_{i,0} + hpPreempt + C_i$ ;
17      if ( $f_{i,0} > \mathcal{V}_i - C_i$ ) then abort endif;
18       $oldHPPreempt \leftarrow hpPreempt$ ;
19       $hpPreempt \leftarrow CalcHPPreempt(i, 0, 0, f_{i,0})$ ;
20    end
21     $d_{i,0} \leftarrow f_{i,0}$ ;
22  end
23  return;
24
25 case (upon completion of  $J_{i,k}$ ):
26  // Schedule release time for  $J_{i,k+1}$ .
27   $d_{i,k+1} \leftarrow \tau_{i,k} + \mathcal{V}_i$ ; // get next deadline for  $J_{i,k+1}$ 
28  //  $r_{i,k+1}$  is also the sampling time for  $J_{i,k+1}$ 
29   $r_{i,k+1} \leftarrow ScheduleRT(i, k+1, C_i, d_{i,k+1})$ ;
30  return;

```

invoking $GetHPPreempt(i, k, t_1, t_2)$, which returns $\Theta_i(t_1, t_2)$. If any higher-priority transaction τ_j ($j < i$) does not have a complete schedule during $[t_1, t_2)$, its schedule \mathcal{S} up to or exceeding t_2 is built on the fly (Lines 13–18). This enables the computation of $\Theta_i(t_1, t_2)$. The latest scheduled deadline of τ_i 's job, LSD_i , indicates the latest deadline of τ_i 's jobs that have been computed.

The following lemma states an important property of *ScheduleRT* when it terminates.

Lemma 26.3

Given a synchronous update transaction set \mathcal{T} and *ScheduleRT*($i, k, C_i, d_{i,k}$) ($1 \leq i \leq m$ and $k \geq 0$), $LSD_l(t) \leq LSD_j(t)$ ($k \geq l \geq j$) holds when *ScheduleRT*($i, k, C_i, d_{i,k}$) terminates at time t .

Proof

This can be proved by contradiction. Suppose that $LSD_l(t) > LSD_j(t)$ ($k \geq l \geq j$) when *ScheduleRT*($i, k, C_i, d_{i,k}$) terminates at t . Let $t_2 = LSD_l(t)$ at Line 13 in *CalcHPPreempt*. As $LSD_j(t) < t_2$ at the same line, *ScheduleRT* has not reached the point to terminate. This contradicts the assumption.

Algorithm 26.2 $\text{ScheduleRT}(i, k, C_i, d_{i,k})$:**Input:** $J_{i,k}$ with C_i and $d_{i,k}$.**Output:** $r_{i,k}$.

```

1   $\text{oldHPPreempt} \leftarrow 0$ ; // initial HP preemptions
2   $\text{hpPreempt} \leftarrow 0$ ;
3   $r_{i,k} \leftarrow d_{i,k} - C_i$ ;
4  // Calculate HP preemptions backwards from  $d_{i,k}$ .
5   $\text{hpPreempt} \leftarrow \text{CalcHPPreempt}(i, k, r_{i,k}, d_{i,k})$ ;
6  while ( $\text{hpPreempt} > \text{oldHPPreempt}$ ) do
7    // Accounting for the interference of HP tasks
8     $r_{i,k} \leftarrow d_{i,k} - \text{hpPreempt} - C_i$ ;
9    if ( $r_{i,k} < d_{i,k-1}$ ) then abort endif;
10    $\text{oldHPPreempt} \leftarrow \text{hpPreempt}$ ;
11    $\text{hpPreempt} \leftarrow \text{GetHPPreempt}(i, k, r_{i,k}, d_{i,k})$ ;
12 end
13 return  $r_{i,k}$ ;

```

Algorithm 26.3 $\text{CalcHPPreempt}(i, k, t_1, t_2)$:**Input:** $J_{i,k}$, and a time interval $[t_1, t_2]$.**Output:** Total cumulative processor demands from higher-priority transactions $\tau_j (1 \leq j \leq i-1)$ during $[t_1, t_2]$.

```

1   $\ell_i \leftarrow k$ ; // Record latest scheduled job of  $\tau_i$ .
2   $d_{i,k} \leftarrow t_2$ ;
3   $\text{LSD}_i \leftarrow t_2$ ;
4  if ( $i = 1$ )
5    then // No preemptions from higher-priority tasks.
6      return 0;
7  elseif ( $\text{LSD}_{i-1} \geq t_2$ )
8    then // Get preemptions from  $\tau_j (\forall j, 1 \leq j < i)$  because  $\tau_j$ 's schedule is complete before  $t_2$ .
9      return  $\text{GetHPPreempt}(i, k, t_1, t_2)$ ;
10 endif
11 // build  $S$  up to or exceeding  $t_2$  for  $\tau_j (1 \leq j < i)$ .
12 for  $j = 1$  to  $i-1$  do
13   while ( $d_{j,\ell_j} < t_2$ ) do
14      $d_{j,\ell_j+1} \leftarrow r_{j,\ell_j} + \mathcal{V}_j$ ;
15      $r_{j,\ell_j+1} \leftarrow \text{ScheduleRT}(j, \ell_j + 1, C_j, d_{j,\ell_j+1})$ ;
16      $\ell_j \leftarrow \ell_j + 1$ ;
17      $\text{LSD}_j \leftarrow d_{j,\ell_j}$ ;
18   end
19 end
20 return  $\text{GetHPPreempt}(i, k, t_1, t_2)$ ;

```

The next example illustrates how DS-FP algorithm works with the transaction set in Example 26.1.

Example 26.2

Table 26.3 presents the comparison of (release time, deadline) pairs of τ_1 , τ_2 , and τ_3 jobs before time 40 in Example 26.1, which are assigned by \mathcal{ML}_{DM} and DS-FP (Algorithm 26.1). Note that τ_1 has same release

TABLE 26.3 Release Time and Deadline Comparison

Job	τ_1	τ_2		τ_3	
	$\mathcal{ML}_{DM}/DS-FP$	\mathcal{ML}_{DM}	DS-FP	\mathcal{ML}_{DM}	DS-FP
0	(0, 1)	(0, 3)	(0, 3)	(0, 6)	(0, 6)
1	(4, 5)	(7, 10)	(7, 10)	(14, 20)	(18, 20)
2	(8, 9)	(14, 17)	(14, 17)	(28, 34)	(35, 38)
3	(12, 13)	(21, 24)	(22, 24)
4	(16, 17)	(28, 31)	(30, 32)		
5	(20, 21)	(35, 38)	(38, 40)		
6	(24, 25)		
7	(28, 29)				
8	(32, 33)				
9	(36, 37)				

times and deadlines for all jobs under \mathcal{ML}_{DM} and DS-FP. However, $J_{2,3}$, $J_{2,4}$, $J_{2,5}$, $J_{3,1}$, and $J_{3,2}$ have different release times and deadlines under \mathcal{ML}_{DM} and DS-FP. Algorithm 26.1 starts at system initialization time. It calculates deadlines for $J_{1,0}$, $J_{2,0}$, and $J_{3,0}$. Upon completion of $J_{3,0}$ at time 6, $d_{3,1}$ is set to $r_{3,0} + \mathcal{V}_3 = 20$. Then Algorithm 26.1 invokes $\text{ScheduleRT}(3, 1, 2, 20)$ at Line 29, which will derive $r_{3,1}$. At this moment, Algorithm 26.1 has already calculated the complete schedule up to $d_{3,0}$ (time 6). But the schedule in the interval $(6, 20]$ has only been partially derived. Specifically, only schedule information of $J_{1,0}$, $J_{1,1}$, $J_{1,2}$, $J_{1,3}$, $J_{2,0}$, and $J_{2,1}$ has been derived for τ_1 and τ_2 . Algorithm 26.2 (ScheduleRT) obtains $r_{3,1} = 20 - 2 = 18$ at Line 3, then invokes $\text{CalcHPPreempt}(3, 1, 18, 20)$. Algorithm 26.3 (CalcHPPreempt) finds out that $\text{LSD}_2 = 10 < t_2 = 20$, then it jumps to the for loop starting at Line 12 to build the complete schedule of τ_1 and τ_2 in the interval $(6, 20]$, where the release times and deadlines for $J_{1,4}$, $J_{1,5}$, $J_{2,2}$, $J_{1,6}$, and $J_{2,3}$ are derived. Thus, higher-priority transactions τ_1 and τ_2 have a complete schedule before time 20. Note that $r_{1,6}$ and $d_{1,6}$ for $J_{1,6}$ are derived when we calculate $r_{2,3}$ and $d_{2,3}$, such that the complete schedule up to $d_{2,3}$ has been built for transactions with priority higher than τ_2 . As $r_{2,2}$ is set to 14 by earlier calculation, $d_{2,3}$ is set to 24. It derives $r_{2,3}$ backwards from $d_{2,3}$ and sets it to 22 because $\Theta_2(22, 24) = 0$. Similarly, $d_{3,1}$ and $r_{3,1}$ are set to 20 and 18, respectively.

26.4.3 Comparison of DS-FP and \mathcal{ML}_{DM}

Note that \mathcal{ML}_{DM} is based on the *periodic* task model, while DS-FP adopts one similar to the *sporadic* task model [1] except that the relative deadline of a transaction in DS-FP is not fixed. Theoretically, the separation of two consecutive jobs of τ_i in DS-FP satisfies the following condition:

$$\mathcal{V}_i - C_i \geq r_{i,j} - r_{i,j-1} \geq \mathcal{V}_i - \text{WCRT}_i \quad (j \geq 1) \quad (26.22)$$

where WCRT_i is the worst-case response time of jobs of τ_i in DS-FP. Note that the maximal separation of $J_{i,j}$ and $J_{i,j-1}$ ($j \geq 1$), $\max_j \{r_{i,j} - r_{i,j-1}\}$, cannot exceed $\mathcal{V}_i - C_i$, which can be obtained when there are no higher-priority preemptions for the execution of job $J_{i,j}$ (e.g., the highest-priority transaction τ_1 always has separation $\mathcal{V}_1 - C_1$ for $J_{1,j}$ and $J_{1,j-1}$). Thus, the processor utilization for DS-FP should be greater than $\sum_{i=1}^m \frac{C_i}{\mathcal{V}_i - C_i}$, which is the CPU workload resulting from the maximal separation $\mathcal{V}_i - C_i$ of each transaction.

\mathcal{ML}_{DM} can be regarded as a special case of DS-FP, in which sampling (or release) time $r_{i,j+1}^{ml}$ and deadline $d_{i,j+1}^{ml}$ ($j \geq 0$) in \mathcal{ML}_{DM} can be specified as follows:

$$d_{i,j+1}^{ml} = r_{i,j}^{ml} + \mathcal{V}_i \quad (26.23)$$

$$r_{i,j+1}^{ml} = d_{i,j+1}^{ml} - (\Theta_i(r_{i,0}^{ml}, f_{i,0}^{ml}) + C_i) \quad (26.24)$$

It is clear that $\Theta_i(r_{i,0}^{ml}, f_{i,0}^{ml}) + C_i = f_{i,0}^{ml}$ when $r_{i,0}^{ml} = 0$ ($1 \leq i \leq m$) in \mathcal{ML}_{DM} .

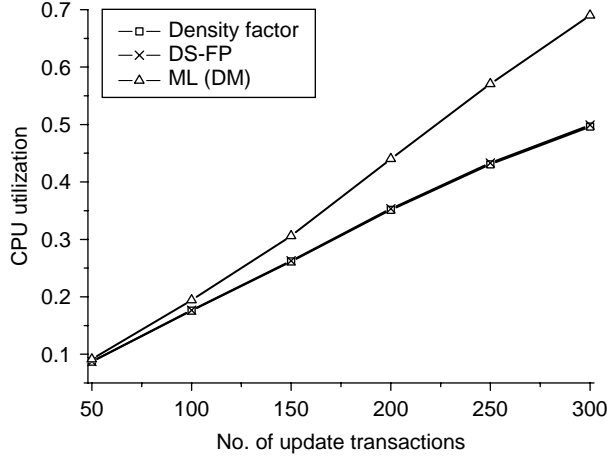


FIGURE 26.7 CPU utilization comparison: DS-FP versus \mathcal{ML}_{DM} .

Following the parameter settings in Ref. 17, we quantitatively compare CPU workloads of update transactions produced by \mathcal{ML}_{DM} and DS-FP in Figure 26.7. We observe that the CPU workload of DS-FP is consistently lower than that of \mathcal{ML}_{DM} . In fact, the difference widens as the number of update transactions increases. Finally, the CPU workload of DS-FP is only slightly higher than the density factor. The improvement of CPU workload under DS-FP is due to the fact that DS-FP adaptively samples real-time data objects at a lower rate.

26.4.4 Schedulability Analysis for DS-FP

The experimental study shows that DS-FP reduces CPU workload resulting from sensor update transactions compared to \mathcal{ML}_{DM} . This section proves that its schedulability is also superior and presents a sufficient condition for its schedulability.

Theorem 26.3

Given a synchronous update transaction set \mathcal{T} with known C_i and \mathcal{V}_i ($1 \leq i \leq m$), if $(\forall i) f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$ in \mathcal{ML}_{DM} , then

$$\text{WCRT}_i \leq f_{i,0}^{ml}$$

where WCRT_i and $f_{i,0}^{ml}$ denote the worst-case response time of τ_i under DS-FP and \mathcal{ML}_{DM} , respectively.

Proof

This can be proved by contradiction. Suppose that τ_k is the highest-priority transaction such that $\text{WCRT}_k > f_{k,0}^{ml}$ holds in DS-FP. Also assume that the response time of $J_{k,n}$ ($n \geq 0$), $R_{k,n}$, is the worst for τ_k in DS-FP. Note that schedules of τ_1 in \mathcal{ML}_{DM} and DS-FP are the same as in both cases, τ_1 jobs have the same relative deadline (C_1) and separation/period ($\mathcal{V}_1 - C_1$). So $1 < k \leq m$ holds.

As $\text{WCRT}_k > f_{k,0}^{ml}$, there must be a transaction τ_l such that (a) τ_l has higher priority than τ_k ($1 \leq l < k$); (b) at least two consecutive jobs of τ_l , $J_{l,j-1}$ and $J_{l,j}$, overlap with $J_{k,n}$, and (c) the separation of $J_{l,j-1}$ and $J_{l,j}$ satisfies the following condition:

$$r_{l,j} - r_{l,j-1} < \mathcal{V}_l - f_{l,0}^{ml} \quad (j > 0) \quad (26.25)$$

where $\mathcal{V}_l - f_{l,0}^{ml}$ is the period (i.e., separation) of jobs of τ_l in \mathcal{ML}_{DM} .

Claim (a) is true because $k > 1$. It is straightforward that if each higher-priority transaction of τ_k only has one job overlapping with $J_{k,n}$, then $R_{k,n} \leq f_{k,0}^{ml}$. This implies that Claim (b) is true. Finally, for $(\forall l < k)$ and $J_{l,j-1}$ and $J_{l,j}$ overlapping with $J_{k,n}$, if

$$r_{l,j} - r_{l,j-1} \geq \mathcal{V}_l - f_{l,0}^{ml} \quad (j > 0)$$

then $R_{k,n} > f_{k,0}^{ml}$ cannot be true because the amount of preemptions from higher-priority transactions received by $J_{k,n}$ in DS-FP is no more than that received by $J_{k,0}$ in \mathcal{ML}_{DM} . Thus, Claim (c) is also true.

We know that release time $r_{l,j}$ in DS-FP is derived as follows:

$$r_{l,j} = d_{l,j} - R_{l,j} \quad (26.26)$$

where $R_{l,j}$ is the calculated response time of job $J_{l,j}$, that is, $\Theta_l(r_{l,j}, d_{l,j}) + C_l$. Following Equations 26.25 and 26.26,

$$\begin{aligned} d_{l,j} - R_{l,j} &= r_{l,j} \quad \{\text{By Equation 26.26}\} \\ &< r_{l,j-1} + \mathcal{V}_l - f_{l,0}^{ml} \quad \{\text{By Equation 26.25}\} \\ &= d_{l,j} - f_{l,0}^{ml} \quad \{\text{By Equation 26.17}\} \end{aligned}$$

Finally,

$$R_{l,j} > f_{l,0}^{ml} \quad (26.27)$$

Equation 26.27 contradicts the assumption that τ_k is the highest-priority transaction that $\text{WCRT}_k > f_{k,0}^{ml}$ holds. Therefore, the theorem is proved.

The following theorem gives a sufficient condition for schedulability of DS-FP.

Theorem 26.4

Given a synchronous update transaction set \mathcal{T} with known C_i and \mathcal{V}_i ($1 \leq i \leq m$), if $(\forall i) f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$ in \mathcal{ML}_{DM} , then \mathcal{T} is schedulable with DS-FP.

Proof

If $f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$, then the worst-case response times of τ_i ($1 \leq i \leq m$) in DS-FP, WCRT_i , satisfy the following condition (by Theorem 26.3):

$$\text{WCRT}_i \leq f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$$

That is, WCRT_i is no more than $\frac{\mathcal{V}_i}{2}$. Because the following three equations hold in DS-FP according to Equation 26.19:

$$r_{i,j} = d_{i,j} - R_{i,j} \quad (26.28)$$

$$d_{i,j+1} = r_{i,j} + \mathcal{V}_i \quad (26.29)$$

$$d_{i,j+1} = r_{i,j+1} + R_{i,j+1} \quad (26.30)$$

Replacing $r_{i,j}$ and $d_{i,j+1}$ in Equation 26.29 with Equations 26.28 and 1.30, respectively, it follows that

$$r_{i,j+1} + R_{i,j+1} = d_{i,j} - R_{i,j} + \mathcal{V}_i$$

That is,

$$r_{i,j+1} - d_{i,j} + R_{i,j+1} + R_{i,j} = \mathcal{V}_i \quad (26.31)$$

Because

$$R_{i,j+1} + R_{i,j} \leq 2 \cdot \text{WCRT}_i \leq \mathcal{V}_i$$

it follows from Equation 26.31 that $r_{i,j+1} - d_{i,j} \geq 0$ holds. This ensures that it is schedulable to schedule two jobs of τ_i in one validity interval \mathcal{V}_i under DS-FP. Thus \mathcal{T} is schedulable with DS-FP.

The following corollary states the correctness of DS-FP.

Corollary 26.1

Given a synchronous update transaction set \mathcal{T} with known C_i and \mathcal{V}_i ($1 \leq i \leq m$), if $(\forall i) f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$ in \mathcal{ML}_{DM} , then DS-FP correctly guarantees the temporal validity of real-time data.

Proof

As deadline assignment in DS-FP follows Equation 26.17, the largest distance of two consecutive jobs, $d_{i,j+1} - r_{i,j}$ ($j \geq 0$), does not exceed \mathcal{V}_i . The *validity constraint* can be satisfied if all jobs meet their deadlines, which is guaranteed by Theorem 26.4.

If \mathcal{T} can be scheduled by \mathcal{ML}_{DM} , then by \mathcal{ML}_{DM} definition $(\forall i) f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$. Thus Corollary 26.2, which states a sufficient schedulability condition for DS-FP, directly follows from Theorem 26.4.

Corollary 26.2

Given a synchronous update transaction set \mathcal{T} with known C_i and \mathcal{V}_i ($1 \leq i \leq m$), if \mathcal{T} can be scheduled by \mathcal{ML}_{DM} , then it can also be scheduled by DS-FP.

26.5 Conclusions

This chapter examines the temporal consistency maintenance problem for real-time update transactions. Assume that deadlines are not greater than their corresponding periods, we investigate the problem with three approaches: \mathcal{ML}_{DM} , $\mathcal{ML}_{\text{EDF}}$, and DS-FP. We also shed light on the theory of DS-FP by presenting a sufficient condition for its schedulability. Our analysis demonstrates that DS-FP is the most efficient approach in terms of minimizing sensor update workload while guaranteeing the validity constraint.

There are still a number of open issues regarding DS-FP, which include:

1. Is time 0 a critical instant for DS-FP?
2. Liu and Layland identified the least upper bound of CPU utilization for *rate monotonic* in Ref. 10. What is the one for DS-FP?
3. Is there a sufficient and necessary condition for the schedulability of DS-FP? If so, what is it?

These problems need to be studied for understanding DS-FP better.

References

1. S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor," *IEEE Real-Time Systems Symposium*, December 1990.
2. A. Burns and R. Davis, "Choosing Task Periods to Minimise System Utilisation in Time Triggered Systems," *Information Processing Letters*, 58, 223–229, 1996.

3. D. Chen and A. K. Mok, "Scheduling Similarity-Constrained Real-Time Tasks," *ESA/VLSI*, pp. 215–221, 2004.
4. T. Gustafsson, and J. Hansson, "Data Management in Real-Time Systems: A Case of On-Demand Updates in Vehicle Control Systems," *IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 182–191, 2004.
5. F. S. Hiller and G. J. Lieberman, "Introduction to Operations Research," McGraw-Hill Publishing Company, New York, 1990.
6. S. Ho, T. Kuo, and A. K. Mok, "Similarity-Based Load Adjustment for Real-Time Data-Intensive Applications," *IEEE Real-Time Systems Symposium*, 1997.
7. K. D. Kang, S. Son, J. A. Stankovic, and T. Abdelzaher, "A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases," *EuroMicro Real-Time Systems Conference*, June 2002.
8. K. Y. Lam, M. Xiong, B. Liang, and Y. Guo, "Statistical Quality of Service Guarantee for Temporal Consistency of Real-Time Data Objects," *IEEE Real-Time Systems Symposium*, 2004.
9. J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, 2, 237–250, 1982.
10. C. L. Liu, and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, 20(1), 46–61, 1973.
11. D. Locke, "Real-Time Databases: Real-World Requirements," in *Real-Time Database Systems: Issues and Applications*, edited by A. Bestavros, K.-J. Lin, and S. H. Son, Kluwer Academic Publishers, Boston, Massachusetts, pp. 83–91, 1997.
12. K. Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases* 1, 199–226, 1993.
13. X. Song and J. W. S. Liu, "Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control," *IEEE Transactions on Knowledge and Data Engineering*, 7(5), 786–796, 1995.
14. J. A. Stankovic, S. Son, and J. Hansson, "Misconceptions About Real-Time Databases," *IEEE Computer*, 32(6), 29–36, 1999.
15. J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, "Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms," Kluwer Academic Publishers, Boston, Massachusetts, 1998.
16. M. Xiong, S. Han, and D. Chen, "Deferrable Scheduling for Temporal Consistency: Schedulability Analysis and Overhead Reduction," *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.
17. M. Xiong, S. Han, and K. Y. Lam, "A Deferrable Scheduling Algorithm for Real-Time Transactions Maintaining Data Freshness," *IEEE Real-Time Systems Symposium*, 2005.
18. M. Xiong and K. Ramamritham, "Deriving Deadlines and Periods for Real-Time Update Transactions," *IEEE Transactions on Computers*, 53(5), 567–583, 2004.
19. M. Xiong, K. Ramamritham, J. A. Stankovic, D. Towsley, and R. M. Sivasankaran, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *IEEE Transactions on Knowledge and Data Engineering*, 14(5), 1155–1166, 2002.
20. M. Xiong, Q. Wang, and K. Ramamritham, "On Earliest Deadline First Scheduling for Temporal Consistency Maintenance," *Bell Labs Technical Report*, 2006. <http://db.bell-labs.com/user/xiong/xiong-edf06.pdf>.

27

Salvaging Resources by Discarding Irreconcilably Conflicting Transactions in Firm Real-Time Database Systems*

Victor C. S. Lee
City University of Hong Kong
Joseph Kee-Yin Ng
Hong Kong Baptist University
Ka Man Ho
City University of Hong Kong

27.1	Introduction	27-1
27.2	Related Work	27-3
	Dynamic Adjustment of Serialization Order • OCC-APR	
27.3	A New Priority Cognizant CC Algorithm	27-5
	Design Rationale of OCC-ED • Details of OCC-ED	
27.4	Experiments	27-6
	Simulation Model • Performance Metrics	
27.5	Results	27-8
	High Data Contention • Low Resource Contention	
27.6	Conclusion	27-13

27.1 Introduction

There are time constraints expressed in terms of deadlines in real-time database systems (RTDBS) [3,7,17,19]. In *firm* RTDBS, transactions are required to commit before their deadlines or else the late results are useless. For example, in electronic stock trading systems, transactions that fail to meet deadlines may lead to a loss of opportunities to trade and make profit. To allow more transactions to finish before their deadlines, for scheduling purpose, a higher priority may be given to transactions with an earlier deadline. Recently, there are many research works on RTDBS. In particular, concurrency control mechanisms for maintaining database consistency in RTDBS attracted significant attention because integrating concurrency control with real-time requirements is nontrivial.

*The work described in this chapter was substantially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1204/03E and HKBU 2174/03E].

In the literature, there are two basic approaches to concurrency control [1]: locking-based and optimistic approach [9]. In Refs. 5, 6, and 13, their works showed that optimistic concurrency control (OCC) performs better in firm RTDBS. In OCC, data conflicts are detected at the validation phase of a transaction. Conflict resolution is based on the adopted strategy. For instance, in OCC-broadcast commit (OCC-BC), the validating transaction is guaranteed to commit while the conflicting transactions have to be restarted. The good performance of OCC-BC can be explained by the *fruitful transaction restarts*. That is, the restart of the conflicting transactions must lead to a commit of the validating transaction. However, in locking-based mechanism, the transaction that blocks a number of conflicting transactions may eventually miss its deadline while the blocked transactions may very likely miss their deadlines as well. This results in a lose-lose situation.

In recent years, many researchers have introduced different OCC algorithms to get a better performance in RTDBS [5,7,8,11]. In these works, they proposed to modify the OCC algorithm by incorporating priority mechanisms in conflict resolution. However, their results showed that the effect is not remarkable as compared to priority incognizant variants. One common objective in these previous works is to increase the likelihood of win-win situation for both the validating transaction and the conflicting transactions to meet their deadlines in case of data conflict. No matter the validating transaction or the conflicting transactions are chosen to restart by the adopted conflict resolution strategy, the rationale behind the decision is to increase the number of transactions meeting deadlines. In most of the algorithms, the decision is based on the number and the relative priorities of transactions involved in the data conflicts. Obviously, it is not always the case that both parties can eventually meet their deadlines. Some transactions may not meet their deadlines and the resources consumed by these transactions are wasted, leading to a more resource-contended situation. Since transactions in RTDBS are time constrained, compared to conventional non-real-time database systems, it is more important for any concurrency control algorithms to minimize the wastage of resources. However, there is no intention in their works to reduce such a waste once a conflict resolution decision is made and fruitful transaction restarts can be expensive.

One major performance barrier in OCC algorithms is the transaction restart problem. On the one hand, restarting transactions resolve data conflicts. On the other hand, extra resources are required to process the restarted transactions to see whether they can meet their deadlines. This problem is particularly serious in OCC algorithms because of the late conflict detection mechanism that incurs heavy restart overhead as transactions are restarted after data accesses. In Refs. 12 and 14, a technique called *dynamic adjustment of serialization order* is devised to adjust the relative positions of the conflicting transactions in the serialization order with respect to the validating transaction. In Refs. 10, 12, 15, and 18, a protocol called *OCC with Time Interval* (OCC-TI) based on the above technique and its extensions is proposed. The results showed that the technique is effective in reducing the number of *unnecessary transaction restarts* and a better performance than conventional and priority conscious OCC algorithms can be obtained [3].

We observe that system performance is mostly bounded by resource contention, not data contention. In a heavily loaded system most transactions are waiting for resources, instead of performing data operations. Therefore, the number of transactions missing their deadlines increases in spite of a decrease in the number of transaction restarts. From our observations, most of the existing algorithms are only effective in the system dominated by data contention. When the state of the system shifts from data contention to resource contention as workload increases, the algorithms cease to function effectively. To push the performance limits of concurrency control mechanisms further, it is inadequate for them to focus on data conflict resolution. That is, in addition to choosing the more fruitful transactions to restart, it is also important to determine how likely the chosen transactions can meet their deadlines after restart. Otherwise, resources consumed by the restarted transactions are wasted and other ongoing transactions are deprived of these resources. This work was motivated by the above observations. We aim at soothing resource contention by proposing a mechanism without unrealistic assumptions to identify and discard those restarted transactions that are not likely to commit before their deadlines. On the one hand, the cost of sacrifices is reduced. On the other hand, more resources can be made available to other ongoing transactions.

This chapter is organized as follows. Section 27.2 describes the two most effective OCC variants, namely OCC-TI and OCC-APR (*OCC-Adaptive-Priority*), in the literature. An extension of OCC-APR called

OCC-ED (Early Discard) is proposed in Section 27.3. The design and implementation issues of the protocol are described in Section 27.4. Section 27.5 discusses the simulation results. Finally, a conclusion is presented in Section 27.6.

27.2 Related Work

In this section, we describe the related work [2,12] on which OCC-ED is based.

27.2.1 Dynamic Adjustment of Serialization Order

It is well known that unnecessary transaction restart is the major performance bottleneck in OCC algorithms. The following example illustrates the problem.

Example

Let $r_i[x]$ and $w_i[x]$ represent read and write operation on data item x by transaction T_i . v_i and c_i indicate the validation and commit operations of transaction T_i , respectively. Consider transactions T_1 , T_2 , and T_3 , and the schedule H :

$$\begin{aligned} T_1 &: r_1[x], w_1[x], r_1[y], w_1[y], v_1, c_1 \\ T_2 &: r_2[y], w_2[y], \dots, v_2 \\ T_3 &: r_3[x], \dots, v_3 \\ H &= r_1[x], w_1[x], r_2[y], w_2[y], r_3[x], r_1[y], w_1[y], v_1, c_1, \dots \end{aligned}$$

By using the conventional OCC with forward validation (OCC-FV) mechanism [4], when T_1 enters the validation phase, T_2 and T_3 will be aborted and restarted as they have conflicts with the validating transaction, T_1 , on data items y or x , respectively. However, consider the data conflict between T_1 and T_3 . T_3 does not need to be aborted if there is no further data conflict with T_1 because T_3 can be considered serialized before T_1 . In contrast, there are read–write and write–write conflicts between T_1 and T_2 and there is no way for T_2 to be serialized with T_1 . Therefore, T_2 has to be aborted to maintain the serializability. In this case, the restart of T_3 is regarded as *unnecessary restart*. To reduce the number of unnecessary restarts, the idea of *dynamic adjustment of serialization order* has been introduced and a protocol called OCC-TI has been proposed in Ref. 12.

In OCC-TI, each data item is associated with a read and a write timestamp, and each transaction has a timestamp interval expressed as lowerbound (lb) and upperbound (ub). The interval is used to determine the serialization order of the transactions. In the beginning, the timestamp interval of each transaction is initialized to $[0, \infty)$. During the execution of a transaction, the interval will be adjusted from time to time to reflect the serialization dependency among concurrent transactions. The interval of a transaction may be adjusted when the transaction accesses a data item or another transaction is validating. In the latter case, the conflicting transactions with the validating transaction are classified as *IRreconcilably Conflicting Transactions* (IRCTs) if the interval shuts out ($lb > ub$) after adjustment. Otherwise, they are classified as *reconcilably conflicting transactions* (RCTs). To commit the validating transaction, all IRCTs have to be aborted and restarted to resolve the data conflicts while RCTs are allowed to proceed with the adjusted intervals. Interested readers may refer to Ref. 12 for detailed description of the dynamic adjustment of serialization order mechanism.

27.2.2 OCC-APR

OCC-APR [2] is based on the technique of dynamic adjustment of serialization order. It is a deadline-sensitive algorithm. It considers transaction deadlines in conflict resolution. If a validating transaction (VT) conflicts with two or more IRCTs with earlier deadlines, the time required for reexecuting the

validating transaction will be estimated. If there is sufficient slack time, the validating transaction will be restarted and all the conflicting transactions will be allowed to proceed. Otherwise, the logic of OCC-TI will be followed. That is, the validating transaction will commit, the IRCTs are restarted, and the timestamp intervals of the RCTs are adjusted accordingly. The rationale behind this protocol is to ensure that either the validating transaction can commit or there is some chance for more than one conflicting transaction with earlier deadlines to commit by restarting the validating transaction with sufficient slack for reexecution. Denote the deadline and the estimated restart time of a transaction T by $\text{deadline}(T)$ and $\text{restart_time}(T)$, respectively, the rule to decide whether the validating transaction can commit or not is as follows:

The restart rule of VT:

```

if  $\exists T_i, T_j \in \text{IRCT}(VT)$ 
  if ( $\text{deadline}(T_i) < \text{deadline}(VT)$  and  $\text{deadline}(T_j) < \text{deadline}(VT)$ ) then
    if ( $\text{restart\_time}(VT) \leq (\text{deadline}(VT) - \text{current\_time})$ )
      restart  $VT$ 

```

The determinant factor of this rule is the accuracy of the estimated restart time of the validating transaction. If it is underestimated, it may lead to a lose–lose situation when both the conflicting transactions and the restarted validating transaction miss their deadlines. On the contrary, a win–win situation may be resulted if both parties can meet their deadlines.

27.2.2.1 Estimating Restart Time of Validating Transaction

The processing of a transaction can be abstracted into a number of critical timings, namely, CPU waiting time (cpu_waiting_time), CPU processing time (cpu_time), disk waiting time (disk_waiting_time), and disk processing time (disk_time). Thus, an estimation of the time to process a transaction, $\text{process_time}(T)$, is as follows:

$$\text{process_time}(T) = \#_of_operations(T) \times (\text{cpu_waiting_time} + \text{cpu_time} + \text{disk_waiting_time} + \text{disk_time}) \quad (27.1)$$

where $\#_of_operations(T)$ is the number of operations in transaction T , which is known when the transaction reaches its validation phase.

In addition, it can be observed that all the requested data items are already loaded into the memory at the validation of the transaction. So, it is not unreasonable to assume that the data items are still in the memory when the transaction is restarted. Therefore, the transaction does not need to access the disk again during its reexecution. Based on this assumption, the restart time of the validating transaction, VT , can be estimated as follows:

$$\text{restart_time}(VT) = \#_of_operations(VT) \times (\text{cpu_waiting_time} + \text{cpu_time}) \quad (27.2)$$

27.2.2.2 Calculating the Adaptive Factor

The CPU waiting time is affected by the system loading. An adaptive factor α , is introduced to capture the impact of system loading on CPU waiting time and the revised formula for estimating the restart time is given below:

$$\text{restart_time}(VT) = \#_of_operations(VT) \times ((\alpha \times \text{cpu_waiting_time}) + \text{cpu_time}) \quad (27.3)$$

In general, there is an increasing number of transactions missing deadlines in a heavily loaded system. So, the value of α is adjusted according to the proportion of transactions missing their deadlines in a sampling period. Initially, α is set to one. If the proportion of transactions missing their deadlines does not decrease compared to the last sampling period, α is increased incrementally and a longer time estimation will be

obtained for restarting the validating transaction in a heavily loaded system. Otherwise, α is decreased and a shorter time estimation will be obtained in a light loading condition.

27.3 A New Priority Cognizant CC Algorithm

In this section, we introduce our new CC algorithm for RTDBS, namely, OCC-ED (Early Discard). Similar to OCC-APR, OCC-ED is also based on dynamic adjustment of serialization order. The design rationale of this algorithm is to reduce the number of transaction restarts and to ease contention in the system, so that more resources are made available and more transactions can commit before their deadlines.

27.3.1 Design Rationale of OCC-ED

OCC-ED can be regarded as an extension of OCC-APR. OCC-ED further exploits the restart time estimation mechanism in OCC-APR. However, the objectives are very different. In OCC-APR, the restart time estimation is used to provide a level of certainty for the validating transaction to have enough time for reexecution in making the conflict resolution decision. However, in a resource-contended situation, OCC-APR becomes less effective because in a heavily loaded system, the conflict resolution mechanism will always prefer to commit the validating transaction and restart the conflicting transactions.

In OCC-ED, we apply the restart time estimation mechanism in those victim transactions. When the validating transaction is allowed to commit the restart time of those IRCTs will be estimated. IRCT will be allowed to restart only if there is sufficient time for them to reexecute. Otherwise, they will be removed from the system even though their deadlines have not been missed yet. Since we do not assume any prior knowledge of the data access pattern of a transaction, the estimation is solely based on the operations already performed by the IRCT at the time of estimation. This pessimistic estimation ensures that even transactions with only little chance to commit will not be discarded.

The objective of OCC-ED is to *early discard* those IRCTs with no hope to commit from the system such that more resources can be salvaged for other executing transactions. Note that this mechanism is applied when the conflict resolution decision is to commit the validating transaction.

27.3.2 Details of OCC-ED

OCC-ED contains four major processes. They are (1) validation phase process (VAL), (2) timestamp interval adjustment in validation phase process (TAV), (3) timestamp interval adjustment in read phase process (TAR), and (4) early discard process (ED). The first three processes are similar to those of OCC-APR with minor modifications for invoking the process ED.

A transaction performs the process VAL in its validation phase. In VAL, it calls TAV to tentatively adjust the timestamp intervals of the conflicting transactions. After adjustment, the transactions will be divided into two types—RCT and IRCT. If there are two or more IRCTs with earlier deadlines than that of the validating transaction, then the reexecution time of the validating transaction will be estimated. If the remaining slack time of the validating transaction is enough for it to reexecute all its operations, the validating transaction will be restarted. In this case, the RESET procedure will be called to restore the timestamp intervals of all the conflicting transactions to their original values before adjustment.

However, if there is only one IRCT with earlier deadline than that of the validating transaction or the remaining slack time of the validating transaction is less than the estimated reexecution time, the validating transaction will be committed to guarantee that at least one transaction can meet its deadline. Consequently, the process ED will be performed on each IRCT. The process ED determines whether there is enough remaining slack time for the IRCT to reexecute. Since the estimation is based on the operations already performed by the IRCT at present, a positive outcome from the process ED means that there is sufficient time for the IRCT to reexecute at least the operations already performed so far. If the remaining slack time of the IRCT is greater than the estimation, it will be restarted. Otherwise, it will be discarded from the system even though the deadline is not missed yet.

The process TAR adjusts the timestamp interval of a transaction when it accesses data items in the read phase. If the timestamp interval shuts out, the process ED will also be called to determine whether the transaction should be restarted or early discarded.

The following pseudocodes describe the detailed mechanism of the processes. For the sake of completeness, the pseudocodes of OCC-APR are also included. For detailed explanation of the part on OCC-APR, readers may refer to Ref. 2. $CS(VT)$ is the set of transactions that conflict with VT . $WS(T)$ and $RS(T)$ stand for the write set and read set of a transaction T , respectively. $RTS(D)$ and $WTS(D)$ are the read timestamp and write timestamp of a data item D , respectively. $TI(VT)$ and $FTS(VT)$ stand for the timestamp interval and final timestamp of VT , respectively.

27.4 Experiments

To evaluate the performance of OCC-ED and compare with that of OCC-APR, we built a simulation model of a RTDBS [2] with CSIM18 [16].

27.4.1 Simulation Model

Table 27.1 lists the parameters used and their default values in the simulation model. The model consists of NumCPU CPUs and NumDisk disks that operate in a disk-resident database. DB_Size specifies the number of data items in the database. We assumed that data items are uniformly distributed among the disks. Each disk has its own queue but there is only one queue for all CPUs. The earliest deadline first (EDF) scheduling policy is applied to all queues.

Process VAL(VT):

Process is called when a transaction VT enters validation phase

$HPICount = 0$;

$FTS(VT) = \text{lower bound of } TI(VT)$;

for each $T_i \in CS(VT)$

$TAV(T_i)$;

if $TI(T_i) = \text{null}$ **then**

if $\text{deadline}(T_i) < \text{deadline}(VT)$ **then**

$HPICount = HPICount + 1$

if $(HPICount = 2)$ **and** $(\text{restart_time}(VT) \leq (\text{deadline}(VT) - \text{current_time}))$ **then**

$\text{RESET}(CS(VT))$;

$\text{restart } VT$;

exit ;

for each $D_j \in RS(VT)$

if $RTS(D_j) < FTS(VT)$ **then**

$RTS(D_j) = FTS(VT)$;

for each $D_j \in WS(VT)$

if $WTS(D_j) < FTS(VT)$ **then**

$WTS(D_j) = FTS(VT)$;

$\text{ED}(\text{Transactions marked for restart})$;

$\text{commit } VT$;

end.

Process TAV(T):

Process is called by process VAL when a transaction T conflicts with VT

for each $D_j \in RS(VT)$

if $D_j \in WS(T)$ **then**

$TI(T) = TI(T) \cap [FTS(VT), \infty)$;

```

for each  $D_j \in WS(VT)$ 
  if  $D_j \in RS(T)$  then
     $TI(T) = TI(T) \cap [0, FTS(VT) - 1]$ ;
  if  $D_j \in WS(T)$  then
     $TI(T) = TI(T) \cap [FTS(VT), \infty)$ ;
if  $TI(T) = \text{null}$  then
  mark  $T$  for restart
end.

```

Process TAR(T):

Process is called during T 's read phase, when it wants to access a data item D

```

if  $T$  wants to read  $D$  then
  read( $D$ );
   $TI(T) = TI(T) \cap [WTS(D), \infty)$ ;
else
  write  $D$  in its local buffer;
   $TI(T) = TI(T) \cap [WTS(D), \infty) \cap [RTS(D), \infty)$ ;
if  $TI(T) = \text{null}$  then
  ED( $T$ )
end.

```

Process ED(T):

Process is called when T 's timestamp interval shuts out

```

if ( $\text{current\_number\_of\_operations}(T) \times ((\alpha \times \text{cpu\_waiting\_time}) + \text{cpu\_time}))$ 
   $\leq (\text{deadline}(T) - \text{current\_time})$  then
  mark  $T$  for restart
else
  mark  $T$  for early discard
end.

```

Transactions are modeled as a sequence of read/write operations that can be varied from read-only to write-only. The proportion depends on WriteProb. The number of data items accessed by a transaction T , $\text{size}(T)$, is drawn from a normal distribution in the range specified by SizeInterval. Each transaction, T , has a deadline, that is assigned as $\text{deadline}(T) = \text{arrival_time}(T) + \text{size}(T) \times (\text{ProcCPU} + \text{ProcDisk}) \times SR$, where SR is the slack ratio uniformly chosen from the range specified by SRInterval.

There is a buffer pool in the system. A transaction may find the data item it wants to read in the buffer with a probability BuffProb. If the data item is not in the buffer, the transaction has to visit the disk

TABLE 27.1 Model Parameters and Their Default Settings

Parameters	Value	Remarks
NumCPU	2	Number of CPUs
NumDisk	4	Number of disks
ProcCPU	10 ms	CPU processing time
ProcDisk	20 ms	Disk processing time
DB_Size	100	Number of data items in the database
ArrivalRate	N/A	Transaction arrival rate
WriteProb	0.25	Probability of write access
BuffProb	0.5	Probability of buffer hit
SizeInterval	[5, 25]	Number of data items accessed per transaction
SRInterval	[2, 8]	Slack ratio

for reading the data item. For a restarted transaction, we assume that all the data items to be accessed are already loaded into the buffer in previous execution. Therefore, the value of BuffProb for restarted transactions is one.

27.4.2 Performance Metrics

The primary performance metric is deadline miss percentage (DMP). DMP is the percentage of transactions that miss their deadlines. For OCC-ED, transactions that are early discarded from the system are also treated as missing their deadlines and are included in this metric. Hence, we have

$$\text{DMP} = \frac{\text{Trans}_{\text{Miss}}}{\text{Trans}_{\text{Total}}} \times 100\% \quad (27.4)$$

where $\text{Trans}_{\text{Miss}}$ is the number of transactions missing their deadlines and $\text{Trans}_{\text{Total}}$ the total number of transactions entering the system.

Transaction restart is a crucial factor in system performance. Another metric that can demonstrate the effectiveness of the algorithms is restart rate (RR).

$$\text{RR} = \frac{\text{Trans}_{\text{Restart}}}{\text{Trans}_{\text{Total}}} \quad (27.5)$$

where $\text{Trans}_{\text{Restart}}$ is the number of transactions being restarted and $\text{Trans}_{\text{Total}}$ the total number of transactions entering the system.

To measure the amount of resources that can be salvaged from early-discarded transactions for other executing transactions, we collect the average CPU response time which is the average time that a transaction needs to wait before getting service from the system.

The early discard effective rate (EDER) is used to measure the effectiveness of the early discard mechanism in OCC-ED. This metric also helps to understand the overhead of the mechanism.

$$\text{EDER} = \frac{\text{Trans}_{\text{Discard}}}{n} \quad (27.6)$$

where $\text{Trans}_{\text{Discard}}$ is the number of transactions being early discarded and n the number of times the early discard mechanism is invoked.

In the following simulation experiments, all the results are collected after 5000 transactions are completed, where the relative error of the mean transaction response time at 90% confidence level is less than 10%.

27.5 Results

We have conducted a series of experiments to evaluate the performance of OCC-ED as compared to OCC-APR. In this section, we discuss the results from the experiments.

Figures 27.1 and 27.2 show the deadline miss percentage and the restart rate of both algorithms, respectively. When the system loading is light (< 6 transactions per second), the deadline miss percentages of both algorithms are very low. It can be observed that a transaction being restarted can still meet its deadline in light loading. As the loading increases, the number of transactions being restarted owing to data conflicts increases and consequently, more transactions miss their deadlines.

When the loading reaches at around 8 transactions per second, the deadline miss percentage of OCC-APR exceeds that of OCC-ED. In Figure 27.2, the restart rate of OCC-APR starts to drop at this loading.

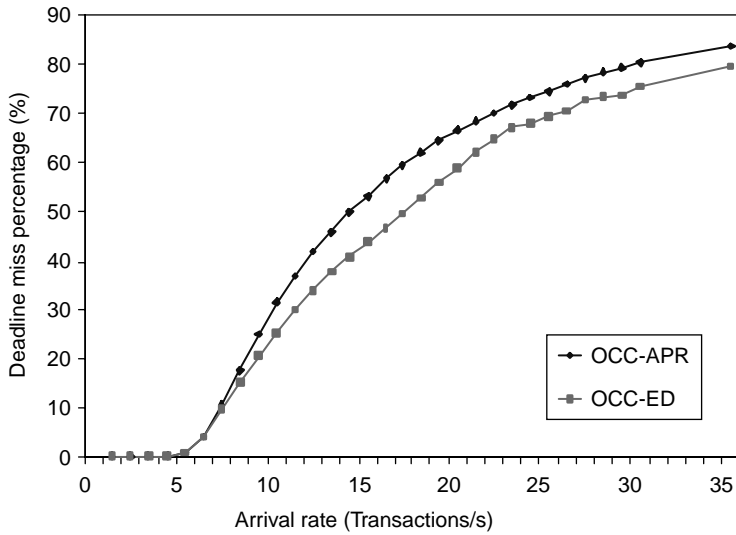


FIGURE 27.1 Deadline miss percentage versus arrival rate.

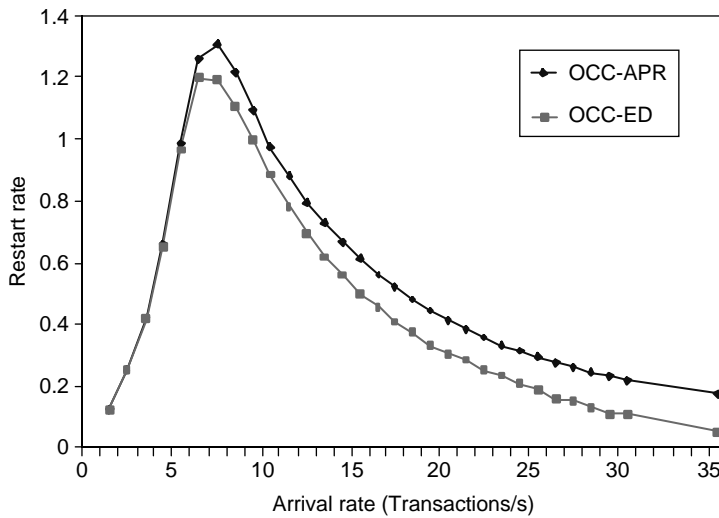


FIGURE 27.2 Restart rate versus arrival rate.

It implies that more transactions miss their deadlines owing to resource contention rather than data contention. The system is congested with newly arrived transactions, and restarted transactions, and it becomes more difficult for both parties to meet their deadlines in such a situation. Moreover, it is likely that those restarted transactions ultimately miss their deadlines. It is because in a resource-contended environment, it takes a longer time for a transaction to execute each operation and the remaining slack time may not be sufficient for it to reexecute all its operations. The resources consumed by them are therefore wasted. However, in case of OCC-ED, some transactions that conflict with the validating transaction are chosen to discard early from the system. So, the restart rate is lower than that of OCC-APR from this point onward. Although some transactions are early discarded and regarded as missing deadlines, the deadline

miss percentage of OCC-ED is lower than that of OCC-APR. It confirms that the sacrifice of these early discarded transactions is fruitful. Resources are salvaged and more transactions are able to meet their deadlines.

Figure 27.3 shows the average CPU response time of both algorithms. The figure clearly shows that the time for a transaction to wait for the CPU is shorter in OCC-ED as compared to OCC-APR, especially in the range of medium loading. It implies that the early discard mechanism can successfully save resources by discarding those IRCTs and the saved resources can in turn help reduce the waiting time of other ongoing transactions, leading to a lower deadline miss percentage. That is, hopeless transactions are refrained from occupying the resources and at the same time consuming the slack time of other

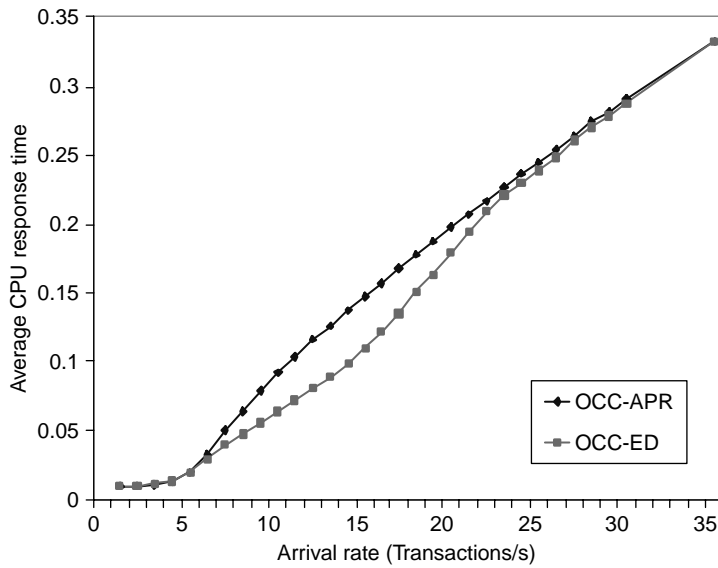


FIGURE 27.3 CPU response time.

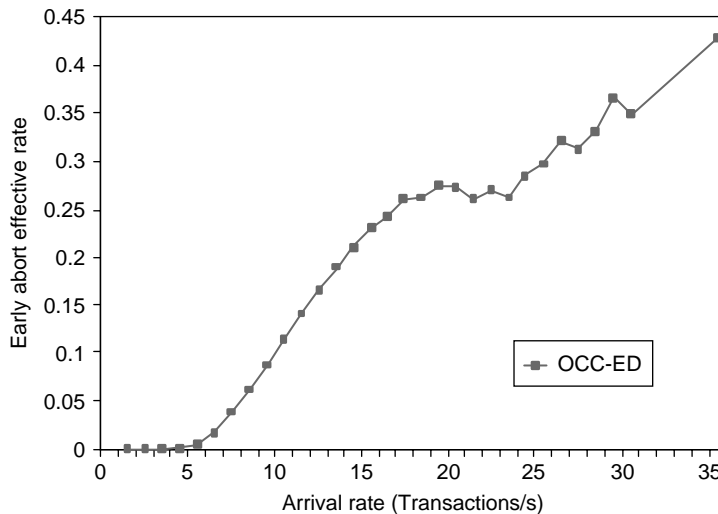


FIGURE 27.4 Early discard effective rate.

ongoing transactions. Hence, resources can be utilized effectively on transactions that are more likely to commit before their deadlines. Conclusively, OCC-ED improves resource availability and effective use of resources.

Figure 27.4 shows the effectiveness of the early discard mechanism in OCC-ED. In light loading, data contention is low and the frequency of invoking the early discard mechanism is also low. As the loading increases, more transactions will be early discarded. For instance, when the system loading is heavy (>25 transactions per second), more than 30% of the transactions will be early discarded instead of restarted. It shows that the early discard mechanism is effective in a resource-contended environment.

27.5.1 High Data Contention

Although the primary objective of OCC-ED is to soothe resource contention, it may be effective in a high data contention environment. On the one hand, OCC-ED helps alleviate the problem of transaction restarts owing to data conflicts because only those IRCTs with some chances to commit are allowed to restart. Hopeless transactions will be discarded. On the other hand, it may help reduce the cascading effect of data conflicts because IRCTs with respect to the same validating transactions are likely to have data conflicts again in the restarted runs. Discarding one or more of the conflicting transactions may reduce the chance of having further data conflicts and improve the chance for the surviving transactions to commit before their deadlines. In this experiment, we want to study the performance of the algorithms in a high data contention environment by setting the write probability of transactions to 0.9. A higher proportion of write operations in a transaction will increase the probability of data conflicts.

Figures 27.5 and 27.6 show the deadline miss percentage and restart rate of the transactions, respectively. Similar to the last experiment, OCC-ED has a better performance than OCC-APR and the results confirm our discussion above. However, we observe a few differences. Transactions start to miss their deadlines in a lighter loading and the highest restart rate is doubled. One obvious reason is that more transactions will be in data conflict with the validating transaction owing to a higher proportion of write operations. Also, in a light loading, there may be sufficient slack time left when data conflicts occur among transactions in the first few instances. This allows the validating transaction or the IRCTs to restart once or more than once.

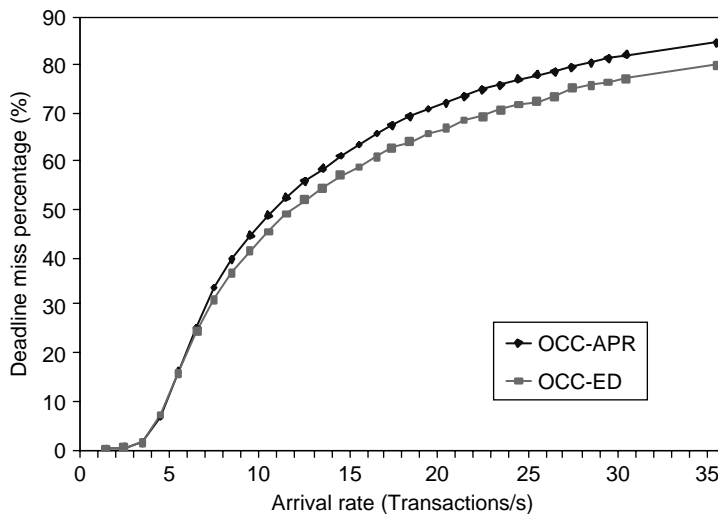


FIGURE 27.5 Deadline miss percentage (high data contention).

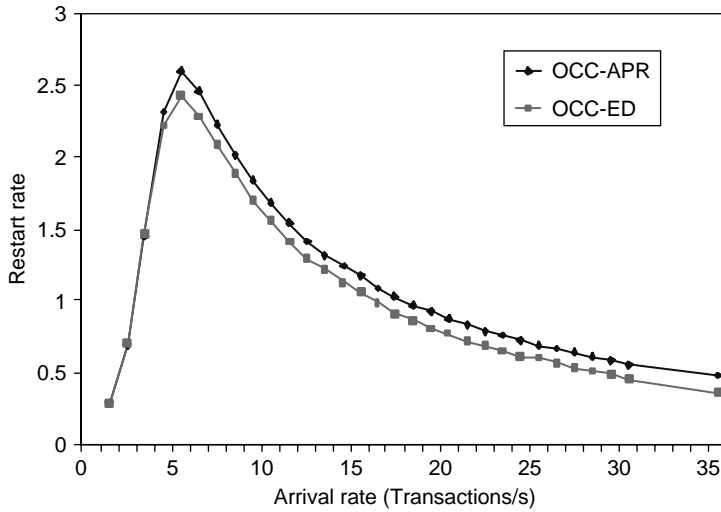


FIGURE 27.6 Restart rate (high data contention).

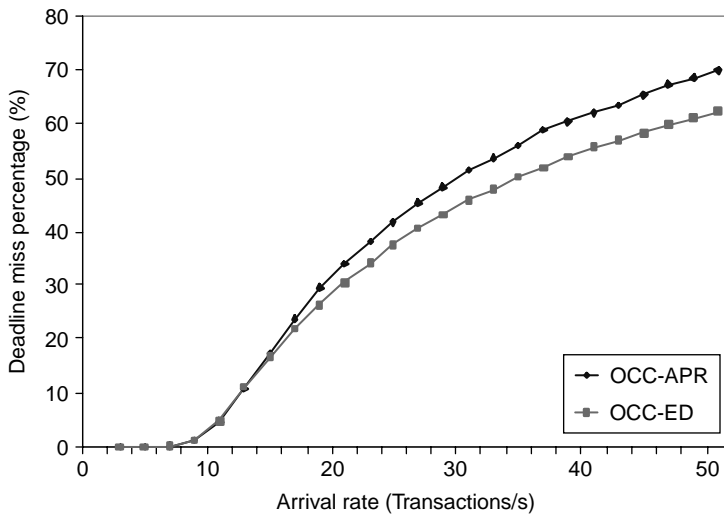


FIGURE 27.7 Deadline miss percentage (more resources).

However, as the loading increases, the early discard mechanism becomes effectual, reducing both restart rate and deadline miss percentage.

27.5.2 Low Resource Contention

Putting more resources may be an alternative to soothe resource contention. Figures 27.7 and 27.8 show the deadline miss percentage and restart rate of OCC-APR and OCC-ED when the number of CPU and disk are increased to 5 and 10, respectively. However, it can be observed that the graphs show a similar pattern. OCC-ED outperforms OCC-APR, though the occurrence shifts to a higher transaction arrival rate. For instance, with more resources, the restart rates reach their peaks at an arrival rate of 10 transactions per second as compared to 5 transactions per second when fewer resources are available. On the whole, the relative performance of the two algorithms remains with no change in spite of additional resources.

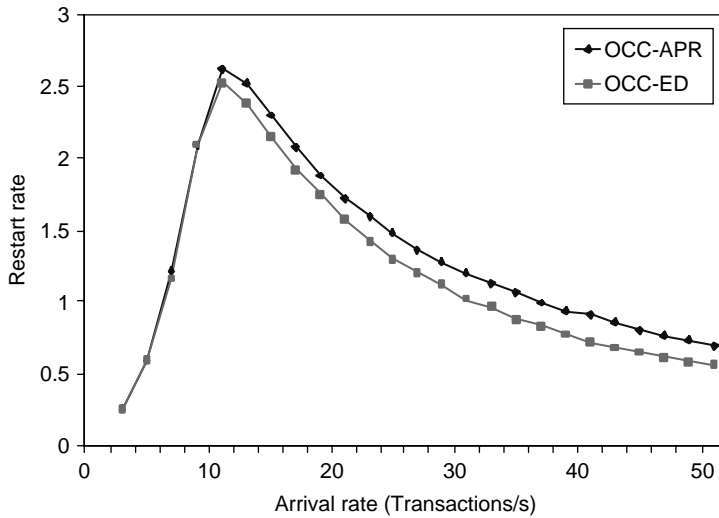


FIGURE 27.8 Restart rate (more resources).

27.6 Conclusion

We observe that one main bottleneck to the performance of optimistic concurrency control protocols in RTDBS is the ineffective use of resources on restarted transactions that ultimately miss their deadlines. In this work, we extend a recently proposed algorithm, OCC-APR, to another algorithm called OCC-ED. In OCC-ED, an IRCT with the validating transaction will be early discarded if the remaining slack time is insufficient for it to reexecute. To avoid discarding any transaction with a chance to commit we adopt a pessimistic approach by estimating the time required to reexecute only the operations that are already performed by the transaction so far. A simulation model is built to evaluate the performance of OCC-ED as compared to OCC-APR. The results show that OCC-ED can successfully salvage resources by early discarding those transactions with no hope to commit. The benefit we get from these salvaged resources is that more transactions can meet their deadlines and thus, improve the system performance.

References

1. Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA (1987).
2. Datta, A. and Son, S.H. Is a Bird in the Hand Worth More than Two in the Bush? Limitations of Priority Cognizance in Conflict Resolution for Firm Real-Time Database System. *IEEE Transactions on Computers*, vol. 49, no. 5, May (2000), 482–502.
3. Datta, A. and Son, S.H. A Study of Concurrency Control in Real-Time, Active Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 3, May–June (2002), 465–484.
4. Haerder, T. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, vol. 9, no. 2 (1984).
5. Haritsa, J.R., Carey, M.J., and Livny, M. Dynamic Real-Time Optimistic Concurrency Control. *Proceedings of 11th IEEE Real-Time Systems Symposium*, Dec. (1990), 94–103.
6. Haritsa, J.R., Carey, M.J., and Livny, M. On Being Optimistic about Real-Time Constraints. *Proceedings of the 9th ACM SIGACT-SIGMOD- SIGART Symposium on Principles of Database Systems*, (1990), 331–343.
7. Haritsa, J.R., Carey, M.J., and Livny, M. Data Access Scheduling in Firm Real-Time Database Systems. *Real-Time Systems*, vol. 4, no. 3 (1992), 203–241.

8. Huang, J., Stankovic, J.A., Ramamritham, K., and Towsley, D.F. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. *Proceedings of the 17th International Conference on Very Large Data Bases*, Sept. (1991), 35–46.
9. Kung, H.T. and Robinson, J.T. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, vol. 6, no. 2 (1981), 213–226.
10. Lam, K.-W., Lam, K.-Y., and Hung, S.-L. Real-Time Optimistic Concurrency Control Protocol with Dynamic Adjustment of Serialization Order. *Proceedings of the Real-Time Technology and Applications Symposium*, May (1995), 174–179.
11. Lam, K.-Y., Kuo, T.-W., Kao, B., Lee, T.S.H., and Cheng, R. Evaluation of Concurrency Control Strategies for Mixed Soft Real-Time Database Systems. *Information Systems*, vol. 27, no. 2 (2002), 123–149.
12. Lee, J. and Son, S.H. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. *Proceedings of IEEE Real-Time Systems Symposium*, Dec. (1993), 66–75.
13. Lee, J. and Son, S.H. Performance of Concurrency Control Algorithms for Real-Time Database Systems. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, Kumar, V. (ed.), Prentice-Hall, New York (1996), 429–460.
14. Lin, Y. and Son, S.H. Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order. *Proceedings of the 11th Real-Time Systems Symposium*, Dec. (1990), 104–112.
15. Lindstrom, J. Extensions to Optimistic Concurrency Control with Time Intervals. *Proceedings of 7th International Conference on Real-Time Computing Systems and Applications*, Dec. (2000), 108–115.
16. Schwetman, H.D. CSIM18—The Simulation Engine. *Proceedings of Simulation Conference*, Dec. (1996), 517–521.
17. Stankovic, J.A., Son, S.H., and Hansson, J. Misconceptions about Real-Time Databases. *Computer*, vol. 32, no. 6, June (1999), 29–36.
18. Wang, Y., Wang, Q., Wang, H., and Dai, G. Dynamic Adjustment of Execution Order in Real-Time Databases. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Apr. (2004), 87.
19. Yu, P.S., Wu, K.-L., Lin, K.-J., and Son, S.H. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, vol. 82, no. 1, Jan. (1994), 140–157.

28

Application-Tailored Databases for Real-Time Systems

Aleksandra Tešanović

Philips Research Laboratories

Jörgen Hansson

Carnegie Mellon University

28.1	Introduction	28-1
28.2	Dimensions of Tailorability	28-2
28.3	Tailorable Real-Time and Embedded Database Systems	28-3
	Existing Real-Time and Embedded Database Systems • Enabling Tailorability with Components and Aspect • Component-Based and Aspect-Oriented Database Systems • Challenges	
28.4	The COMET Approach	28-6
	Components and Aspects • COMET Configurations • Automated Configuration and Analysis	
28.5	Summary	28-16

28.1 Introduction

The increasing number of real-time applications, and their intrinsic need to manage larger amounts of data, present the system designer with software architectural challenges at the time of development and through the system life cycle. Real-time database research has successfully shown the merits of integrating database support for dealing with data as a resource, because a data-centric view enables simpler capturing of real-world data requirements; e.g., temporal validity, consistency, and quality, without losing semantics under a strict task-centric approach. Each application imposes requirements on the architecture with respect to the combination of memory consumption, processing capacity, hardware and operating system platform, data quality requirements, workload arrival patterns, stringency and urgency of timeliness, etc. These requirements imply that common support for dealing with real-time data efficiently needs to adhere to a unique combination of requirements as it must be tailored for the application. The incentives for developing a real-time database platform that can be specifically tailored to a range of applications are (i) economic as software is reused and development time decreases, (ii) simplified maintainability as a common database platform can be used across a family of real-time systems, and (iii) easier modeling of data requirements.

The purpose of this chapter is twofold.

- We identify the challenges of application-tailored databases regarding desirable tailoring capabilities and their impact on the database architecture.
- We present the COMET database platform using which we show how tailorability of the database software architecture can be achieved by capitalizing on the benefits of component-based software

engineering and aspect-oriented methodology. We also demonstrate the types of analysis needed to ensure that the resulting database configuration meets functional and nonfunctional requirements*.

The chapter is organized as follows. In Section 28.2, we discuss distinct dimensions of real-time database tailorability. Enabling technologies for tailorability and their impact to databases in general and the real-time domain in particular are discussed in Section 28.3. Section 28.4 gives a description of the COMET platform and elaborates how COMET can be tailored. The chapter concludes with a summary and directions for future work in Section 28.5.

28.2 Dimensions of Tailorability

A real-time database normally includes

- A physical storage of data items.
- A database schema, representing a way of structuring interrelated data items and their metadata.
- A database management system (DBMS) that manages data items. For example, a DBMS ensures that temporal validity, consistency, and quality of data, are enforced.

Hence, data items in a real-time system are stored on a physical storage according to the database schema and are managed by the DBMS.

Characteristics of a real-time system using a database strongly influence characteristics of the database and, therefore, the dimensions of database tailorability. Available storage medias in a real-time system dictate the way data are going to be organized and interrelated on the media; hence, a database schema represents a dimension of real-time database tailorability. The algorithms in the DBMS used for maintaining temporal validity, consistency, and quality of data should be chosen to satisfy the requirements of the underlying real-time system. These requirements, combined with the sparse memory and CPU resources, have a profound impact on the architecture of a DBMS, which has to be flexible and susceptible to changes and evolution. Moreover, as most embedded systems need to be able to run without human presence for long periods of time, a database in such systems should have an architecture that facilitates software updates online while maintaining a high degree of availability [27]. This yields in an essential dimension of the database tailorability, namely tailorability of the database software architecture, i.e., DBMS.

In summary, we can observe that the characteristics of the applications using the database determine desired database qualities and, therefore, produce the need for the tailoring of

- A database schema to meet the needs of the underlying hardware
- A database software architecture to meet diversity of functional and nonfunctional requirements placed on a database system by an application, e.g., optimization of CPU usage and meeting memory and timeliness requirements.

We illustrate dimensions of database tailorability in more detail on a typical example of an embedded real-time application area, vehicle control systems, e.g., engine control and driver display control [11,26]. Software controlling a vehicle typically consists of several distributed nodes called electronic control units (ECUs), each implementing a specific functionality. Although nodes depend on each other and collaborate to provide the required behavior for the overall vehicle control system, each node can be viewed as a stand-alone real-time system with specific data management characteristics. Therefore, the distinct functionality of a node and its available memory determines the uniqueness of real-time databases used for data management in the node.

The size of the nodes can vary significantly, from very small nodes to large nodes. For instance, a vehicle control system could consist of a small number of resource adequate ECUs responsible for the overall

*Nonfunctional requirements are sometimes referred to as extrafunctional requirements [7].

performance of the vehicle, e.g., a 32-bit CPU with a few megabytes of RAM, and a large number of ECUs responsible for controlling specific subsystems in the vehicle, which are significantly resource-constrained, e.g., an 8-bit micro controller and a few kilobytes of RAM [26]. In severely resource-constrained nodes, simple and customized database schemas are preferable, while larger nodes allow more elaborate database schemas to be implemented. In safety-critical nodes, tasks are often nonpreemptive and scheduled offline, avoiding concurrency by allowing only one task to be active at any given time. This, in turn, influences functionality of a database in a given node with respect to concurrency control. Less critical and larger nodes having preemptable tasks would require concurrency control mechanisms and support for database queries formulated during runtime.

One can optimize CPU usage in a node by adding the so-called on-demand algorithms that dynamically adjust the update frequencies of data based on how rapidly sensor values change in the environment. On-demand algorithms ensure that data are updated only when it is no longer fresh [11] (this is in contrast to approaches [13,18,20,47], where sensor values are always updated when data are assumed to become obsolete at the end of a constant time period, representing the worst case). As the system evolves, more elaborate schemes for data management may be applicable [25]. Namely, adding active behavior to the system in terms of event-condition-action (ECA) rules is essential to embedded applications that are time-triggered but require actions on data to be performed even when a certain event occurs, e.g., when the engine overheats appropriate updates on data items should be triggered to ensure cooling down. Each rule in the ECA paradigm reacts to certain events, evaluates a condition, and, based on the truth value of the condition, may carry out an action [2].

Additionally, the trend in the vehicular industry is to incorporate quality of service (QoS) guarantees in the new generation of vehicle control systems [25,33], implying that certain nodes are enhanced with various QoS management requirements. Typically, one is interested in controlling the performance, e.g., utilization, of a real-time database systems using the metric deadline miss ratio [22], which gives the ratio of tasks that have missed their deadlines. This can be achieved by using a feedback-based QoS management method referred to as FC-M [22], where deadline miss ratio is controlled by modifying the admitted load. Another way to change the requested utilization is to apply the policy used in an approach called QMF [15], where the quality of data in real-time databases is manipulated to achieve the desired deadline miss ratio.

We can conclude that to provide support for data management in embedded and real-time systems controlling a vehicle, it is necessary to ensure that database functionality can be tailored both on the level of DBMS functionality and on the level of database schemas. The DBMS functionality should be tailored to suit the needs of each node with respect to memory consumption, concurrency control, recovery, scheduling techniques, transaction models, QoS requirements, and storage models. Also, the architecture of a DBMS should be flexible and allow modifications and adding new functionality to the system as it evolves.

28.3 Tailorable Real-Time and Embedded Database Systems

As we have already identified, enabling tailorability of a real-time database for a specific application or a family of applications has potential. In this section, we investigate tailorability of existing real-time and embedded database systems. We then present representative software engineering techniques developed specifically to facilitate configuration and reuse of general-purpose software. Finally, we inspect how these techniques have been applied to the area of database systems, building up the knowledge base that enables us to identify concrete challenges in tailoring a database system for resource-constrained real-time applications.

28.3.1 Existing Real-Time and Embedded Database Systems

A significant amount of research in real-time databases has been done in the past years, resulting in real-time database platforms such as ART-RTDB [17], BeeHive [35], DeeDS [1], and RODAIN [21]. The research in these projects focused primarily on various schemes for concurrency control, transaction scheduling, active behavior, logging, and recovery. Hence, these databases have been developed with a

specific application in mind; their monolithic structure makes them intrinsically hard to modify or extend with new required functionality. Also it is not possible to reuse a (part of a) database in a different real-time application. Additionally, the majority of mentioned real-time platforms were developed with the assumption that there is no trade-off between the desired database functionality and available resources, i.e., that there are no constraints on available memory or CPU. In embedded environments, however, resources such as memory and CPU are sparse and a real-time database used in such an environment needs to be developed carefully to use the available resources efficiently. Trading off functionality to gain resources is one of the main concerns addressed by the embedded databases on the market, e.g., Polyhedra [31], RDM and Velocis [23], Pervasive.SQL [30], Berkeley DB [3], and TimesTen [44]. However, similar to the research platforms mentioned, all these embedded databases have different characteristics and are designed with a specific application in mind. They support different data models, e.g., relational versus object-relational model, and different operating system platforms [39]. Moreover, they have different memory requirements and provide various types of interfaces for users to access data in the database. Application developers must carefully choose the embedded database their application requires, and find the balance between required and offered database functionality.

Given the traits of existing real-time and embedded databases, system designers are faced with a difficult, time-consuming, and costly process when choosing and adapting an existing database for their applications.

28.3.2 Enabling Tailorability with Components and Aspect

The focus in software engineering research community in the past decade was on the techniques that enable development of software quickly, with low development costs and high degree of tailorability. Component-based software development (CBSD) has proved to be efficient for development of complex software systems from a predefined set of components explicitly developed for reuse [36]. CBSD postulates that software systems can be developed rapidly and tailored explicitly for a specific application or a product by choosing appropriate components from the component library and composing them into a resulting software system, i.e., a configuration or an assembly. Often, a system configuration can be viewed as one composite component, which can be (re)used in a larger system, e.g., a database system might consist of a number of components, but when used in a real-time application could be viewed as a constituting component of that application.

Components communicate with their environment through well-defined interfaces, e.g., interfaces in COM and CORBA are defined in an interface definition language (IDL), Microsoft IDL and CORBA IDL, respectively. In an interface, a collection of operations implemented in a component is normally listed together with the descriptions and the protocols of these operations [7]. Components are often defined with more than one interface. For example, a component may have three types of interfaces: provided, required, and configuration interfaces [4]. Provided and required interfaces are intended for the interaction with other components, whereas configuration interfaces are intended for use by the user of the component, i.e., a software engineer (developer) who is constructing a system using reusable components. Having well-defined interfaces ensures that an implementation part of a component can be replaced in the system without changing the interface.

Even though CBSD offers modularity, reusability, and configurability of software in terms of components, many features of software systems, e.g., synchronization, memory optimization, and temporal attributes, crosscut the overall system and thereby cannot be encapsulated in a component with a well-defined interface. Aspect-oriented software development (AOSD) [16] has emerged as a technology that provides an efficient way of modularizing crosscutting concerns in software systems. AOSD ensures encapsulation of the system's crosscutting concerns in "modules" called aspects. Aspects are written in an aspect language that corresponds to the language in which the software system is written, e.g., AspectC [6] for software written in C, AspectC++ [34] for C++ or C, and AspectJ [46] for Java-based software.*

*All existing aspect languages are conceptually very similar to AspectJ.

Aspects can be developed independently of the software and added to the system in a process called aspect weaving. Aspect weaving is done using join points of the software for which the aspect is written. A join point is a point in the code of the program where aspects should be weaved, e.g., a method, a type (struct or union). A declaration in an aspect used to specify code that should run when the join points are reached is referred as an advice. Different types of advices can be declared as follows:

- *Before advice*, which is executed before the join point
- *After advice*, which is executed immediately after the join point
- *Around advice*, which is executed instead of the join point

A special type of an advice, called *intertype advice*, or an introduction can be used for adding members to the structures or classes.

28.3.3 Component-Based and Aspect-Oriented Database Systems

SADES [32], a semiautonomous database evolution system, is an example of general-purpose object-oriented database systems, where customizations of database schema are done using AOSD. In SADES, database schema can be modified by applying aspects that (i) change links among entities, e.g., predecessor/successor links between object versions or class versions, inheritance links between classes, etc.; (ii) change the version strategy for object and class versioning; and (iii) extend the system with new metaclasses.

When it comes to tailorability in DBMS dimension, there are a number of general-purpose databases where database functionality can be extended or tailored by means of components. For example, Oracle8i [29], Informix Universal Server with its DataBlade technology [14], Sybase Adaptive Server [28], and DB2 Universal Database [5] are extendable databases that can be augmented with nonstandard functionality. Namely, each of the database is a fully functional DBMS that implements all standard DBMS functionality [8]. Nonstandard features and functionality not yet supported, e.g., abstract data types or implementations of new index structures, can be plugged into a predefined plug-ins in the DBMS architecture. OLE DB [24] is an example of a database that integrates existing data stores into a database system and provides users and applications with a uniform view of the entire system. In this model, the architecture introduces a common intermediate format into which local data formats can be translated. Components, also known as wrappers, are located between the DBMS and each data source. Each component mediates between the data source and the DBMS, i.e., it wraps the data source. KIDS [9] is an example of databases that enable composition of nonstandard DBMSs out of reusable components.

While the mentioned database systems are tailorable they do not provide support for enforcing real-time behavior nor do they scale down to a size and functionality acceptable in resource-constrained systems.

28.3.4 Challenges

When developing tailorable databases for resource-constrained real-time systems, a number of challenges arises. First, based on the application domain the focal dimensions of tailorability should be determined. We found that in most real-time applications it is more important to ensure flexibility in the real-time database architecture and provide variability of data management algorithms that optimize resource usage, than enabling tailoring of database schema.

Further, to achieve fast and cost-effective development of database systems for specific real-time application or families of applications, one should apply appropriate software development technology. Given that CBSD and AOSD have been developed to address complementary concerns in software design and development, it follows intuitively that combining the two in the development of software systems could further improve configurability of software while keeping the costs of development and time to produce the software low. We have explored the combination of the two techniques in real-time system development within the ACCORD* framework [43] and demonstrated the benefits of combining the techniques

*ACCORD stands for aspectual component-based real-time system development [43].

in this domain. In the next sections, we demonstrate the benefits of the combination of two approaches in the development of a tailorable embedded real-time database platform, COMET.

Note that, if a real-time system is developed using reusable artifacts, then ensuring that the database configuration satisfies functional and nonfunctional requirements is essential. Meeting nonfunctional requirements in terms of available memory footprint and timeliness, e.g., determining allowed worst-case execution times (WCETs) of transactions such that deadlines can be met, is especially interesting for the real-time and embedded domain. Therefore, together with technology for tailoring the database, providing accompanied analysis methods and tools is also necessary.

28.4 The COMET Approach

COMET is a real-time embedded database platform that can be tailored to fit the needs of various real-time and embedded applications. In the remainder of this section, we present the relevant details of the RTCOM model and aspect packages as these are the main facilitators of COMET tailorability. We then elaborate how COMET can be tailored for various applications by adding new features via aspect packages.

28.4.1 Components and Aspects

Tailorability in COMET refers primarily to tailorability of its software architecture, which is achieved by defining two basic building and reusable units of the architecture, components, and aspect packages.

28.4.1.1 Components

They implement a part of a well-defined database functionality and exhibit strong internal cohesion but loose coupling with other parts of database functionality. Components are designed and implemented using a real-time component model (RTCOM) [43] and can be tailored by aspect weaving.

RTCOM defines how a component in a real-time environment should be designed and implemented such that tailoring of components for specific needs of an application can be done while preserving information, hiding and enabling component reuse, and analysis of system's temporal behavior, e.g., WCET analysis or formal verification using timed automata [38,40]. RTCOM components are “gray” as they are encapsulated in interfaces, but changes to their behavior can be performed in a predictable way using aspects. Hence, components provide certain initial functionality that can be modified or changed by weaving of aspects. Each RTCOM component has two types of functional interfaces: provided and required (see Figure 28.1). Provided interfaces reflect a set of operations that a component provides to other components, while required interfaces reflect a set of operations that the component requires (uses) from other components. The composition interface declares join points where changes can be made in the component functionality and thereby defines places in the component where modifications can be done. Join points need to be explicitly declared in a separate interface to ensure evolution of the system. Namely,

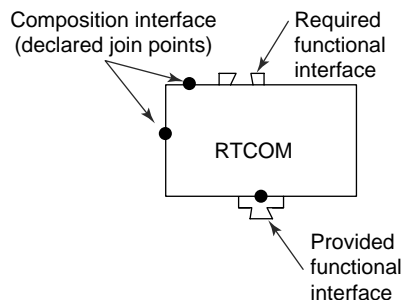


FIGURE 28.1 RTCOM in a nutshell.

when developing new aspects, the aspect developer does not necessarily require to have full knowledge of the component code to develop aspects which would tailor that component quickly and successfully.

A composition of COMET components providing the basic functionality is denoted as basic COMET configuration (or simply as basic COMET). We refer to a minimal subset of database functionality that can constitute a working database as the basic database functionality. The minimal subset of functionality typically implies having an interface for applications to access the database and functionality for defining and executing transactions that read and update data on a physical storage. Hence, we developed the following COMET components constituting a basic COMET:

- User interface component (UIC)
- Scheduling manager component (SMC)
- Indexing manager component (IMC)
- Transaction manager component (TMC)
- Memory manager component (MMC)

The UIC provides a database interface to the application, which enables a user (i.e., an application using the database) to query and manipulate data elements. User requests are parsed by the user interface and are then converted into an execution plan. The TMC is responsible for executing incoming execution plans, thereby performing the actual manipulation of data. The IMC is responsible for maintaining an index of all tuples in the database. The SMC keeps records of transactions that exist in the system. Note that in the basic configuration of COMET we have a basic data model and a basic transaction model. The basic data model contains metadata used for concurrency control algorithms in databases. The basic transaction model characterizes each transaction τ_i only with a period p_i and a relative deadline d_i . The basic COMET is especially suitable for small embedded vehicular systems [43].

28.4.1.2 Aspect Packages

Aspect packages implement a new, nonbasic, functionality that can be added to the existing database functionality and thereby create a new variant of the database system. All nonbasic functionalities are considered to be features of the database. The features in the real-time database domain typically are real-time policies facilitating concurrent access to the database, various indexing strategies, enabling active behavior, providing QoS guarantees, etc. The creation of database configurations with new feature(s) is done by adding the appropriate aspect package(s) to the basic COMET. Hence, an aspect package represents a way of packaging the specification and implementation of real-time features, e.g., concurrency control, QoS, indexing, and scheduling, for reuse and configuration.

At an abstract level, an aspect package represents a way of specifying a real-time database feature, where a database feature is specified independently of an application by means of aspects and components. At a concrete level, an aspect package consists of a set of aspects and (possibly empty) set of components implementing a variety of real-time policies. Components from the aspect package contain the main functional infrastructure of the features (policies), e.g., QoS management algorithms. In addition, there are three types of aspects defined in an aspect package as follows:

- Policy aspects adapt the system to provide a variety of related real-time policies, e.g., feedback-based QoS policies or concurrency control policies.
- Transaction model aspects adapt the transaction model of a real-time database system to the model used by the policies by adding various attributes to transactions, e.g., a utilization transaction model needs to be used for a feedback-based QoS policy where utilization of the system is controlled.
- Connector aspects facilitate composition of an existing system with the components from the aspect package.

Using an aspect package, a system designer can develop several applications with similar characteristics with respect to a certain real-time policy as illustrated in Figure 28.2. Therefore, each family of applications would have its unique aspect package.

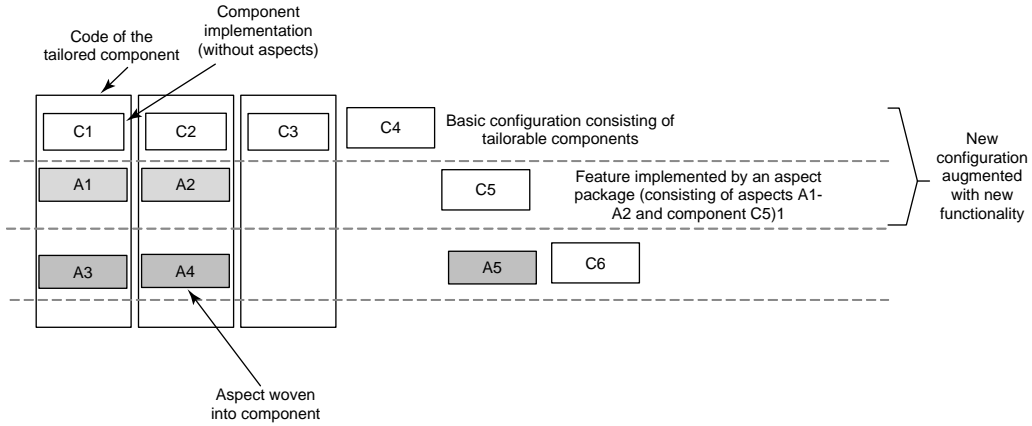


FIGURE 28.2 Creating a family of databases using aspect packages.

We developed the following COMET aspect packages:

- The concurrency control aspect package represents an implementation of a set of concurrency control policies that enable producing a family of databases with distinct policies on handling data conflicts. The concurrency control aspect package consists of one component, the locking manager component (LMC), and aspects implementing high-priority two-phase locking (HP-2PL) and optimistic divergence control (ODC) protocols.
- The index aspect package represents an implementation of indexing policies that facilitate efficient data access. The index aspect package consists of one component, an alternative IMC that implements B-tree structure (IMC_B-tree), and the aspect implementing GUARD indexing policy.
- The active behavior package enables creating database system configurations capable of reacting on aperiodic events. It consists of aspects implementing on-demand algorithms and ECA rules.
- The QoS aspect package gives a set of components and aspects that implement a variety of feedback-based QoS policies. The QoS aspect package consists of two components, the QoS admission controller component (QAC) and the feedback controller component (FCC), and the aspects implementing a number of feedback control-based QoS policies.

Note that, even though the content of an aspect package varies depending on the functionality provided within the package, each package typically has one or a number of aspects that implement a specific set of (related) policies. Additionally, if a policy requires changes in the transaction model of COMET, the package contains appropriate aspects augmenting this model. To illustrate how the transaction and data model need to be adapted to the policy used, consider the FC-M and QMF QoS algorithms. These algorithms require more complex data and transaction models that can capture additional metadata such as mean interarrival and execution times. Both FC-M and QMF require a transaction model where transaction τ_i is classified as either an update or a user transaction. Update transactions arrive periodically and may only write to base (temporal) data objects. User transactions arrive aperiodically and may read temporal and read/write nontemporal data. In this model, denoted the utilization transaction model, each transaction has the following characteristics:

- Period p_i (update transactions)
- Estimated mean interarrival time $r_{E,i}$ (user transactions)
- Actual mean interarrival time $r_{A,i}$ (user transactions)
- Estimated execution time $x_{E,i}$
- Actual execution time $x_{A,i}$
- Relative deadline d_i

TABLE 28.1 The Utilization Transaction Model

Attribute	Periodic Transactions	Aperiodic Transactions
d_i	$d_i = p_i$	$d_i = r_{A,i}$
$u_{E,i}$	$u_{E,i} = x_{E,i} / p_i$	$u_{E,i} = x_{E,i} / r_{E,i}$
$u_{A,i}$	$u_{A,i} = x_{A,i} / p_i$	$u_{A,i} = x_{A,i} / r_{A,i}$

- Estimated utilization* $u_{E,i}$
- Actual utilization $u_{A,i}$

Table 28.1 presents the complete utilization transaction model. Upon arrival, a transaction presents the estimated average utilization $u_{E,i}$ and the relative deadline d_i to the system. The actual utilization of the transaction $u_{A,i}$ is not known in advance due to variations in the execution time.

28.4.2 COMET Configurations

A significant number of COMET configurations fulfilling distinct requirements can be made from existing COMET components and aspects. In this section, we describe a subset of possible COMET configurations, their relationship, and constituents.

The basic configuration consists of five COMET components (see Table 28.2): UIC, SMC, TMC, IMC, and MMC. As mentioned, the configuration uses the basic transaction model, and it is especially suitable for small resource-constrained embedded vehicular systems [25]. All remaining database configurations are built upon the basic COMET configuration by adding appropriate aspects and components from available aspect packages.

The high-priority two-phase locking concurrency control configuration (COMET HP-2PL) includes all basic components, as well as the LMC and the HP-2PL aspects from the concurrency control aspect package. This configuration is suitable for more complex real-time systems where transactions can execute concurrently and conflict resolution can be done by employing pessimistic concurrency control.

The optimistic divergence control configuration (COMET ODC) is built on top of the basic COMET configuration by adding the following constituents of the concurrency control aspect package: the LMC, the ODC aspect, and the epsilon-based transaction model aspect, as indicated in Table 28.2. This configuration, therefore, enables database system to resolve conflicts based on the ODC policy. ODC [45] is based on an optimistic concurrency control method introduced by Yu and Dias [48], which uses weak and strong locks and assumes transaction execution in three phases: read, write, and validation phase. As such, COMET ODC is suitable for soft real-time applications having concurrent transactions where conflict resolution can be postponed until the validation phase of the transaction.

The GUARD-link configuration (COMET GUARD) requires all components from the basic COMET configuration, except the IMC. It also requires the constituents of the concurrency control aspect package. Furthermore, from the index aspect package the IMC_B-tree and the GUARD link policy aspect are required. Haritsa and Seshadri [12] proposed the GUARD-link indexing protocol to further strengthen the index management of a soft real-time system by taking into account transaction deadlines, which we adopted in COMET as a part of the index aspect package. Two distinct COMET GUARD configurations can be made depending on the choice of the concurrency control policy from the concurrency control aspect package. Namely, if the HP-2PL with similarity aspect is used, COMET GUARD with HP-2PL is created. If ODC is used, the configuration of COMET GUARD with ODC is made (see Table 28.2). COMET GUARD configurations are desirable in soft real-time applications where indexing structure should be further optimized in terms of meeting transaction deadlines.

*Utilization is also referred to as load.

TABLE 28.2 Relationship between Different Parts of the Concurrency Control and the Index Aspect Package and Various COMET Configurations

		COMET Configurations				
		Basic COMET	Concurrent COMET		COMET GUARD	
			COMET HP-2PL	COMET ODC	with HP-2PL	with ODC
Basic COMET components	UIC	X	X	X	X	X
	TMC	X	X	X	X	X
	SMC	X	X	X	X	X
	IMC	X	X	X		
	MMC	X	X	X	X	X
Concurrency Control Aspect Package						
Policy aspects	HP-2PL		X		X	
	ODC			X		X
Transaction model aspects	Epsilon-based			X		X
Connector aspects						
Components	LMC		X	X	X	X
Index Aspect Package						
Policy aspects	GUARD link				X	X
Transaction model aspects						
Connector aspects						
Components	IMC_B-tree				X	X

Note: X in the table means that an aspect/component is part of a configuration. UIC: User Interface Component; IMC: index manager component; TMC: transaction manager component; MMC: memory manager component; SMC: scheduling manager component; LMC: locking manager component; COMET HP-2PL: The high-priority two-phase locking configuration; COMET ODC: The optimistic divergence control configuration.

A number of COMET QoS configurations can be made using the COMET QoS aspect package together with different combinations of aspects and components from the concurrency control aspect package and the index aspect package. For simplicity, here we discuss only COMET QoS configurations that are distinct with respect to QoS-related aspect and components. Any of the concurrent or GUARD COMET configurations from Table 28.2 can be used as a foundation for adding aspects and components from the COMET QoS aspect package and creating a COMET QoS configuration. Hence, we discuss in detail five distinct COMET QoS configurations that provide admission control, FC-M, QMF, self-tuning, and least squares regression QoS. However, note that depending on the chosen aspects and components from the COMET QoS aspect package, the number of possible configurations in COMET QoS family is higher (see Figure 28.3). Table 28.3 illustrates the elements of the QoS aspect package used in the five representative COMET QoS configurations.

The admission control QoS configuration includes one component from the QoS aspect package, the QAC. The configuration also requires two aspects, the QAC connector aspect and the utilization transaction model aspect. The QAC connector aspect adds the QAC to the existing controlled system, while the utilization transaction model aspect extends the transaction model (see Table 28.3). The admission control configuration is simple as it only provides facilities for admission control.

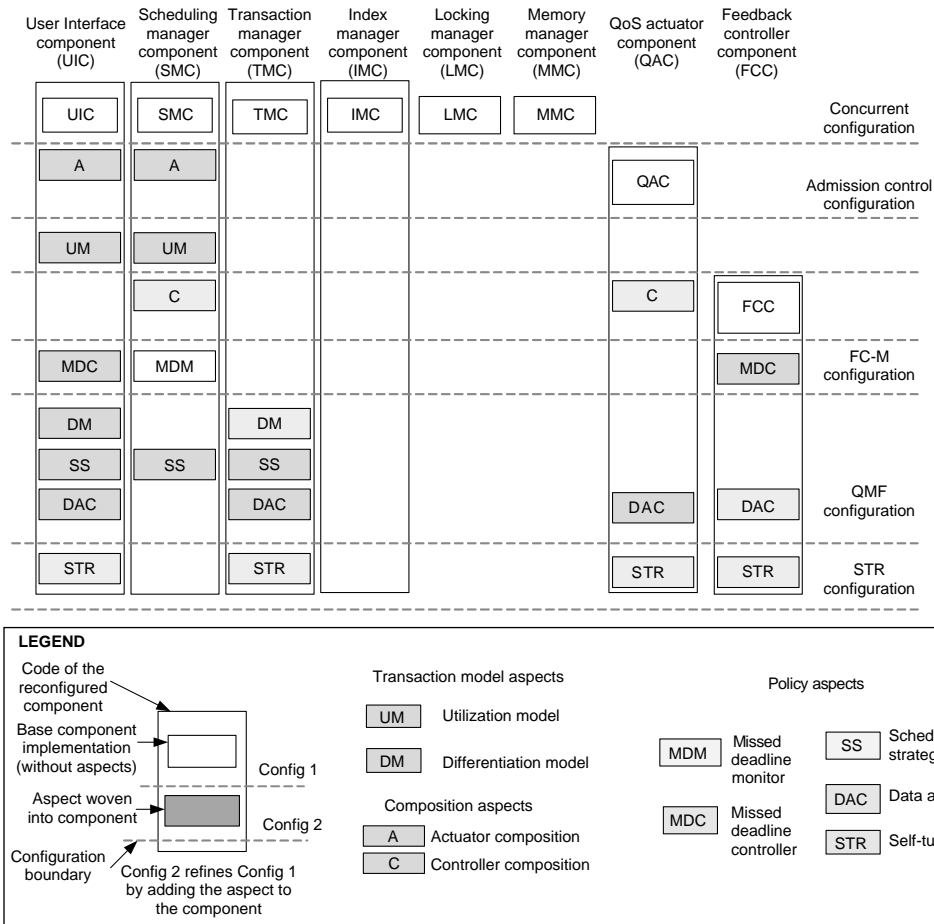


FIGURE 28.3 Creating a family of real-time systems from the COMET QoS aspect package.

The *miss-ratio feedback configuration* (COMET FC-M) provides QoS guarantees based on the FC-M policy. The configuration includes the QAC and FCC components and their corresponding connector aspects, the utilization transaction model aspect, the missed deadline monitor aspect, and the missed deadline controller aspect (see Table 28.3). These aspects modify the policy of the SMC and FCC to ensure that QoS with respect to controlling the number of deadline misses is satisfied in soft-real-time systems where workload characteristics are generally unpredictable.

The *update scheduling configuration* (COMET QMF) provides the QoS guarantees based on the QMF policy. Here the data differentiation aspect and scheduling strategy aspect are used to enrich the transaction model. Moreover, the data access monitor aspect is required to ensure the metric used in QMF. Also, the QoS through update scheduling aspect is used to further adjust the policy of QAC to suit the QMF algorithm. COMET QMF is especially suitable for soft-real-time applications where maintaining a certain level of data quality is of interest.

The *self-tuning regulator configuration* (COMET STR) provides adaptive QoS control where the control algorithm parameters are adjusted by using the self-tuning regulator. This aspect is added to the aspects and component constituting the COMET FC-M configuration to ensure the adaptability of the control already provided by the COMET FC-M configuration. The COMET STR configuration can be used for real-time systems that cannot be modeled as time-invariant.

TABLE 28.3 Relationship between Different Parts of the QoS Package and Various COMET QoS Configurations

		COMET Configurations				
		Admission control	COMET FC-M	COMET QMF	COMET STR	COMET RM
Policy aspects	Actuator utilization policy	X	X	X	X	X
	Missed deadline monitor		X	X	X	X
	Missed deadline controller		X	X	X	X
	Scheduling strategy			X		
	Data access monitor			X		
	QoS through update scheduling			X		
	Self-tuning regulator				X	
	Adaptive regression model					X
Transaction model aspects	Utilization transaction model	X	X	X	X	X
	Data differentiation			X		
Connector aspects	Actuator connector	X	X	X	X	X
	Controller connector		X	X	X	X
Components	QAC	X	X	X	X	X
	FCC		X	X	X	X

Note: X in the table means that an aspect (or a component) is part of a configuration. QAC: QoS actuator component; FCC: Feedback controller component; COMET FC-M: The miss-ratio feedback configuration; COMET STR: The self-tuning regulator configuration; COMET QMF: The update scheduling configuration; COMET RM: The regression model configuration.

The regression model configuration (COMET RM) provides adaptive QoS control where the control algorithm parameters are adjusted by using the least square technique and the regression model. This aspect also requires all the aspects needed for the FC-M configuration to ensure adaptability of QoS management.

Note that the process of adding aspect packages to the basic COMET configuration is incremental. That is, when making QoS COMET configuration, first the concurrency control aspect package should be applied on the basic configuration, and then appropriate index package (optional), and only then aspects and component from QoS aspect package can be applied to configure COMET QoS configuration.

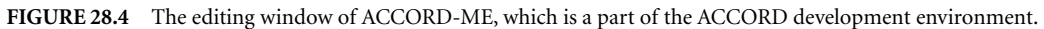
28.4.3 Automated Configuration and Analysis

We developed a number of analysis techniques that can be used to check if the obtained COMET configuration is suitable for the underlying real-time environment.

We embodied these techniques into ACCORD modeling environment (ACCORD-ME) tool suite, which supports both configuration and analysis. ACCORD-ME is a constituting part of the tool-set called ACCORD tool development environment. The overall ACCORD tool environment is developed to provide support for development, analysis, and deployment of reconfigurable real-time systems [37].*

ACCORD-ME is a model-based tool, implemented using the generic modeling environment (GME), a toolkit for creating domain-specific modeling environments [10]. In general, the creation of a GME-based tool is accomplished by defining metamodels that specify the modeling paradigm (modeling language) of the application domain. In our case, based on the content of the aspect packages, modeling paradigms are defined for ACCORD-ME. The aspect package modeling paradigms are given as UML diagrams and they define the relationship between components and aspects in the package, their possible relationship to other components of COMET, and possibilities of combining components and aspects into different configurations.

*The name ACCORD-ME originates from the name of the approach for ACCORD [42] for which this tool was initially developed.



An Example

- R1:** The application performs computations using data obtained from sensors and forwards the computation results directly to actuators; recall that sensor data items are typically referred to as base data items.
- R2:** Sensor data should reflect the state of the controlled environment implying that transactions used for updating data items should be associated with real-time properties, such as periods and deadlines.
- R3:** Values of data should be updated only if they are stale* to save computation time of the CPU.
- R4:** Multiple tasks can execute concurrently in an ECU, and operate on the same data. Hence, consistency needs to be enforced.

*A data item is stale if its value does not reflect the current state of the environment.

R5: The tasks in the ECU should be scheduled according to priorities.

R6: The memory footprint of the system should be within the *memBound*, which can be obtained from the ECU specifications.

When configuring COMET ECU we start by specifying the requirements using the configurator tool in ACCORD-ME (see Figure 28.4). The configurator provides three levels of support, expert, configuration, and requirement-based, based on the expertise and preferences of the developer. The expert option is used by developers familiar with the content of the aspect packages and all of its provided functionality and policies. The configuration-based option gives a list of possible configurations of the system that can be created from the aspect packages. The requirement-based option provides the system developer with a list of possible requirements from which the developer can choose a relevant subset suitable for a particular application. Thus, developers do not need to know the contents of the aspect packages to be able to configure QoS management.

Given that we know the system requirements, then the requirement-based option in the configurator can be used to guide the system composition. Now, based on the requirements R1–R5 we can choose options in the requirement-based form (shown in Figure 28.5) as follows. The configuration of the COMET database suitable for the ECU contains base data items. Since tasks using the base data are not storing intermediate results, there is no need for storing derived data (R1). Furthermore, the database should provide mechanisms for dealing with concurrency such that conflicts on data items are resolved (R4) and

Requirement-based Configurations

Data Model : ☐ Base data item ☐ Base and derived data item

Data Access Control : ☐ None ☒ HP-2PL with similarity ☐ ODC

Index access control: ☐ None ☒ Simple (mutex-based) ☐ High performance Guard-Link

Transaction model: ☐ Based on transaction ID

☒ Option1: Based on transaction ID, period and deadline

☐ Based on transaction ID, period and deadline(epsilon transactions)

☐ Option2: Option1 + utilization and execution time

☐ Option3: Option2 + immediate and on-demand update transaction

Scheduling policy ☐ None ☐ Earliest Deadline First (EDF) ☒ Rate Monotonic Scheduling (RMS)

QoS policy: ☒ None

☐ Utilization-based admission test

☐ Feedback-based control of deadline miss ratio

☐ Feedback-based control of deadline miss ratio through update scheduling

Submit

FIGURE 28.5 Requirement-based configuration of the real-time database system.

data items that are stale are updated (R3). This can be achieved using HP-2PL with similarity [19]. The transaction model should be chosen such that transactions are associated with periods and deadlines (R2). We choose the RMS scheduling policy to enforce priority-based scheduling of tasks (R5), performance guarantees in terms of levels of QoS are not required.

When these decisions are submitted, the configurator loads the relevant components and aspects into the ACCORD-ME editing area, where the designer can assemble the system by connecting the components and aspects, as shown in Figure 28.4. Each aspect and component contains information stating with which components it can be connected to form a functionally correct COMET configuration. Hence, the configurator aids in obtaining a configuration of a system that satisfies specific needs of the application and that is functionally correctly assembled. The configuration, i.e., components, aspects, and their relationships, is stored in an XML file. A system configuration satisfying functional requirements R1–R5 is shown in the upper part of Figure 28.6 (denoted as step 1 in the figure). Note that in ACCORD-ME, ovals are used as the graphical representation of aspects, while squares represent components.

The obtained configuration can be analyzed by other subtools available in ACCORD-ME, namely M&W analyzer and formal verifier. Specifically, formal analysis can be performed to ensure that adding constituents of the aspect packages preserves already proven properties of the original system [38], thereby ensuring that functionality of the original system is preserved or modified in a well-understood way. For the purposes of formal analysis, a COMET configuration is transformed from a UML-based model to a timed automata-based model.

When the composition of the system is made, it should be analyzed to determine the memory needs of the configuration and contrast these to the available memory in the ECU. Hence, when the configuration part of the system development is finished then the obtained configuration can be analyzed using the M&W analyzer tool plug-in that analyzes a configuration with respect to WCET and memory requirements using the aspect-level WCET analysis we developed earlier [40,41]. It takes as input an XML configuration file

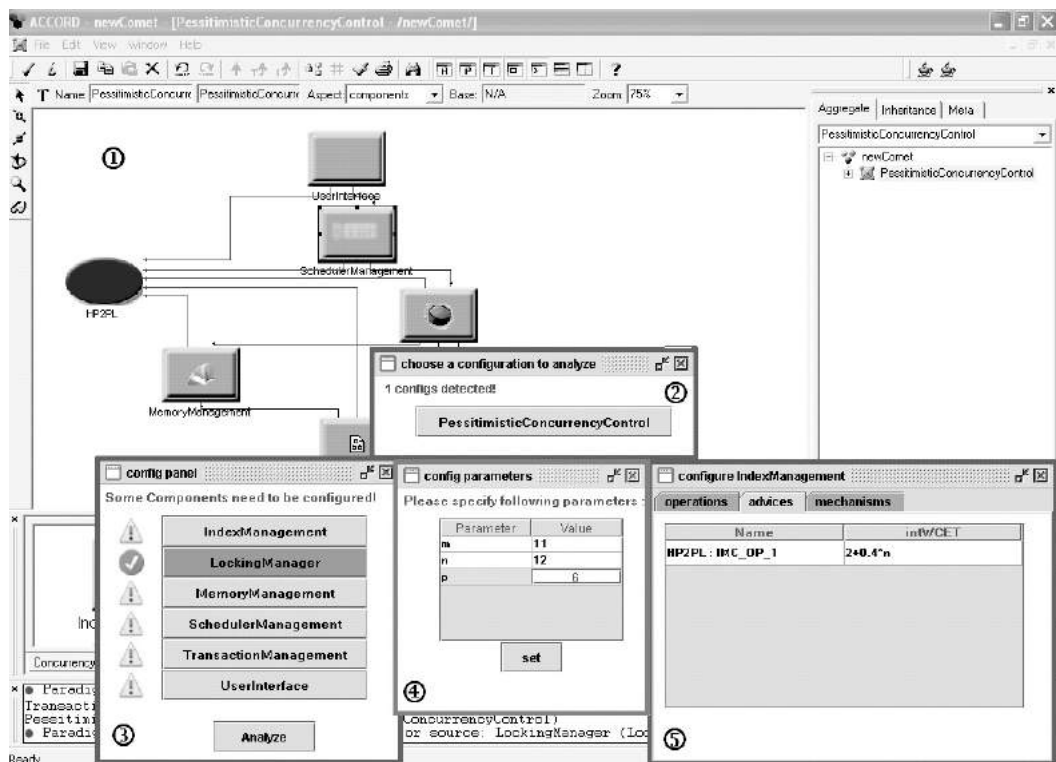


FIGURE 28.6 The snapshot of ACCORD-ME when doing analysis on a real-time system configuration.

produced by the configurator and runtime properties of components and aspects. Runtime properties are also expressed in XML and contain the runtime information needed by the tool to calculate the impact of aspect weaving and system composition with respect to WCET or static memory consumption.

WCET's needs are expressed in terms of symbolic expressions. Hence, they are a function of one or several parameters that abstract the properties of the underlying runtime environment. The symbolic expressions need to be reevaluated for each runtime environment, i.e., parameters in symbolic expressions should be instantiated in the analysis. Hence, the M&W analyzer provides a list of symbolic expressions and the list of necessary parameters that need to be configured. Since it is possible to assemble a number of configurations in ACCORD-ME, e.g., when using the expert option, the M&W analyzer detects the configurations and enables developers to choose which configuration she/he would like to analyze. Moreover, it is possible to choose whether WCET or memory analysis of the system should be performed.

Figure 28.6 is the snapshot of the analysis process. When the M&W analyzer is invoked, it detects the configuration(s) one might have made in ACCORD-ME and prompts for the choice of a configuration. In our example, we created only one configuration and denoted it `PessimisticConcurrencyControl`. This configuration is detected by the M&W analyzer, as illustrated by step 2 in Figure 28.6. After the configuration is chosen, the appropriate files describing runtime aspects of components and aspects are loaded for analysis. Since runtime properties of aspects and components are described in terms of symbolic expressions with parameters, the values of these parameters are instantiated during analysis, and the list of components that require instantiation of parameters is displayed during analysis (step 3 in the figure). One can also make an inspection of the symbolic expressions and input the values of parameters in the expressions, as depicted by step 4. Note that advices that modify components are included in the component runtime description as shown in step 5. Once the values of parameters are set for this particular ECU, the tool outputs the resulting WCET and/or memory consumption values which can be compared with the values given in the memory requirement (R6).

If the obtained configuration satisfies the requirements of the target ECU, the next step is to compile the system and deploy it into the runtime environment, which is done using the configuration compiler. As mentioned previously, the configuration compiler also provides documentation of the composed configuration of COMET.

28.5 Summary

There is a clear need for enabling development of real-time database systems that can be tailored to meet specific application needs. In this chapter, we have analyzed real-time application characteristics and identified those that have the greatest impact on the database functional and nonfunctional requirements. We have shown that available hardware dictates the organization of database schema yielding the need for tailorability of the database. Furthermore, software architecture of the real-time and embedded databases should be tailorable to facilitate cost-effective and fast development of families of database systems.

We have illustrated how tailorability of the database software architecture can be achieved on the example of the COMET database system. In COMET, the flexibility and extendability of the architecture are achieved by the means of components and aspects, and depending on the application requirements, the designer takes appropriate components and aspect from the COMET library and forms the COMET configuration. Over 30 database configurations with distinct characteristics can be developed using artifacts from the COMET library. The designer is aided in the tailoring and analysis process by appropriate tools.

Based on our experience, we believe that the proposed technologies for facilitating tailorability into the real-time domain enable us to quickly develop functionally correct application-specific database systems that meet nonfunctional constraints, e.g., small memory footprint, adequate WCETs that ensure meeting deadlines, and improved CPU performance.

Acknowledgments

The authors would like to thank Jack Goossen and Rob van Ommering for the initial reviewing of the manuscript. This work was financially supported by the Swedish Foundation for Strategic Research (SSF) via the SAVE project and the Center for Industrial Information Technology (CENIIT) under contract 01.07.

References

1. S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and N. Efring. DeeDS towards a distributed and active real-time database system. *ACM SIGMOD Record*, 25(1), 1996.
2. P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems: Concepts, Languages and Architectures*. McGraw-Hill, New York, 1999. Chapter Active Databases.
3. D. B. Berkeley. Sleepycat Software Inc., <http://www.sleepycat.com>, March 2003.
4. J. Bosch. *Design and Use of Software Architectures*. ACM Press in collaboration with Addison-Wesley, New York, 2000.
5. M. J. Carey, L. M. Haas, J. Kleewein, and B. Reinwald. Data access interoperability in the IBM database family. *IEEE Quarterly Bulletin on Data Engineering: Special Issue on Interoperability*, 21(3): 4–11, 1998.
6. Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2002.
7. I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Real-Time Systems*. Artech House Publishers, Boston, July 2002.
8. K. R. Dittrich and A. Geppert. Component database systems: Introduction, foundations, and overview. In *Component Database Systems*. Morgan Kaufmann Publishers, 2000.
9. A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich, September 1997.
10. The generic modeling environment. Institute for Software Integrated Systems, Vanderbilt University, USA. <http://www.isis.vanderbilt.edu/Projects/gme/>, December 2004.
11. T. Gustafsson and J. Hansson. Data management in real-time systems: A case of on-demand updates in vehicle control systems. In *Proceedings of 10th IEEE Real-Time Applications Symposium (RTAS'04)*, pp. 182–191. IEEE Computer Society Press, 2004.
12. J. R. Haritsa and S. Seshadri. Real-time index concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 12(3): 429–447, 2000.
13. S.-J. Ho, T.-W. Kuo, and A. K. Mok. Similarity-based load adjustment for real-time data-intensive applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pp. 144–154. IEEE Computer Society Press, 1997.
14. Developing DataBlade modules for Informix-Universal Server. Informix DataBlade Technology. Informix Corporation, 22 March 2001. <http://www.informix.com/datablades/>.
15. K.-D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-sensitive approach for timeliness and freshness guarantees in real-time databases. In *Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems (ECRTS'02)*, pp. 203–212. IEEE Computer Society Press, 2002.
16. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer, Berlin, 1997.

17. Y. Kim, M. Lehr, D. George, and S. Son. A database server for distributed real-time systems: Issues and experiences. In *Proceedings of the 2nd Workshop on Parallel and Distributed Real-Time Systems*, Cancun, Mexico, April 1994.
18. Y.-K. Kim and S. H. Son. Supporting predictability in real-time database systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pp. 38–48. IEEE Computer Society Press, 1996.
19. K. Y. Lam and W. C. Yau. On using similarity for concurrency control in real-time database systems. *The Journal of Systems and Software*, 43(3): 223–232, 1998.
20. C.-G. Lee, Y.-K. Kim, S. H. Son, S. L. Min, and C. S. Kim. Efficiently supporting hard/soft deadline transactions in real-time database systems. In *Proceedings of the 3rd International Workshop on Real-Time Computing Systems and Applications*, pp. 74–80, 1996.
21. J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Proceedings of the International Workshop on Databases in Telecommunications*, volume 1819 of *Lecture Notes in Computer Science*, pp. 158–173. Springer, Berlin, 1999.
22. C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Journal of Real-Time Systems*, 23(1/2), 2002.
23. MBrane Ltd. RDM Database Manager. <http://www.mbrane.com>.
24. Universal data access through OLE DB. OLE DB Technical Materials. OLE DB White Papers, 12 April 2001. <http://www.microsoft.com/data/techmat.htm>.
25. D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A component-based real-time database for automotive systems. In *Proceedings of the IEEE Workshop on Software Engineering for Automotive Systems*, pp. 1–8, May 2004.
26. D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N.-E. Bänkestad. Data management issues in vehicle control systems: A case study. In *Proceedings of the 14th IEEE Euromicro International Conference on Real-Time Systems (ECRTS'02)*, pp. 249–256. IEEE Computer Society Press, 2002.
27. M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9): 27–34, 2000.
28. S. Olson, R. Pledereder, P. Shaw, and D. Yach. The Sybase architecture for extensible data management. *Data Engineering Bulletin*, 21(3): 12–24, 1998.
29. All your data: The Oracle extensibility architecture. Oracle Technical White Paper. Oracle Corporation. Redwood Shores, CA, February 1999.
30. Pervasive Software Inc. Pervasive.SQL Database Manager. <http://www.pervasive.com>.
31. Polyhedra Plc. Polyhedra database manager. <http://www.polyhedra.com>.
32. A. Rashid. A hybrid approach to separation of concerns: the story of SADES. In *Proceedings of the 3rd International REFLECTION Conference*, volume 2192 of *Lecture Notes in Computer Science*, pp. 231–249, Kyoto, Japan, September 2001. Springer, Berlin.
33. M. Sanfridson. Problem formulations for qos management in automatic control. Technical Report TRITA-MMK 2000:3, ISSN 1400-1179, ISRN KTH/MMK-00/3-SE, Mechatronics Lab KTH, Royal Institute of Technology (KTH), Sweden, March 2000.
34. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'02)*, Sydney, Australia, February 2002. Australian Computer Society.
35. J. Stankovic, S. Son, and J. Liebeherr. BeeHive: Global multimedia database support for dependable, real-time applications. In A. Bestaros and F. Fay-Wolfe (eds), *Real-Time Databases and Information Systems*, pp. 409–422. Kluwer Academic Publishers, Boston, MA, 1997.
36. C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
37. A. Tešanović, P. Mu, and J. Hansson. Development environment for configuration and analysis of embedded real-time systems. In *Proceedings of the 4th International Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'05)*, March 2005.

38. A. Tešanović, S. Nadjm-Tehrani, and J. Hansson. *Component-Based Software Development for Embedded Systems — An Overview on Current Research Trends*, volume 3778 of *Lecture Notes in Computer Science*, chapter Modular Verification of Reconfigurable Components, pp. 59–81. Springer, Berlin, 2005.
39. A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Embedded databases for embedded real-time systems: A component-based approach. Technical Report, Department of Computer Science, Linköping University, and Department of Computer Engineering, Mälardalen University, 2002.
40. A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Integrating symbolic worst-case execution time analysis into aspect-oriented software development. OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.
41. A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components. In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*, Poland, May 2003. Elsevier.
42. A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), 2004.
43. A. Tešanović. *Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, March 2006.
44. TimesTen Performance Software. TimesTen DB. <http://www.timesten.com>.
45. K. L. Wu, P. S. Yu, and C. Pu. Divergence control algorithms for epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 9(2): 262–274, 1997.
46. Xerox Corporation. *The AspectJ Programming Guide*, September 2002. <http://aspectj.org/doc/dist/progguide/index.html>.
47. M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pp. 32–43. IEEE Computer Society Press, 1999.
48. P. S. Yu and D. M. Dias. Impact of large memory on the performance of optimistic concurrency control schemes. In *Proceedings of the International Conference on Databases, Parallel Architectures, and Their Applications (PARBASE'90)*, pp. 86–90. IEEE Computer Society Press, 1990.

29

DeeDS NG: Architecture, Design, and Sample Application Scenario

Sten F. Andler
University of Skövde

Marcus Brohede
University of Skövde

Sanny Gustavsson
University of Skövde

Gunnar Mathiason
University of Skövde

29.1	Introduction	29-1
	DeeDS Design Philosophy and Capabilities • Wildfire Scenario • Legacy DeeDS Architecture	
29.2	Active Research Problems	29-4
	Optimistic Replication • Scalability • Distributed Real-Time Simulation	
29.3	DeeDS NG	29-5
	Database Model and Eventual Consistency • Optimistic Replication • Scalability and Virtual Full Replication • Simulation DeeDS: A Way to Implement Real-Time Simulations	
29.4	Related Work	29-16
29.5	Summary	29-17
	Conclusions • Future Work	

29.1 Introduction

A database system (DBS) consists of a Database management system for defining and manipulating data and a database for storing the actual data. A distributed real-time DBS (DRTDBS) is a distributed DBS that supports real-time constraints on transactions.

In this chapter, we discuss some requirements of emergent embedded applications in security and emergency management. These applications often require resource predictability, high availability, data consistency, and support for recovery. We propose that these requirements can be met by using a DRTDBS as infrastructure, or by including features from a DRTDBS in support for certain applications. Further, we claim that the DeeDS NG (Next Generation) architecture and prototype is a suitable DRTDBS for meeting the requirements, and describe how we address some of the critical features needed in the DeeDS NG design. We build on experience from the original DeeDS architecture and prototype, and describe current research undertaken in the new DeeDS NG.

29.1.1 DeeDS Design Philosophy and Capabilities

The original DeeDS architecture was based on the assumption that time constraints are strict and very important in each node where a critical transaction is executing, but that replication of data to other nodes, which may use the same information is less strict. The architecture of DeeDS has been guided by the

following design decisions: main memory residence, full replication, and detached replication. The resulting architecture allows predictable access and update as well as autonomous execution during partitioning of the system. Issues that were focused on earlier include mixed workload deadline scheduling and overload resolution, predictable event-condition-action (ECA) rule management, and resource-predictable event monitoring [4]. In the first phase of the project, the requirements were established and the elements of the architecture elaborated; in the second phase, an implementation of a prototype was undertaken based on public-domain software. In the third phase of the project, experience was gained in using the research prototype to support distributed simulation of helicopters and vehicles in the WITAS Autonomous UAV project [13].

Emergent applications need strong support in certain functions, such as data consistency, system scalability, and fault tolerance, which has motivated further work in DeeDS NG to support certain key features: eventual consistency and application tolerance of temporary inconsistencies [20]; scalability and virtual full replication [30]; and distributed fault-tolerant simulation [8]. We are also exploring the use of DeeDS NG in information fusion infrastructure support [7].

29.1.2 Wildfire Scenario

The wildfire scenario is an example of an emergent embedded application of information fusion in emergency management, and can be used to illustrate the new requirements that must be met by the infrastructure, such as resource predictability and high availability, in particular, a degree of autonomy when disconnected or partitioned.

We assume that rescue teams in the wildfire-fighting scenario are equipped with wearable computers that can communicate. In addition, autonomous vehicles can be sent into especially dangerous areas. Each actor shares its view of the local surroundings with other actors through a distributed real-time database, and each actor has access to local as well as (a suitable subset of) global information. Particular subsets of the actors may also share specific local information of mutual interest. Each actor has local hard deadlines to consider, such as updates to status of open retreat paths and remaining supplies. There may be communication hubs that can aggregate knowledge from all actors; for example, there are one or more command centers that have a refined global view of the situation and provide actors with this shared global view as well as operative commands based on that view and the common strategy of the command center. Local information available to an actor may conflict with the global view or information received from other actors (owing to, e.g., separate or incomplete information sources); such conflicts must be detected and resolved in bounded time.

For example, when an actor has access to up-to-date local data, it may take decisions that are locally optimized but that conflicts with the orders assigned to it by a command center. By updating the distributed database, its local information will eventually be made available to other actors in the system. In particular, actors must rely solely on local information in case they become disconnected (unable to communicate with any part of the system other than the local node) or partitioned (able to communicate with a subset of nodes but not with the global command center).

This scenario comprises a dynamically changing environment that is suitable for demonstrating the currently active research problems in DeeDS NG: guaranteeing eventual consistency of replicas, scalability, and durable simulations. It is a relevant scenario, since coordination in catastrophic situations is known to be a problem, in particular, between different agencies, as is the case for large-scale fire fighting.

A distributed database has been proposed by Tatomir and Rothkrantz [39] as a suitable infrastructure for emergency management. It may be used as a whiteboard (or blackboard [33]) for reading and publishing current information about the state of a mission, supporting situation awareness through information collected cooperatively by all the actors. Using a database with transaction support ensures support for consistency of data and the whiteboard architecture avoids the need for specific addressing of recipients within the communication infrastructure. A whiteboard approach for communication has been used in many areas, in particular, as a software architecture approach [16] or for loosely coupled agents [31].

We believe that using a distributed real-time database to support whiteboard communication between nodes is a suitable infrastructure in rescue missions. Publishing and using exchange data through the whiteboard facilitates structured storage and access, implicit consistency management between replicas, fault tolerance, and higher independence of user location by lower coupling between applications. With a whiteboard there is no need to explicitly coordinate communication in distributed applications, which reduces complexity, in particular, where the groups that communicate change dynamically.

29.1.3 Legacy DeeDS Architecture

DeeDS, a distributed active real-time database architecture and prototype, was presented by Andler et al. [3]. DeeDS was developed by the DRTS research group at University of Skövde, Sweden, to address important problems with distributed real-time databases. In this section, we provide a brief overview of the initial design of DeeDS.

The core architecture of a DeeDS node (Figure 29.1) consists of a set of tightly coupled, application-related modules. The *tdbm* module is an off-the-shelf storage manager with basic transaction support. OBST is an independent OBject STore whose storage manager was replaced with *tdbm* to allow nested transactions and improved handling of in-memory data structures. OBST allows object-oriented database schemas to be specified. The rule manager component allows specification and triggering of ECA rules, and was designed and developed specifically for DeeDS.

In addition to the tightly coupled modules, an event-monitoring service and a task scheduler can run asynchronously (loosely coupled) on a separate processor. By decoupling these services from the core modules, their operation does not affect the predictability of the core modules. The event monitor module supports predictable processing of both primitive and composite events, and interacts with

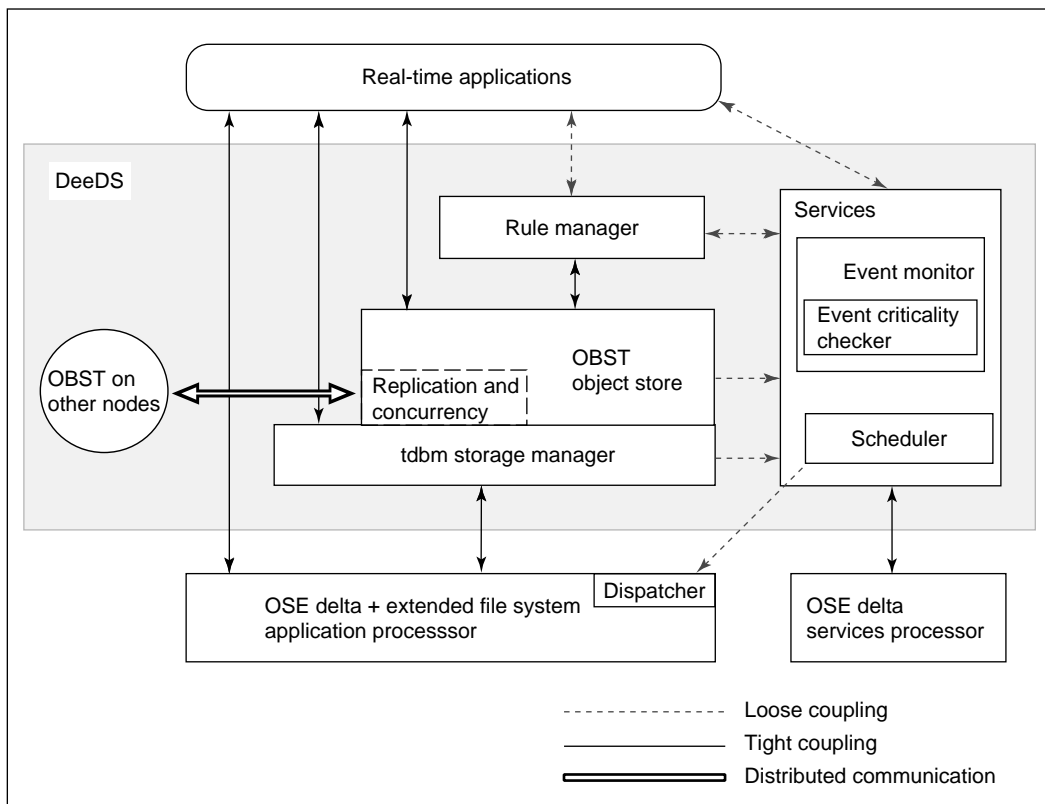


FIGURE 29.1 Component view of the DeeDS design.

the rule manager to enable dynamic execution of ECA rules. The scheduler module supports a mixed criticality workload with overload resolution. During a transient overload, the scheduler allows alternative, less resource-demanding versions of tasks to be executed to ensure predictable operation. While the architecture is designed to allow off-loading of these asynchronous modules to a separate processor, they could also share a processor with the core modules by fixed CPU time allocation.

One of the main goals of DeeDS is to provide a distributed database suitable for systems with hard real-time constraints. Thus, care has been taken to eliminate all sources of unpredictability normally associated with DBSs. For example, DeeDS avoids unpredictable disk access by placing the entire database in main memory. Furthermore, transactions are allowed to commit locally without the need for (potentially unpredictable) remote data accesses and distributed commit protocols. Since worst-case execution times of transactions must be considered when designing a real-time database, using disk storage or distributed commit protocols as in traditional databases would lead to overly pessimistic worst-case execution times [38].

Section 29.2 defines more recent research problems, motivated by emerging applications areas such as sensor networks, distributed simulation, and information fusion. These problems are addressed in an extension of the DeeDS design called DeeDS NG. Section 29.3 details our work on DeeDS NG and our solutions to the research problems of eventual consistency, replication scalability, and fault tolerance introduced in Section 29.2. Section 29.4 describes how our design relates to other work in the areas of consistency, replication, and real-time simulation. Finally, in Section 29.5 we present our contributions and provide a few future directions of our research.

29.2 Active Research Problems

The recent surge of applications in areas, such as information fusion, sensor networks, and distributed simulation, have motivated an extension of the DeeDS design as well as more thorough implementations of certain aspects of DeeDS. A distributed real-time database is a suitable platform for applications in these areas, since they often have real-time constraints and operate in an environment with a dynamic and distributed set of actors. The currently active research problems of optimistic replication, database scalability, and distributed simulations are described in this section, along with notes about how they relate to the wildfire scenario described in Section 29.1.2.

29.2.1 Optimistic Replication

As described in the previous section, DeeDS allows transactions to commit locally without executing a distributed commit protocol. Such optimistic replication potentially introduces conflicts between concurrent transactions executing on separate nodes. Furthermore, transactions may read stale data, i.e., data for which remote updates have not yet been propagated to the local node. Decisions may thus be based on out-of-date data and may need to be refined later as needed. It is required that, in spite of update conflicts, the database continually converges toward a globally consistent state.

In the wildfire scenario, conflicts may occur between directions given by global management and decisions made by actors based on locally available information. The actors must decide whether to follow the global directions or prioritize recently updated local information. Conflict resolution must be based on application-specific policies; for example, if an autonomous vehicle is in danger of colliding with a human, it must be able to make the decision to stop without relying on distributed computation.

29.2.2 Scalability

The transaction model in DeeDS assumes full replication; all data are replicated on all nodes. However, a database that is fully replicated does not scale [18]. To resolve this, scalability in terms of memory usage in DeeDS is achieved by the use of *virtual full replication* through segmentation [29,30]. Virtual full replication ensures that all database objects that are accessed at a node are locally available there. Therefore, applications perceive an image of a fully replicated database. Virtual full replication fulfills all

the properties required by applications to perceive full replication, including availability, data location, consistency, and other application requirements on data.

Under the assumption that replicas of data objects will only be used at a bounded subset of the nodes, the required degree of replication becomes bounded. The usage of the key resource of bandwidth, storage, and processing depends on the degree of replication. These resources are wasted in a fully replicated database compared to a database with a bounded degree of replication. By using knowledge about actual bounded data needs, irrelevant replication is avoided and resource usage is bounded, and thereby scalability can be achieved. However, such virtual full replication makes data available only for pre-specified data needs, and the database loses the flexibility of full replication to execute arbitrary transactions at arbitrary nodes with predictable execution time. To regain flexibility and still maintain scalability, a virtually fully replicated database must also detect and adapt to unspecified data needs that arise during execution.

An emergency management system for use in a wildfire mission needs to be scalable and dynamic. Many different actors need to coordinate and distribute information in real time. Also, the situation may grow very large, involving many more teams and brigades. In such a scenario, actors and information are added and removed dynamically when the mission situation suddenly changes. In a communication infrastructure, each actor will most of the time use only parts of the information for its local actions and for collaboration with close-by peers, and the information needed will change over time.

29.2.3 Distributed Real-Time Simulation

A particularly interesting application area for distributed databases is distributed real-time simulations. Such simulations often mix simulated actors with actual real-time entities, and an execution of the simulation may be very time-consuming and expensive. In particular, it may be very expensive, impractical, or impossible to restart the actual real-time entities if simulation failures occur. Thus, it is important that the simulation can recover and resume execution quickly from a checkpoint in case of a failure. For example, a simulation of a wildfire fighting operation can be very beneficial in the education of fire fighters. Restarting the practice session owing to a simulation failure could be very time-consuming and a waste of resources.

The recovery can be made in two fundamentally different ways either using *rollback recovery* to resume the execution from a previously known healthy state, or by *forward recovery* where the execution is resumed at a future healthy state. The choice of recovery method is determined by the amount of time available for recovery or if there are parts in the simulation process that cannot be rolled back, for example, a real-world action. We define three different degrees of fault tolerance that can be used depending on the recovery requirements. *Fault masking* is the highest degree of fault tolerance. The actual downtime owing to a failure is zero; failures are masked away. *Bounded recovery* is the second highest degree of fault tolerance. The time to detect and recover from a failure is bounded. In other words, if $t_{\text{detection}} + t_{\text{recovery}} \leq t_{\text{deadline}}$, this is a less resource-intensive fault tolerance degree than fault masking. *Best-effort recovery* is the lowest degree of fault tolerance (apart from not having recovery at all). No specific deadline guarantees on how long the system will take to recover is given. However, the value of a recovery is better than doing a full restart of the simulation. The value can be in processor time or some other resources.

A whiteboard architecture supported by a distributed database is a suitable infrastructure for communication and collaboration among nodes in a distributed simulation. DeeDS provides a transaction-based shared memory whiteboard architecture that guarantees real time at the local nodes. DeeDS is also capable of both rollback and forward recoveries, and can implement all the fault tolerance degrees. A simulation infrastructure based on this architecture can be made to support real-time simulations.

29.3 DeeDS NG

In this section, we describe the database model used in the DeeDS NG architecture and provide a definition of eventual consistency, which is a fundamental concept in DeeDS. Solutions to the research problems presented in Section 29.2 are also presented.

29.3.1 Database Model and Eventual Consistency

A *replicated database* maintains a finite set of *logical objects* $\mathcal{O} = \{o_0, o_1, \dots\}$ representing database values. A logical object is only conceptual; it has no explicit manifestation in the database. Instead, logical objects are represented in the database by a set of *object replicas* $\mathcal{R} = \{r_0, r_1, \dots\}$. A database is distributed over a finite set of nodes $\mathcal{N} = \{N_0, N_1, \dots\}$.

A replica $r \in \mathcal{R}$ of logical object o is one node's local view of the state of o . Ideally, all replicas of an object o agree on a common, consistent state of object o (i.e., they are *mutually consistent*), but this may not always be the case if replicas are allowed to be updated separately; an *optimistic replication protocol* allows replica states to temporarily diverge as long as the replication protocol guarantees that replicas eventually become mutually consistent.

A distributed *database* D is a tuple $\langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$, where \mathcal{O} is the set of logical objects in D , \mathcal{R} the set of replicas of objects in \mathcal{O} , and \mathcal{N} a set of nodes such that each node $N \in \mathcal{N}$ hosts at least one replica in \mathcal{R} .

For a database $D = \langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$, the function $R : \mathcal{O} \times \mathcal{N} \rightarrow \mathcal{R} \cup \{\perp\}$ identifies the replica $r \in \mathcal{R}$ of a logical object $o \in \mathcal{O}$ on a particular node $N \in \mathcal{N}$ if such a replica exists: $R(o, N) = r$ if r is the replica of o on node N . If no such replica exists, $R(o, N) = \perp$. The function *node* : $\mathcal{R} \rightarrow \mathcal{N}$ identifies the *host node* $N \in \mathcal{N}$ of a replica $r \in \mathcal{R}$, i.e., $\text{node}(R(o, N)) \doteq N$ is the node where replica r is located. Similarly, the function *object* : $\mathcal{R} \rightarrow \mathcal{O}$ identifies the logical object $o \in \mathcal{O}$ that a replica $r \in \mathcal{R}$ represents, i.e., $\text{object}(R(o, N)) \doteq o$. For a set of replicas \mathcal{R} of logical objects in a set \mathcal{O} , the set $\mathcal{R}(o) \subseteq \mathcal{R}$ consists of all replicas of object $o \in \mathcal{O}$, i.e., $\mathcal{R}(o) \doteq \{r \in \mathcal{R} \mid \text{object}(r) = o\}$.

An *update* operation u accesses and modifies the state of a logical object o . It is *executed* on a single replica in $\mathcal{R}(o)$ but must eventually be applied to every replica of o . For a set of updates \mathcal{U} in a database $\langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$, the function *object* : $\mathcal{U} \rightarrow \mathcal{O}$ identifies the object $o \in \mathcal{O}$ modified by an update $u \in \mathcal{U}$. A *read* operation accesses the state of an object o . Effectively, a read of an object o is executed on some replica $r \in \mathcal{R}(o)$. Read operations and updates can be grouped into a *transaction* T . Execution of a transaction must guarantee the atomicity, consistency, isolation and durability (ACID) transaction properties [19]. The *read set* of a transaction T , written as \mathcal{A}_T , is the set of logical objects accessed by T . The *write set* of T , written as \mathcal{M}_T , is the set of logical objects modified by updates in T . Transactions in DeeDS are always executed on a single node; for a set of transactions \mathcal{T} in a database $\langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$, the function *node* : $\mathcal{T} \rightarrow \mathcal{N}$ identifies the node $n \in \mathcal{N}$ where a transaction $T \in \mathcal{T}$ is executed.

The consistency model for replicated data in DeeDS is based on the concept of eventual mutual consistency of replicas. Informally, replicas of a logical object are eventually mutually consistent if they converge toward an identical state. The definition below is based on a recent formalization of eventual consistency [37]. It is centered around the concept of an *update schedule*; an update schedule for a replica r of object o on node N is an *ordered* set containing all updates u to o that have been integrated but not stabilized on N .

A database $\langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$ is *eventually consistent* if it holds, for every object $o \in \mathcal{O}$, that

1. All replicas in $\mathcal{R}(o)$ begin in identical states (i.e., they are mutually consistent);
2. The update schedules of all replicas in $\mathcal{R}(o)$ have a monotonically increasing and equivalent *committed prefix*;
3. All nonaborted updates in committed prefixes satisfy their preconditions;
4. For any update u to object o submitted at any node, either u or an *abort marker* for u , \bar{u} , eventually becomes part of all committed prefixes of replicas in $\mathcal{R}(o)$. An abort marker denotes that u was aborted/rejected due to a conflict.

In the original definition, the term operation was used instead of update (e.g., update schedules are referred to as operation schedules). Since we use this definition to define eventual mutual consistency of replicas, operations other than updates are irrelevant; the terminology was thus changed to better fit with the rest of the chapter. The precondition mentioned in criterion 3 can vary depending on the system and allows system designers to tailor the specification to their specific requirements [37]. For example, in Bayou [40], preconditions can be expressed as part of the transaction specification.

Eventual consistency is a very flexible concept, in that it effectively allows applications to choose whether to use consistent or tentative data. Let the *stable state* of a replica r be the state of r after applying all the updates in the committed prefix of its update schedule. Similarly, let the *optimistic state* of r be the state of r after applying all updates in the update schedule. Applications can choose to read either the stable or the tentative state of any object they access. Furthermore, different consistency guarantees can be used for the committed prefix and the tentative suffix; for example, the committed prefix can be kept sequentially consistent [6].

DeeDS NG uses a consistency model based on eventual consistency. In DeeDS NG, stabilization of an update u includes moving u from the tentative suffix of the relevant update schedule to the committed prefix. Committed prefixes are guaranteed to be sequentially and causally consistent, and application-specific policies are used to resolve conflicts among causally concurrent updates. Although updates in a tentative suffix are ordered according to their causal order, updates executed on remote nodes may still be missing from the suffix.

29.3.2 Optimistic Replication

The approach for data replication in DeeDS is based on the conjecture that in a typical DRTDBS, predictability, efficiency, and consistency of local operations are more critical than immediate global consistency. In other words, there are hard or firm real-time constraints on the commit time and resource usage of local transactions, while temporal constraints on data sharing between applications or subsystems on different nodes are more relaxed. This section contains a brief overview of PRiDe (Protocol for Replication in DeeDS), a replication protocol that has been designed to ensure eventual consistency as defined in the previous section.

29.3.2.1 Replication Protocol Overview

PRiDe, the replication protocol in DeeDS, is an optimistic replication protocol [37] that consists of five phases: the local update phase, the propagation phase, the integration phase, the stabilization phase, and the compensation phase. In the *local update* phase, an application running on node N submits a transaction T to the local database manager on N , which executes the transaction locally and either commits or aborts the transaction. The commit decision is made without contacting any of the database managers on remote nodes. If T commits, the protocol enters the *propagation* phase, during which information about T 's updates are sent in a *propagation message*, p_T , to all nodes that host replicas of objects updated by T . This set of nodes is referred to as the *recipient set* of T , \mathcal{REC}_T .

The *integration phase* is executed separately on each node in \mathcal{REC}_T , and can be initiated at any time after the reception of p_T as long as all deadlines are met. During the integration phase, each update contained in p_T is integrated in the local database, and the data structures used for conflict detection and resolution are updated as required.

The *stabilization phase* is executed separately on each node $M \in \mathcal{REC}_T$ for each update u in p_T that has been integrated on M . During this phase, any conflicts involving u are resolved, and unless u is undone in conflict resolution, it is permanently applied to the local database at M . Resolving all conflicts involving u and applying u to the database is referred to as *stabilizing u* .

Owing to conflicts and conflict resolution, the actions of a transaction T may be compromised during stabilization; for example, an account withdrawal may be discarded due to concurrent withdrawals emptying the account. During the *compensation phase*, an application can react to such compromises by *compensating* for the changes made in conflict resolution. This is discussed in more detail in Section 29.3.2.5.

The stabilization and compensation phase must not be executed for an update u until it can be guaranteed that all information necessary to resolve any conflicts involving u is available.

29.3.2.2 Conflict Types

Optimistic replication introduces temporary inconsistencies between local databases by allowing execution of operations on a single node's replicas before coordinating with other nodes that host replicas of the same objects. Temporary inconsistency have two distinct consequences for applications: the possibility of reading *stale* data and the risk of executing *conflicting updates* on separate nodes [21].

A *stale read* occurs if an application A submits an update u of object o on node N and another application B later reads o on another node M before u has been integrated on M . The temporary mutual inconsistency between the replicas of o on nodes N and M caused application B to read a value of o , which was not up-to-date with regard to u . A *write–write conflict* occurs if two applications individually submit updates u, v of object o to nodes N, M , and neither update causally precedes the other; that is, v was submitted before u was integrated on M , and u was submitted before v was integrated on N . Until the conflict between u and v has been resolved, N and M will host mutually inconsistent replicas of o .

While pessimistic replication approaches aim to *prevent* conflicts by enforcing safety constraint on concurrent operation execution, an optimistic approach speculates that no conflicts will occur and instead *detects* and *resolves* the conflicts that may occur. Optimistic replication protocols are correct only if conflicts are correctly detected and resolved on all nodes. Therefore, correctness of PRiDe depends on its ability to guarantee that update conflicts are eventually detected and resolved deterministically on all nodes that host replicas of the updated objects. In other words, the protocol must provide eventual consistency as per the definition in Section 29.3.1.

29.3.2.3 Conflict Detection

In PRiDe, updates in an update schedule are grouped by their *generation numbers*, which are effectively logical timestamps [27] that impose a causal order on updates to the same replica. Let $g(u)$ be the generation number of an update u . If update u precedes update v , written $u < v$, it holds that $g(u) < g(v)$. For a single object o , updates with the same generation number (said to be in the same *generation*) are not causally ordered; a write–write conflict exists between all such updates. Let G be a generation of updates in an update schedule S . G is *complete* if it can be guaranteed that no other updates in G will be added to S in the future. Updates in a generation G can be stabilized safely only if G is complete.

Let r be a replica of object o on node N , and define $S(r)$ to be the update schedule of r . To ensure that generation numbers accurately model causal dependencies, the generation number of a local update u to object o on node N is set to the one higher than the highest generation number among updates in $S(r)$ at the time of executing u . When a node receives a propagation message containing an update u of object o in generation G , if it has not performed a local update to o in G , it sends a *stabilization message* for G to the nodes hosting replicas of o . Stabilization messages are used to decide whether generations are complete. Let r be the replica of object o on node N ; a generation G of updates in $S(r)$ is guaranteed to be complete if, for every node $M \neq N$ hosting a replica of o , $S(r)$ contains an update to o by M in G , or N has received a stabilization message for G from M .

This approach to conflict detection allows each node to independently create a causally consistent update order as it receives propagation messages from other nodes. Conflict detection is thus entirely distributed and deterministic, removing the need for any central update scheduler.

29.3.2.4 Conflict Resolution

Most optimistic replication protocols resolve conflicts by undoing [47] or compensating for [11] conflicting updates. This type of *backward* conflict resolution (which rolls the system *back* to a previous, correct state) works well in systems where a simple consistency model is sufficient and average update throughput is more important than meeting temporal constraints.

In *forward* conflict resolution, conflicting updates or transactions are not undone; instead, conflict resolution transforms the database state to a *new* state that merges or prioritizes among the conflicting updates or transactions while respecting consistency constraints in the database. Typically, forward conflict resolution is performed manually, such as in version management systems [10] or file systems [32].

Automatic forward conflict resolution is rare, but is used in, for example, Bayou [40] and some collaborative editing approaches [41].

In safety-critical real-time systems, undoing conflicting updates or rolling back conflicting transactions may violate temporal constraints. In such systems, the *worst-case* execution times of transactions is the critical metric, since the system must meet deadlines during the worst possible conditions. If transactions can be rolled back and reexecuted, this possibility must be considered when computing worst-case transaction execution times. Furthermore, transactions in real-time systems may perform *external actions*, i.e., actions that affect the physical environment (such as drilling a hole). Such actions cannot easily be undone, making transaction rollback difficult.

In PRiDe, the unit of conflict resolution is a generation. Whenever a generation is complete, any conflicts among updates in that generation are resolved, and the (possibly modified) updates are moved to the committed prefix of their update schedule. The stable version of the updated replica is also replaced to reflect the update stabilization.

Like conflict detection, conflict resolution is entirely distributed; each individual node performs conflict resolution for each generation of updates. It is guaranteed that every node sees the same updates in every generation; however, to ensure eventual mutual consistency the conflict resolution policy used must be the same on every node, and its results must be deterministic and independent of the node where it was executed.

29.3.2.5 Transaction Conflicts and Compensation

As presented in the previous sections, conflict management in DeeDS is data centric, and conflicts are detected and resolved on the update level. However, a transaction can contain a set of database accesses in the form of read and write operations, each of which can be in conflict with operations performed by concurrent transactions.

While a transaction's updates are handled separately in conflict detection and conflict resolution, each transaction T can define a *compensation procedure* that can be used to react to any compromises made for T 's updates. Once all updates made by a transaction T have stabilized on the node N where it was originally executed, its compensation procedure is executed on N . The compensation method can access and evaluate the result of each of T 's updates (e.g., an update's parameters could have been changed in conflict resolution, or the update could have been undone). Based on this information, compensating measures may be taken; for example, if a transaction's updates were discarded during conflict resolution, the updates could be resubmitted in a new transaction during compensation.

A *read-write conflict* occurs when transactions concurrently update and read an object. Such conflicts are typically not a big problem in systems that use optimistic replication, since the choice of this type of replication suggests that applications can tolerate reads of inconsistent or stale data. However, PRiDe allows transactions to use *assertions* to read database values. An assertion is a read operation (i.e., it does not change the database state) that is added to the update schedules of all replicas of the queried object just as if it were an update. Conflict resolution policies can thus react to any assertions present in a stabilizing generation. Furthermore, assertions allow compensation procedures to react to inconsistent reads; the compensation procedure for a transaction T performing an assertion a has access both to the original result of a (i.e., the optimistic value returned to T when executing a) and the actual value of the queried object after stabilizing the generation containing a . The compensation method for a transaction T is not executed until all of T 's updates *and* assertions have stabilized.

29.3.3 Scalability and Virtual Full Replication

Scalability is an increasingly important issue for distributed real-time databases, since real-time database applications become increasingly larger, in terms of the size of the logical database, the number of nodes involved, and the number of users of the data, such as with vehicle databases [34]. For some systems the available resources are very scarce, such as with sensor networks where bandwidth, storage, and processing are very limited [15].

With *virtual full replication* [3,29], the database clients are provided with an image of a fully replicated database. It is a resource management approach that lets the DBS exploit knowledge of individual client data access requirements to provide them with such an image. Virtual full replication ensures that data objects are replicated on the subset of nodes where those objects are needed, while bounding resource costs for maintaining an image of full replication. Formally, for a database $D = \langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$ and a set of transactions \mathcal{T} executing on D , virtual replication ensures that $\forall o \in \mathcal{O}, \forall T \in \mathcal{T} (o \in \{\mathcal{A}_T \cup \mathcal{M}_T\} \rightarrow \exists r \in \mathcal{R} (r = R(o, \text{node}(T))))$, where $\text{node}(T)$ is the node where T was executed.

The *scale factor* is a system parameter along which the system is scaled, such as the size of the database (number of objects or number of transactions) or the number of nodes used. If the degree of replication, k , is bounded by a constant or a function that grows significantly slower than the scale factor, then we can show that replication processing is bounded. With such bounded resource management, scalability is ensured without changing the application's assumption of having a complete database replica available at each local node.

Obviously, virtual full replication based only on *a priori* specified data accesses does not make arbitrary data available for accesses at an arbitrarily selected node. This makes such a system less flexible for transaction scheduling than a fully replicated database. To support unspecified data accesses and changes in data requirements, virtual full replication needs to *adapt* to such changes by incrementally reconfiguring the initial replication setup. This preserves scalability by managing resource predictability over time.

We assess scalability using analysis, simulation, and implementation, while evaluating the proposed scalability enablers. We introduce the concept of *segmentation* to group data objects that share properties, such as where to allocate replicas of the objects. As an initial evaluation of segmentation, as a means of implementing virtual full replication for a reasonably sized system, we have analyzed and implemented static segmentation in the DeeDS NG database prototype. With an adaptive virtually fully replicated database, however, scalability is more complex to analyze. Therefore, we have developed an accurate simulation of adaptive virtual full replication, to evaluate scalability of adaptive segmentation. The simulation allows the study of large-scale systems without requiring a large amount of physical resources. We finally present our findings about data properties and their relations to each other, which are used in our segmentation approach.

29.3.3.1 Static Segmentation

Our approach for static segmentation [30] is based on sets of properties associated with each database object; objects with identical or similar property sets are combined into segments that each have a bounded number of replicas. Consider as a first example segmentation for a single property, the *access location* for the data objects. For the example, consider a database $\langle \{o_1, \dots, o_6\}, \mathcal{R}, \{N_1, \dots, N_5\} \rangle$, where the contents of \mathcal{R} are determined by the segmentation approach. Seven transactions, T_1, \dots, T_7 , execute in the system. Each transaction has a read set, a write set, and a *host node* where it is initiated and executed. In the following *a priori* specification, each transaction is associated with a tuple specifying its read set, its write set, and its host node $T = \langle \mathcal{A}_T, \mathcal{M}_T, N \rangle$. $T_1 : \langle \{o_1, o_6\}, \{o_6\}, N_1 \rangle$, $T_2 : \langle \{o_1, o_6\}, \{o_6\}, N_4 \rangle$, $T_3 : \langle \{o_3\}, \{o_5\}, N_3 \rangle$, $T_4 : \langle \{o_5\}, \{o_3\}, N_2 \rangle$, $T_5 : \langle \{o_2\}, \{o_2\}, N_2 \rangle$, $T_6 : \langle \{o_2\}, \{o_2\}, N_5 \rangle$, and $T_7 : \langle \{o_4\}, N_3 \rangle$.

The data accesses result in the following access sets: $o_1 : \{N_1, N_4\}$, $o_2 : \{N_2, N_5\}$, $o_3 : \{N_2, N_3\}$, $o_4 : \{N_3\}$, $o_5 : \{N_2, N_3\}$, and $o_6 : \{N_1, N_4\}$. An optimal replica allocation needs to use four segments. Representing a segment by a pair $\langle \mathcal{O}, \mathcal{N} \rangle$, where \mathcal{O} is the data objects in the segment and \mathcal{N} the set of nodes hosting replicas of objects in the segment, the resulting segments in the example are $s_1 = \langle \{o_4\}, \{N_3\} \rangle$, $s_2 = \langle \{o_3, o_5\}, \{N_2, N_3\} \rangle$, $s_3 = \langle \{o_1, o_6\}, \{N_1, N_4\} \rangle$, and $s_4 = \langle \{o_2\}, \{N_2, N_5\} \rangle$.

Objects are assigned to segments by the segmentation algorithm, which represents objects accesses as a table, where the rows represent data objects and columns represent nodes (Figure 29.2, left). By assigning a binary value to each column, each row can be interpreted as a binary number that forms a segment key to identify the nodes where a replica of the object is needed. The table is sorted on the segment key (Figure 29.2, right). Passing through the sorted table once, rows with the same key value can be grouped into unique segments and allocated to nodes as required. The sort operation in this algorithm contributes with the highest computational complexity. Thus, this algorithm segments a database $\langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$ in time $O(|\mathcal{O}| \log |\mathcal{O}|)$.

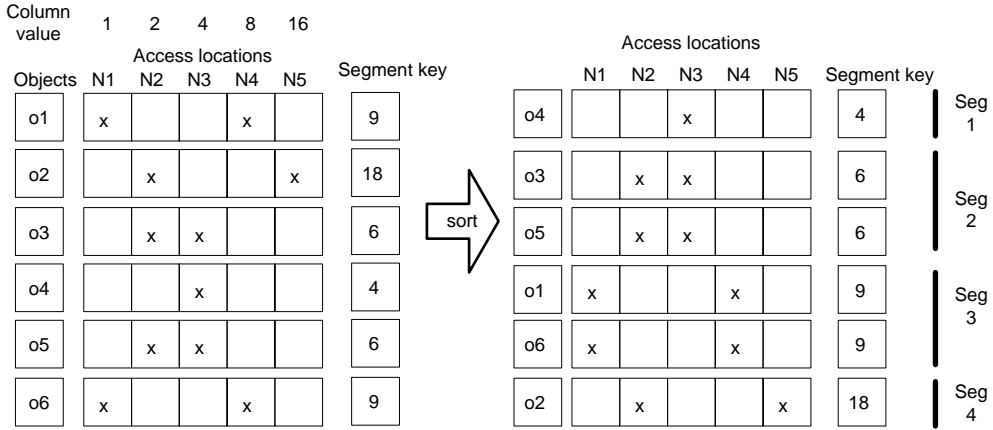


FIGURE 29.2 Static segmentation for data allocation to meet access locations.

Grouping by access locations can be generalized to any generic property. Consider an extension to the first example where we add *replication requirements*. The database clients require that object o_2 must have a bound on the replication time. Objects o_1 and o_6 may be temporarily inconsistent, while objects o_3, o_4 , and o_5 need to be immediately consistent at every update. Further, objects o_1, o_3, o_5 , and o_6 are specified to be stored in main memory, while objects o_2 and o_4 could be stored on disk. The property sets of the objects then become (extending the notation for an access set to a tuple that includes the new properties)

$$\begin{aligned}
 o_1 &: \{ \langle N_1, N_4 \rangle, \text{asap}, \text{memory} \}, & o_2 &: \{ \langle N_2, N_5 \rangle, \text{bounded}, \text{disk} \}, \\
 o_3 &: \{ \langle N_2, N_3 \rangle, \text{immediate}, \text{disk} \}, & o_4 &: \{ \langle N_3 \rangle, \text{immediate}, \text{disk} \}, \\
 o_5 &: \{ \langle N_2, N_3 \rangle, \text{immediate}, \text{memory} \}, & o_6 &: \{ \langle N_1, N_4 \rangle, \text{asap}, \text{memory} \}.
 \end{aligned}$$

Each row can still be represented as a binary number by assigning a unique binary value (e.g., a perfect hash value) to each individual property value. By selecting different subset of properties, multiple segmentations can be generated from a subset of the complete property set. For instance, units of physical allocation can be generated from a combination of “access” and “storage medium” properties. By executing the segmentation algorithm on this subset, the resulting segmentation gives the segments for physical allocation to nodes that can also be used for recovery of individual segments. Also with multiple segmentations, the algorithm segments the database in time $O(|\mathcal{O}| \log |\mathcal{O}|)$.

29.3.3.2 Properties, Dependencies, and Rules

Certain conditions must be satisfied for combinations of properties. For instance, to guarantee transaction timeliness three conditions must be fulfilled: main memory residence, local availability, and detached replication. Detached replication can be done as soon as possible (asap), where there is no guarantee for timeliness of replication, or in bounded time (bounded) with a bound on the replication time. Consider the property set of o_2 in the extended example. To guarantee timeliness, the storage of object o_2 must be changed into main memory storage to make the replication method property consistent with the storage property. Thus, known dependencies between properties are used to ensure compatibility between properties.

Also, objects o_3 and o_5 can be combined into the same segment by changing the storage method for either object. Both need to be immediately consistent, but there is no property relation rule that requires a certain type of storage for that consistency. The objects can thus be stored on disk, which is normally a cheaper resource than memory. By additionally allocating o_4 to node N_2 , a segment of objects $\{o_3, o_4, o_5\}$ can be created. The additional storage is not optimal according to specified data accesses but may still be a valid choice for reducing the number of segments. Other rules can be added; for example, a *required*

minimum degree of replication can be specified if a certain level of fault tolerance must be guaranteed, or objects can be manually *clustered* into groups of objects with identical values set for all their properties.

Resource analysis: To evaluate resource usage of static segmentation in a database D , we choose to analyze three key resources. Let \mathcal{I} be the set of *transaction instances* used in D . A transaction instance is a series of executions of a particular *transaction program*. A transaction instance $T_j \in \mathcal{I}$ executes its transaction program with a frequency f_j . It is assumed that all executions of a transaction instance use the read and write sets of the transaction program; for this analysis, only the size of the write set is relevant. Let $WRITE_{T_j}$ be the write set of every execution of a transaction instance $T_j \in \mathcal{I}$, and let $w_j = |WRITE_{T_j}|$.

Bandwidth usage depends on the number of updates replicated. Every execution of transaction instance T_j generates w_j update messages; one for each update in $WRITE_{T_j}$. We assume point-to-point replication, so each message is sent to the $k - 1$ nodes hosting replicas of the updated object. Such network messages are generated by all transaction instances independent of the execution node. We express bandwidth usage as $(k - 1) \sum_{T_j \in \mathcal{I}} w_j * f_j$ [messages/s]. For a virtually fully replicated database, with a limit k on the degree of replication, bandwidth usage scales with number of replicas, $O(k)$, rather than with number of nodes, $O(n)$, as is the case with a fully replicated database. It also scales linearly with the number of updates made by each transaction and by the transaction frequency. However, the number of transactions often depends on the number of nodes, $O(n)$, so bandwidth usage may become $O(kn)$ for the virtually replicated database as compared to $O(n^2)$ for the fully replicated database.

For *storage* of a database with a set of segments \mathcal{S} , there are k_i replicas of each segment $s_i \in \mathcal{S}$. We express the required storage as $\sum_{s_i \in \mathcal{S}} (|s_i| * k_i)$ [objects]. For the purpose of this analysis, we assume the same degree of replication, k , for all segments. Then, each data object o in the database is replicated at k nodes and the required storage can be expressed as $o * k$. With full replication to n nodes, n replicas of each data object is stored and the required storage is $o * n$. This means that a virtually fully replicated database scales with the limit on replication, k , rather than with the number of nodes, n , in the system, as is the case with a fully replicated database.

Processing time for executing updates on a local node, L , includes time for locking, updating, and unlocking the updated data objects. Propagation of local updates to remote nodes requires logging, marshaling, and sending the update on the network, P . With point-to-point communication, P is required for each other replica of an updated object. Updates for replicas are *integrated* at the receiving node requiring time for receiving, unmarshaling, detecting conflicts, resolving conflicts, locking data objects, updating database objects, and finally unlocking updated data objects, I . Conflict detection and resolution time is denoted by C . The processing time used for replication of a transaction instance T_j depends on the size of T_j 's write set, w_j , resulting in propagation time, P , and integration time, I , for each node to which the transaction's updates must be replicated. We express the overall processing time as $\sum_{T_j \in \mathcal{I}} f_j [Lw_j + (k - 1)\{(P + I)w_j + C\}]$. The formula shows that processing is constant with the degree of replication and not growing with the number of nodes in the system.

29.3.3.3 Adaptive Segmentation

With a static specification of the database usage, the virtually fully replicated database makes data objects available only for the *a priori* specified data accesses. A wildfire scenario is highly dynamic and a *priori* specification of data accesses may not be possible. To increase flexibility in scheduling of transactions, we introduce adaptive segmentation, which allows unspecified data accesses for soft real-time transactions. Unspecified data objects are loaded on demand and the database is resegmented based on new information that becomes available during execution time. A scalable approach lists the nodes for the replicas of the segments stored at a node, to enable detection of missing data objects at the node. However, it is not scalable to store information at all nodes of allocation of all database objects, so we use a replicated directory service to allow nodes to find replicas of missing objects, where the directory information has a bounded degree of replication in the system.

For adaptiveness, we primarily consider the following changes to the replication schema: (1) unspecified accesses, (2) added and removed nodes, and (3) added and removed data objects. Also, redundancy for fault tolerance is considered. We can apply a minimum degree of replication for a segment by manipulating the

property table. When a node fails, the minimum replication degree may need to be restored, to maintain the level of fault tolerance. The segmentation of the database is changed incrementally using *add()* and *remove()* operations, at the local node and at the directory. These operations commute, and execute in $O(s + o)$ time, where s is the number of segments and o is the number of objects.

We use incremental recovery [5] to load segments for data objects missing at a node. On-demand loading of segments has an unbounded delay for the first transaction that is missing the data, unless there is a bound on recovery time. This is caused by the unpredictability of the network and by incremental recovery where the load on the recovery source is unknown. This makes adaptive segmentation unsuitable when there are hard transactions only. Loading and unloading time points of segments must be selected carefully. The combination of the amount and the frequency of changes, and the available storage, may cause “thrashing,” where the DBS is busy with loading and unloading data without leaving processing time for database transactions. Adding and removing data objects from nodes have some resemblance to database buffering, virtual memory, cache coherency, and garbage collection. Our approach relates to these concepts in that only a part of a larger data or instruction set needs to be available, and that this changes over time.

Removal of replicas must not cause a local replica to be unavailable at a time when it needs to be accessed by a hard transaction. *Pinning* of such data objects is one approach to prevent data objects that are accessed by transactions with hard real-time requirements from being removed from memory. This is done by an *a priori* specification of which objects need to be pinned into main memory.

Adaptive segmentation with prefetch: We further extend adaptive segmentation with prefetch of data to nodes, to make data available at a certain time point or repeating time points, which enables adaptation for hard transactions as well. A requirement is that the time instant of an access can be specified, either as a single instance or as a minimum interval for periodic accesses. Bounded time incremental recovery is also required. For soft real-time transactions, prefetch reduces the waiting time at the first access.

It may also be possible to detect more complex access patterns by more advanced pattern detection and prediction, which may be used to further improve timeliness for soft real-time transactions.

29.3.4 Simulation DeeDS: A Way to Implement Real-Time Simulations

Optimistic real-time simulations based on *Time Warp* [25] has been implemented for parallel real-time simulation (e.g., Aggressive No-Risk Time Warp [17]), where there is a shared memory to use for communication. The distributed, active real-time database (DRTDBS) described in Section 29.1.3 provides a shared memory (whiteboard) architecture that can guarantee local hard real-time. By putting the data structures used in Time Warp into a DRTDBS we provide a Time Warp implementation, called Simulation DeeDS, that is distributed and supports real-time. In the original Time Warp, a *logical process* (LP) mimics a real-world process, for example, a rescue team in the wildfire scenario, and each LP has its own *local virtual time* (LVT), input and output message queues, and a state vector.

For our implementation each LP is connected to a unique database node. This means that each database node will hold the LVT, input and output queues, and state for all the LPs in the distributed simulation. All the LPs share a common time called *global virtual time* (GVT), which is the lowest LVT among the LPs. Figure 29.3 illustrates a simulation with three LPs and the corresponding database nodes. The GVT can be calculated continuously by using a tree structure where the LVTs of the LPs are leaf nodes and the GVT is the root node [17]. Furthermore, since the database is active, it is possible to specify rules in the database that automatically update the GVT tree. For example, a rule could look like this: *ON update(LP_i.LVT) IF LP_i ≠ root & LP_i.LVT < LP_i.parent THEN update(LVT_i.parent)*, where LVT_i is the LVT of logical process LP_i.

The basic operation of the database-driven approach follows. Messages in input and output queues are tuples consisting of a time of occurrence (timestamp) and the corresponding action(s), for example, $m_1(15, x = 1)$ means that x is set to 1 at LVT 15. The queues are sorted on time of occurrence with the lowest time first. When the simulation starts, each LP's LVT is set to ∞ and the GVT is set to 0. The processing in each logical process LP_i consists of four steps: (1) Take the first message (m_{head}) in LP_i's

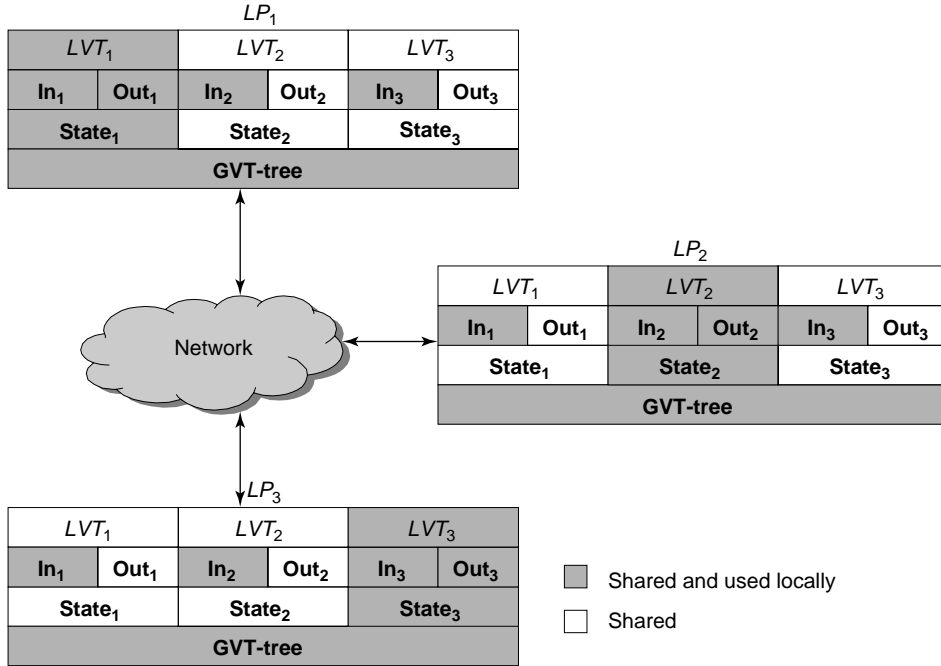


FIGURE 29.3 Data structures of Time Warp in the database.

input queue. If $m_{head}.timestamp < LVT_i$ then a straggler* has been found and processing must be rolled back to a suitable recovery point and restarted. Then, set $LVT_i = m_{head}.timestamp$ and perform the actions in m_{head} on LP_i 's state. (2) After successfully processing a message the result must be sent to all LPs that use (subscribe to) the result. This is done in the original Time Warp by writing the result in LP_i 's output queue and the input queue of every process LP_j such that LP_j uses the result of LP_i . (3) Update the GVT by checking if $LVT_i < LVT_{parent}$ in the GVT tree and update the tree if true. (4) Check if the simulation has finished, i.e., if $GVT = \infty$.

29.3.4.1 Improved Database Approach

The original Time Warp protocol described in the previous section does not use many of the DRTDBS features, such as active functionality; it merely uses the database as a message communication system and a storage facility. By adapting the Aggressive No-Risk Time Warp protocol to exploit more of the database features an improved database approach can be obtained. For example, memory usage can be reduced by removing the message queues, using active functionality rules instead to notify LPs of changes that were previously stored as messages. Since the database is fully replicated and active, it is not necessary to explicitly send messages between LPs by storing values in their respective input queues. Updates can be made directly to the state variables, and active rules can monitor such updates. The event queues effectively store unprocessed values of the state variables up to the GVT. This means that for a simulation with n logical processes, a single state with no explicit message queues could be used instead of $2 * n$ message queues and n states. The single state, however, would be a merger of the n states and each state variable is complemented by the event queue that maintains a record of generated future values with $timestamps > GVT$. Figure 29.4 shows three LPs that share state and have no explicit input or output queues. The "messaging" is provided by the automatic replication and the active functionality, which triggers the LPs to act when updates to state variables that they subscribe to are detected.

*A straggler is a message that has been processed out of order and will trigger a rollback.

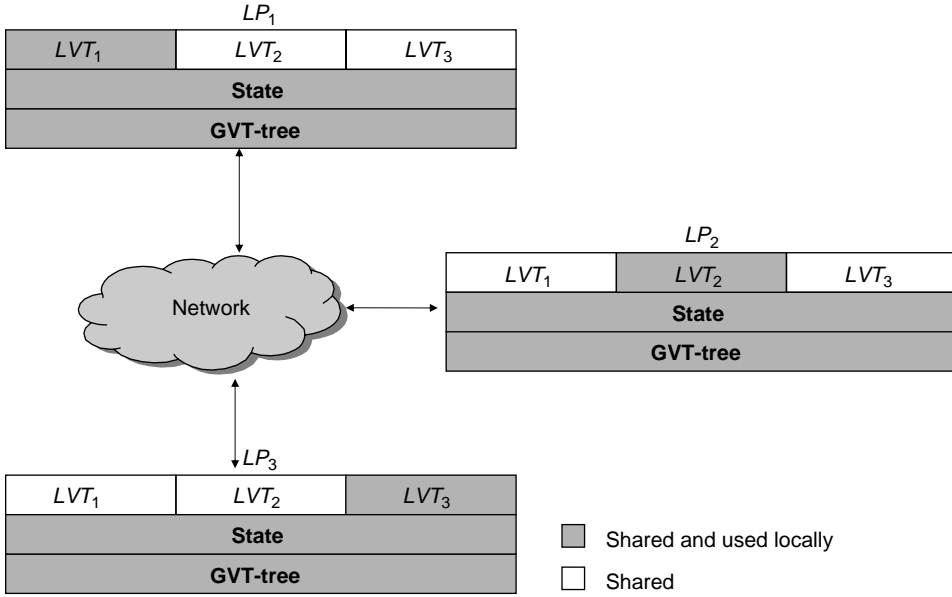


FIGURE 29.4 Aggressive No-Risk Time Warp in the database using active functionality.

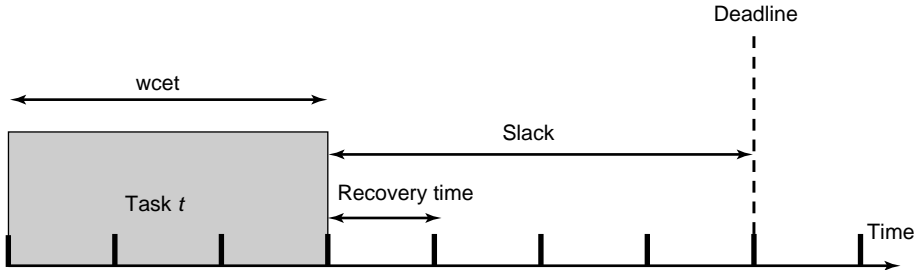


FIGURE 29.5 The slack time for a task t .

In real-time systems, all critical tasks t have deadlines t_{deadline} and requires an estimate of their worst-case execution times (t_{wcet}). For a task t , the slack time t_{slack} is $t_{\text{slack}} = t_{\text{deadline}} - t_{\text{wcet}}$ [28]. Assuming that a task has a bounded recovery time (t_{recover}) we can guarantee that the task t will finish before it's deadline iff $t_{\text{slack}} \geq t_{\text{recover}}$ * (see Figure 29.5). This is true under the assumption that a task fails at most once. Checkpoints can be used to avoid restarting a crashed task from the start. The use of checkpoints divides a task into smaller units that, once executed to completion, do not need to be reexecuted in case of a crash. Instead, the recovery procedure resumes execution at the nearest checkpoint prior to the crash. The wcet for a task with n parts is then defined as $\sum_{i=1}^n t_{\text{wcet_part}}^i$. If there is a crash in part j then the following formula must hold: $\sum_{i=1}^j t_{\text{wcet_part}}^i + t_{\text{recover}} + \sum_{i=j}^n t_{\text{wcet_part}}^i \leq t_{\text{deadline}}$. Factoring leads to $t_{\text{recover}} + t_{\text{wcet_part}}^j \leq t_{\text{slack}}$, i.e., the slack must be greater than the recovery time and the wcet for the reexecution for the crashed part.

A *recovery line* (also called as snapshot) is a set of checkpoints taken at the same time in all participating nodes in a distributed system. In a distributed simulation, assume that a recovery line is forced (just)

*Actually $n * t_{\text{recover}}$ if n recoveries are needed.

before interaction with the real world. All real-time interaction is in response to an event (or time trigger) t_e , and a response is required at or before t_{deadline} , where $t_{\text{wcet}} + t_{\text{slack}} = t_{\text{deadline}} - t_e$. Thus, we need to ensure that recovery time and reexecution fits within the slack. In other words, we can guarantee that we are able to recover from a crash if the slack is designed right, and if t_{recover} is small enough. In the case of a real-world interaction, enough slack must thus be allowed in the interaction so that we may tolerate crashes in the simulation parts of our system. Depending on the amount of slack allowed, this may require fault masking or forward recovery, impacting the fault tolerance degree required (see Section 29.2.3).

29.4 Related Work

This section briefly describes related approaches to optimistic replication, virtual full replication, and real-time simulation.

Bayou [40] is a replicated storage system intended primarily for mobile environments; that is, environments where nodes frequently become disconnected from parts of the network. Like DeeDS, Bayou does not aim to provide transparent replication, with the argument that applications must be aware of possible inconsistencies, and aid in resolving them. Bayou updates are executions of stored procedures and are tentative until they have become stable. Updates are propagated to and reexecuted on other database nodes containing replicas of the updated objects, and applications are allowed to read values performed by tentative updates. Tentative updates are kept in an update schedule, and a central scheduler continuously creates a canonical update order. Updates in a node's update schedule stabilize as information about the canonical order is retrieved from the central scheduler, and are thus moved from the tentative suffix of the log to the committed prefix. Since this may change the order of the updates in the log, updates in the tentative suffix may be undone and reexecuted.

The stored procedures in Bayou are specified as SQL queries, and contain an application-specific evaluation of update preconditions in the form of a *dependency check*. If the dependency check evaluates to *false*, a semantic conflict has been detected, and forward conflict resolution is performed through an application-specific *merge procedure*. For example, if a Bayou update attempts to reserve a meeting room at a specific time, the dependency check may detect that an earlier update in the canonical order conflicts with the reservation. The merge procedure may then try to reserve another room, or reserve the room at an alternative time.

Bayou is similar to DeeDS in that it uses semantic conflict detection in addition to syntactic conflict detection (the central scheduler uses version vectors [36] to detect causal dependencies between updates). Furthermore, Bayou, like DeeDS, uses application-specific forward conflict resolution when possible. The most important difference between DeeDS and Bayou is that where Bayou depends on the availability of a central scheduler, the replication protocol in DeeDS is fully distributed; the canonical update order is derived from the generation numbers on each individual node.

Scalable real-time databases: For replication models of large-scale distributed databases, there are several approaches. In contrast to our peer node approach, the hierarchical asynchronous replication protocol [1] orders nodes in a hierarchy, and replicates to neighbor nodes with two different replication methods: one that uses global commit and another that propagates independently where updates eventually reaches all nodes. Also, epidemic replication is used to disseminate updates [22] in very large networks. Optimal placement of replicas minimizes distributed transactions, and allows large systems [45]. Our approach eliminates the need for distributed transactions.

A framework is available for comparing replication techniques for distributed systems, in particular, for distributed databases [42].

Adaptive replication and local replicas: The need for adaptive change of allocation and replication of database objects in (large) distributed databases has been identified in literature [44], including relational [9] and object database approaches [46]. Our approach for adaptive allocation of data objects (both allocation and deallocation) also relates to the abundant work available in the areas of cache coherency [2,35,43], database buffer management [12,23,24] and virtual memory [14].

Real-time simulation: Using Simulation DeeDS allows implicit storage of simulation data, i.e., no special attention is needed to save simulation states or results. This can be compared with, for example, HLA where one common way to achieve storage of simulation data is to have one passive simulation node (called passive federate) tap into the simulation (called federation) and collect all simulation data [26].

29.5 Summary

In this section, we present our conclusions and outline our future work with DeeDS NG.

29.5.1 Conclusions

A database model for a DRTDBS has been presented and justified.

An optimistic replication protocol, PRiDe, that supports local transaction commit has been presented. The protocol allows applications to read either optimistic or stable data, and is designed to allow deterministic forward resolution of update conflicts. Applications can compensate for stale reads and update conflicts by defining application-specific forward compensation procedures. The protocol is particularly suited for real-time databases since it allows local commit of transactions, avoiding distributed transactions and the potential resource unpredictability inherent in distributed commit protocols.

With virtual full replication, we have presented a novel approach for resource management in distributed real-time databases, which is used to achieve scalability through bounded resource usage. Based on knowledge about actual data needs, an image of full replication is available for database clients. We quantify scalability in the setting to evaluate the scalability achieved, and find out the usage conditions.

A major benefit in using our optimistic concurrency protocol is that real-time simulations with external actions can store all their state variables in the database, which can be seen as a whiteboard or a shared memory. Even though the simulation applications are distributed, communication issues are hidden from the application programmer or simulation engineer. Also, adding and removing simulation nodes are potentially less cumbersome, since all logical processes are decoupled by the database, as opposed to, for example, distributed simulations with peer-to-peer connections. All communication is catered for by the underlying replication mechanism. The database can provide local real-time guarantees, since it is main memory based, is fully replicated, and allows local commits. However, as indicated above, this optimistic design comes with a price: inconsistencies between nodes can exist, even though locally at each node the database is consistent, and must be tolerated by applications.

29.5.2 Future Work

The replication architecture as described in this chapter is very sensitive to faults; it therefore assumes reliable message delivery, as a single message omission would cause the protocol to fail. While reliable message delivery is provided by many communication protocols, this requirement could be relaxed and progress of the replication protocol could be improved in lossy networks, by adding redundancy to the protocol. For example, nodes could preemptively forward update information that appears to be missing on other nodes, based on information in the conflict manager.

Our adaptive approach for virtual full replication considers simple access patterns for prefetching of data, like single instances of periodically executing transactions. By using more advanced pattern detection and prediction techniques, the waiting time for segment adaptation could be shortened and allow faster adaptation, also in advance of changed data needs.

In the present work, we provide no guidelines for how to design database applications to achieve scalability for distributed real-time databases. By studying how virtual full replication supports scalability we can define usage conditions for when scalability is achievable. These findings should allow us to formulate guidelines for application designers.

References

1. N. Adly, M. Nagi, and J. Bacon. A hierarchical asynchronous replication protocol for large scale systems. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pp. 152–157, 1993.
2. R. Alonso, D. Barbara, and H. García-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3): 359–384, 1990.
3. S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Efring. DeeDS towards a distributed and active real-time database system. *ACM SIGMOD Record*, 25(1): 38–40, 1996.
4. S. Andler, J. Hansson, J. Mellin, J. Eriksson, and B. Efring. An overview of the DeeDS real-time database architecture. In *Proceedings of the 6th International Workshop on Parallel and Distributed Real-Time Systems*, 1998.
5. S. F. Andler, E. Örn Leifsson, and J. Mellin. Diskless real-time database recovery in distributed systems. In *Work in Progress at Real-Time Systems Symposium (RTSS'99)*, 1999.
6. J. Barreto. Information sharing in mobile networks: a survey on replication strategies. Technical Report RT/015/03, Instituto Superior Técnico, Lisboa, September 2003.
7. M. Brohede and S. F. Andler. Using distributed active real-time database functionality in information-fusion infrastructures. In *Real-Time in Sweden 2005 (RTiS2005)*, Skövde, Sweden, August 2005.
8. M. Brohede, S. F. Andler, and S. H. Son. Optimistic database-driven distributed real-time simulation (05F-SIW-031). In *Fall 2005 Simulation Interoperability Workshop (FallSIW 2005)*, Orlando, FL, September 2005.
9. A. Brunstrom, S. T. Leutenegger, and R. Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, pp. 395–402, 1995.
10. P. Cederqvist. Version management with CVS. <http://www.cvshome.org/docs/manual>, 2003.
11. S. Ceri, M. Houtsma, A. Keller, and P. Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(1): 225–246, 1995.
12. A. Datta, S. Mukherjee, and I. R. Viguier. Buffer management in real-time active database systems. *The Journal of Systems and Software*, 42(3): 227–246, 1998.
13. P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The WITAS unmanned aerial vehicle project. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00)*, pp. 747–755. IOS Press, Amsterdam, Netherlands, 2000.
14. W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4): 560–595, 1984.
15. D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 263–270, Seattle, WA, 1999.
16. D. Garlan. Software architecture: a roadmap. In *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, pp. 91–101, ACM Press, New York, NY, 2000.
17. K. Ghosh, K. Panesar, R. M. Fujimoto, and K. Schwan. Ports: a parallel, optimistic, real-time simulator. In *Parallel and Distributed Simulation (PADS'94)*. ACM Press, New York, NY, USA, 1994.
18. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Record*, 25(2): 173–182, 1996.
19. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, chapter 1. Morgan Kaufmann, San Mateo, CA, 1993.
20. S. Gustavsson and S. Andler. Continuous consistency management in distributed real-time databases with multiple writers of replicated data. In *Proceedings of the 13th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'05)*, Denver, CO, 2005.

21. J. Heidemann, A. Goel, and G. Popek. Defining and measuring conflicts in optimistic replication. Technical Report UCLA-CSD-950033, University of California, Los Angeles, September 1995.
22. J. Holliday, D. Agrawal, and A. El Abbadi. Partial database replication using epidemic communication. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems*, pp. 485–493, 2002.
23. J. Huang and J. A. Stankovic. Buffer management in real-time databases. Technical Report UM-CS-1990-065, University of Massachusetts, 1990.
24. R. Jauhari, M. J. Carey, and M. Livny. Priority-hints: an algorithm for priority-based buffer management. In *Proceedings of the 16th International Conference on Very Large Databases*, pp. 708–721. Morgan Kaufmann, San Francisco, CA, 1990.
25. D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3): 404–425, 1985.
26. F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice-Hall, Upper Saddle River, NJ, 1999.
27. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7): 558–565, 1978.
28. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1): 46–61, 1973.
29. G. Mathiason and S. F. Andler. Virtual full replication: achieving scalability in distributed real-time main-memory systems. In *Proceedings of the Work-in-Progress Session of the 15th Euromicro Conference on Real-Time Systems*, pp. 33–36, Porto, Portugal, July 2003.
30. G. Mathiason, S. F. Andler, and D. Jagszent. Virtual full replication by static segmentation for multiple properties of data objects. In *Proceedings of Real-Time in Sweden (RTiS 05)*, pp. 11–18, August 2005.
31. J. McManus and W. Bynum. Design and analysis techniques for concurrent blackboard systems. *IEEE Transactions on Systems, Man and Cybernetics*, 26(6): 669–680, 1996.
32. A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, 2002.
33. P. Nii. The blackboard model of problem solving. *AI Magazine*, 7(2): 38–53, 1986.
34. D. Nyström, A. Tesanovic, C. Norström, J. Hansson, and N.-E. Bänkestad. Data management issues in vehicle control systems: a case study. In *14th Euromicro Conference on Real-Time Systems*, pp. 249–256, June 2002.
35. S. Park, D. Lee, M. Lim, and C. Yu. Scalable data management using user-based caching and prefetching in distributed virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pp. 121–126, November 2001.
36. D. Parker, G. Popek, and G. Rudisin. Detection of mutual inconsistency in distributed systems. In *Proceedings of the 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1981.
37. Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, March 2005.
38. J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, June 1999.
39. B. Tatomir and L. Rothkrantz. Crisis management using mobile ad-hoc wireless networks. In *Proceedings of Information Systems for Crisis Response and Management ISCRAM 2005*, pp. 147–149, April 2005.
40. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, 1995.

41. N. Vidot, M. Cart, J. Ferri, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work*, Philadelphia, PA, pp. 171–180, 2000.
42. M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pp. 464–474, April 2000.
43. O. Wolfson and Y. Huang. Competitive analysis of caching in distributed databases. *IEEE Transactions on Parallel and Distributed Systems*, 9(4): 391–409, 1998.
44. O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2): 255–314, 1997.
45. O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Transactions on Database Systems*, 16(1): 181–205, 1991.
46. L. Wujuan and B. Veeravalli. An adaptive object allocation and replication algorithm in distributed databases. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems Workshop (DARES)*, pp. 132–137, 2003.
47. W. Zhou and W. Jia. The design and evaluation of a token-based independent update protocol. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, pp. 887–892, November 3–6, 1999. ACTA Press, Anaheim, CA.

VI

Formalisms, Methods, and Tools

30

State Space Abstractions for Time Petri Nets

30.1	Introduction	30-1
30.2	Time Petri Nets and Their State Space	30-2
	Time Petri Nets • States and Firing Schedules • Illustration • Some General Theorems	
30.3	State Space Abstractions Preserving Markings and Traces	30-5
	State Classes • Illustration • Properties Preserved by the SCG • Variations	
30.4	State Space Abstractions Preserving States and Traces	30-8
	Clock Domains • Construction of the SSCG • Checking Clock Domain Equivalence \equiv • Illustration • Properties Preserved • Variations	
30.5	Abstractions Preserving States and Branching Properties	30-13
	Preserving Branching Properties • Partitioning a Strong Class • Building the Atomic State Class Graph • Illustration • Properties Preserved	
30.6	Computing Experiments	30-15
30.7	Conclusion and Further Issues	30-16
	On State Space Abstractions • Extensions of TPNs	

Bernard Berthomieu

*Laboratoire d'Architecture
et d'Analyse des Systèmes du CNRS*

François Vernadat

*Laboratoire d'Architecture
et d'Analyse des Systèmes du CNRS*

30.1 Introduction

Since their introduction in Ref. 19, Time Petri nets (TPNs for short) have been widely used for the specification and verification of systems in which satisfiability of time constraints is essential like communication protocols [3,20], hardware components [27], or real-time systems [12,29,32].

TPNs extend Petri nets with temporal intervals associated with transitions, specifying firing delay ranges for the transitions. Assuming transition t became last enabled at time θ , and the endpoints of its time interval are α and β , then t cannot fire earlier than time $\theta + \alpha$ and must fire no later than $\theta + \beta$, unless disabled by firing some other transition. Firing a transition takes no time. Many other Petri-net-based models with time extensions have been proposed, but none reached the acceptance of TPNs. Availability of effective analysis methods, prompted by Ref. 5, certainly contributed to their widespread use, together with their ability to cope with a wide variety of modeling problems for real-time systems.

As with many formal models for real-time systems, the state spaces of TPNs are typically infinite. Model-checking TPNs first requires to produce finite abstractions for their state spaces, which are labeled transition

systems that preserve some classes of properties of the state space. This chapter overviews three such constructions, the classical one introduced in Ref. 5 and two more recent abstractions proposed in Ref. 8.

The abstraction of Ref. 5, further developed in Ref. 3, computes a graph of so-called state classes. State classes represent some infinite sets of states by a marking of the net and a polyhedron capturing the times at which the transitions enabled at that marking may fire. The method has been used in many works in various contexts and has been implemented in several tools. The state class graph produced is finite iff the *TPN* is bounded (admits a finite number of markings); it preserves markings, as well as the traces and complete traces of the state graph of the net, it is thus suitable for marking reachability analysis and verification of the formulas of linear time temporal logics like *LTL*.

The *state classes* construction of Ref. 5 is too coarse, however, for checking state reachability properties (a state associates a marking with firing time intervals for all enabled transitions). A finer abstraction allowing one to decide state reachability was proposed in Ref. 8, called *strong state classes*. But neither this latter abstraction nor that of Ref. 5 preserve branching properties of the state space, as expressed by formulas of branching time temporal logics like *CTL* or modal logics like *HML* or the μ -calculus. An abstraction preserving branching properties was first proposed in Ref. 31, called the *atomic state classes*. An equivalent but simpler construction was later introduced in Ref. 8, obtained from the previous strong state classes by a partition refinement process. This abstraction produces a graph which is bisimilar with the state graph of the net, hence preserving its branching properties.

The chapter is organized as follows. Section 30.2 reviews the terminology of *TPNs* and the definitions of their state spaces. The classical “state classes” construction, preserving markings and *LTL* properties, is explained in Section 30.3, and its properties are discussed. Section 30.4 explains the richer “strong state classes” abstraction that allows in addition to decide state reachability properties. Section 30.5 discusses preservation of branching properties and explains the “atomic state classes” construction. Some computing experiments are reported in Section 30.6. Finally, Section 30.7 discusses a number of related issues and recent results, including extensions of these methods to handle enriched classes of *TPNs*.

30.2 Time Petri Nets and Their State Space

30.2.1 Time Petri Nets

\mathbf{R}^+ and \mathbf{Q}^+ are the sets of nonnegative reals and rationals, respectively. Let \mathbf{I}^+ be the set of nonempty real intervals with nonnegative rational endpoints. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left endpoint and $\uparrow i$ its right endpoint (if i is bounded) or ∞ . For any $\theta \in \mathbf{R}^+$, $i \div \theta$ denotes the interval $\{x - \theta \mid x \in i \wedge x \geq \theta\}$.

Definition 30.1

A Time Petri net (or *TPN*) is a tuple $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$, in which $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0 \rangle$ is a Petri net, and $I_s : T \rightarrow \mathbf{I}^+$ is a function called the static interval function.

P is the set of *places*, T the set of *transitions*, $\mathbf{Pre}, \mathbf{Post} : T \rightarrow P \rightarrow \mathbf{N}^+$ are the *pre-* and *postcondition* functions, and $m^0 : P \rightarrow \mathbf{N}^+$ the *initial marking*. *TPNs* add to Petri nets the *static interval function* I_s , which associates a temporal interval $I_s(t) \in \mathbf{I}^+$ with every transition of the net. $Eft_s(t) = \downarrow I_s(t)$ and $Lft_s(t) = \uparrow I_s(t)$ are called the *static earliest firing time* and *static latest firing time* of t , respectively. A *TPN* is shown in Figure 30.1.

30.2.2 States and Firing Schedules

For any function f , let $f[E]$ stand for the restriction of f to its domain intersected with E . For any $k \in \mathbf{N}$ and $f, g : P \rightarrow \mathbf{N}^+$, $f \geq g$ stands for $(\forall p \in P)(f(p) \geq g(p))$ and $f + g$ (resp. $f - g$, resp. $k.f$) for the function mapping $f(p) + g(p)$ (resp. $f(p) - g(p)$, resp. $k.f(p)$) with every $p \in P$.

Transition t is *enabled at marking* m iff $m \geq \mathbf{Pre}(t)$. $\mathcal{E}(m)$ denotes the set of transitions enabled at m .

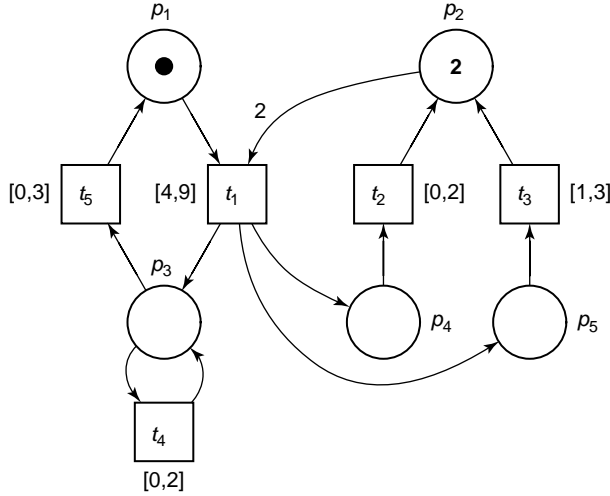


FIGURE 30.1 A Time Petri net.

Definition 30.2 (states)

A state of a TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$ is a pair $s = (m, I)$ in which $m : P \rightarrow \mathbf{N}^+$ is a marking and function $I : T \rightarrow \mathbf{I}^+$ associates a temporal interval with every transition enabled at m . The initial state is $s_0 = (m_0, I_0)$, where $I_0 = I_s[\mathcal{E}(m_0)]$ (I_s restricted to the transitions enabled at m_0).

When it is more convenient, the temporal information in states will be seen as firing domains, instead of interval functions. The *firing domain* of state (m, I) is the set of real vectors $\{\underline{\phi} \mid (\forall k)(\underline{\phi}_k \in I(k))\}$, with their components indexed by the enabled transitions.

Definition 30.3 (state transitions)

States may evolve by discrete or continuous transitions. Discrete transitions are the result of firing transitions of the net, continuous (or delay) transitions are the result of elapsing of time. These transitions are defined as follows, respectively. We have

$(m, I) \xrightarrow{t} (m', I')$ iff $t \in T$ and:

1. $m \geq \mathbf{Pre}(t)$
2. $0 \in I(t)$
3. $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
4. $(\forall k \in \mathcal{E}(m'))(I'(k) = \text{if } k \neq t \wedge m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k) \text{ then } I(k) \text{ else } I_s(k))$

$(m, I) \xrightarrow{\theta} (m, I')$ iff $\theta \in \mathbf{R}^+$ and:

5. $(\forall k \in \mathcal{E}(m))(\theta \leq \uparrow I(k))$
6. $(\forall k \in \mathcal{E}(m))(I'(k) = I(k) \div \theta)$

(3) is the standard marking transformation. From (4), the transitions not in conflict with t retain their firing interval, while those newly enabled are assigned their static intervals. A transition remaining enabled during its own firing is considered newly enabled. By (6), all firing intervals are shifted synchronously toward the origin as time elapses, and truncated to nonnegative times. (5) prevents time to elapse as soon as the latest firing time of some transition is reached. These conditions ensure that enabled transitions fire in their temporal interval, unless disabled by firing some other transition. Firing a transition takes no time.

The states as defined above associate exactly one firing interval with every transition enabled, whether that transition is multienabled or not (t is multienabled at m if there is some integer $k > 1$ such that $m \geq k \cdot \mathbf{Pre}(t)$). An alternative interpretation of multienabledness will be briefly discussed in Section 30.3.4.

The behavior of a *TPN* is characterized by its *state graph* defined as follows. The state graph *SG* captures all states resulting from delay (continuous) transitions and discrete transitions.

Definition 30.4

The state graph of a *TPN* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$ is the structure

$$SG = (S, \overset{a}{\rightsquigarrow}, s_0),$$

where $a \in T \cup \mathbf{R}^+$, $S = P^{\mathbf{N}^+}$ times $T^{\mathbf{I}^+}$, and $s_0 = (m_0, I_s[\mathcal{E}(m_0)])$.

Any sequence of delay transitions is equivalent to a single delay transition labeled with the sum of the labels of the transitions in the sequence. Further, for each state s , we have $s \overset{0}{\rightsquigarrow} s$. So, any finite sequence of transitions ending with a discrete transition is equivalent to a sequence alternating delay and discrete transitions, called a *firing schedule* or a *time transition sequence*. An alternative definition of the state space of a *TPN* is often used in the literature, only capturing the states reachable by some firing schedule. In Refs. 3, 5, and 8, for instance, the state graph is defined as $(S, \overset{t@t}{\rightarrow}, s_0)$, where $s \overset{t@t}{\rightarrow} s'$ is defined as $(\exists s'')(s \overset{t}{\rightsquigarrow} s'' \wedge s'' \overset{t}{\rightsquigarrow} s')$. This graph will be called the *discrete state graph*, or *DSG*, defined as follows:

Definition 30.5

The discrete state graph of a *TPN* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$ is the structure

$$DSG = (S, \overset{t}{\rightarrow}, s_0)$$

where $S = P^{\mathbf{N}^+} \times T^{\mathbf{I}^+}$, $s_0 = (m_0, I_s[\mathcal{E}(m_0)])$, and $s \overset{t}{\rightarrow} s' \Leftrightarrow (\exists \theta)(\exists s'')(s \overset{\theta}{\rightsquigarrow} s'' \wedge s'' \overset{t}{\rightsquigarrow} s')$.

The states of the *DSG* are those of the *SG* reachable by sequences of transitions ending with a discrete transition. Any state of the *DSG* is a state of the *SG*, and any state of the *SG* which is not in the *DSG* is reachable from some state of the *DSG* by a continuous transition.

In spite of its name, the *DSG* is a dense graph: states may have uncountable numbers of successors by $\overset{t}{\rightarrow}$. Finitely representing state spaces involve the grouping of some sets of states. Several groupings can be defined depending on the properties of the state space one would like to preserve, three essential constructions are presented in Sections 30.3 through 30.5. As will be seen, these state space abstractions are abstractions of the *DSG* rather than the *SG*. We will explain the consequences of this approximation when needed.

30.2.3 Illustration

A state $s = (m, I)$ is represented by a pair (m, D) , in which m is a marking and D a firing domain. If t is enabled at m , projection t of D is the firing interval $I(t)$ associated with t in s . Firing domains are described by systems of linear inequalities with one variable per transition enabled.

This way, the initial state $s_0 = (m_0, D_0)$ of the net shown in Figure 30.1 is represented by:

$$\begin{aligned} m_0 : & p_1, p_2 * 2 \\ D_0 : & 4 \leq \phi_{t_1} \leq 9 \end{aligned}$$

Waiting $\theta_1 \in [4, 9]$ units of time then firing t_1 , leads to state $s_1 = (m_1, D_1)$, is described as follows:

$$\begin{aligned} m_1 : & p_3, p_4, p_5 \\ D_1 : & 0 \leq \phi_{t_2} \leq 2 \\ & 1 \leq \phi_{t_3} \leq 3 \\ & 0 \leq \phi_{t_4} \leq 2 \\ & 0 \leq \phi_{t_5} \leq 3 \end{aligned}$$

Firing t_2 from s_1 after waiting $\theta_2 \in [0, 2]$ units of time, leads to state $s_2 = (m_2, D_2)$, is described by

$$\begin{aligned} m_2 : & p_2, p_3, p_5 \\ D_2 : & \max(0, 1 - \theta_2) \leq \phi_{t_3} \leq 3 - \theta_2 \\ & 0 \leq \phi_{t_4} \leq 2 - \theta_2 \\ & 0 \leq \phi_{t_5} \leq 3 - \theta_2 \end{aligned}$$

State s_1 admits an infinity of successor states by t_2 , one for each possible θ_2 .

30.2.4 Some General Theorems

A *TPN* is *bounded* if the marking of each place is bounded by some integer; boundedness implies finiteness of the marking reachability set. A state is *reachable* if it is a node in the SG of the net, a marking is *reachable* if some state with that marking is reachable, and a net is *live* if, from any reachable state s and any transition t , some firing schedule firing t is firable from s . It is shown in Ref. 16 that the marking reachability problem for *TPNs* is undecidable; undecidability of the other problems follows.

Theorem 30.1

For Time Petri nets, the problems of marking reachability, of state reachability, of boundedness, and of liveness are undecidable.

In some subclasses of *TPNs*, the reachability relation \xrightarrow{t} is finitely branching. These *TPNs* admit finite discrete state spaces iff they are bounded. The following theorem is easily proven by induction.

Theorem 30.2

The discrete state space of a bounded Time Petri net is finite if either

- (i) *The static firing intervals of all transitions are unbounded*
- or*
- (ii) *The static firing intervals of all transitions are punctual*

In addition, for case (i), and for case (ii) when all static intervals are equal, their discrete state spaces are isomorphic with the marking space of the underlying Petri net.

Theorem 30.2 allows one to interpret Petri nets as *TPNs* in various ways. The most frequent interpretation is that Petri nets are *TPNs* in which all transitions bear static interval $[0, \infty[$.

30.3 State Space Abstractions Preserving Markings and Traces

30.3.1 State Classes

The state space of a *TPN* may be infinite for two reasons: on the one hand because some state may admit an infinity of successor states, as already seen, on the other because the net may be unbounded. The latter case will be discussed in Section 30.3.4. For handling the former case, some particular sets of states will be grouped into *state classes*. Several groupings are possible, we review in this section the grouping method introduced in Ref. 5 and further developed in Ref. 3.

Definition 30.6

For each $\sigma \in T^$, C_σ is inductively defined by: $C_\epsilon = \{s_0\}$ and $C_{\sigma,t} = \{s' \mid (\exists s \in C_\sigma)(s \xrightarrow{t} s')\}$.*

C_σ is the set of states reached in the DSG by sequence σ , every state of the DSGs in some C_σ . For each such set of states C_σ , let us define its marking $\mathcal{M}(C_\sigma)$ as the marking of any of the states it contains (all states in C_σ necessarily bear the same marking), and its firing domain $\mathcal{F}(C_\sigma)$ as the union of the firing domains of all the states it contains. Finally, let us denote by \cong the relation satisfied by two such sets of states when they have same marking and firing domains:

Definition 30.7

$C_\sigma \cong C_{\sigma'} \Leftrightarrow \mathcal{M}(C_\sigma) = \mathcal{M}(C_{\sigma'}) \wedge \mathcal{F}(C_\sigma) = \mathcal{F}(C_{\sigma'})$.

The state class construction of Ref. 5 stems from the following observation.

Theorem 30.3 (Ref. 5)

If $C_\sigma \cong C_{\sigma'}$, then any firing schedule firable from C_σ is firable from $C_{\sigma'}$, and conversely.

The state classes of Ref. 5 are the above sets C_σ , for all firable σ , considered modulo equivalence \cong . The initial class C_ϵ holds the sole initial state. The set of classes is equipped with the transition relation: $C_\sigma \xrightarrow{t} X \Leftrightarrow C_{\sigma.t} \cong X$. The graph of state classes (SCG for short) is obtained by the following algorithm. State classes are represented by pairs (m, D) in which m is a marking and D a firing domain described by a system of linear inequalities $W\phi \leq \underline{w}$. Variables ϕ are bijectively associated with the transitions enabled at m . We have $(m, D) \cong (m', D')$ iff $m = m'$ and the systems describing D and D' have equal solution sets.

Algorithm 30.1 (computing the SCG)

With each firable sequence σ , a pair L_σ can be computed as follows. Compute the smallest set C of pairs containing L_ϵ and such that, whenever $L_\sigma \in C$ and $\sigma.t$ is firable, we have $X \cong L_{\sigma.t}$ for some $X \in C$.

- The initial pair is $L_\epsilon = (m_0, \{Eft_s(t) \leq \underline{\phi}_t \leq Lft_s(t) \mid m_0 \geq \mathbf{Pre}(t)\})$
 - If σ is firable and $L_\sigma = (m, D)$, then $\sigma.t$ is firable if and only if:
 - (i) $m \geq \mathbf{Pre}(t)$ (t is enabled at m) and;
 - (ii) The system $D \wedge \{\underline{\phi}_t \leq \underline{\phi}_i \mid i \neq t \wedge m \geq \mathbf{Pre}(i)\}$ is consistent.
 - If $\sigma.t$ is firable, then $L_{\sigma.t} = (m', D')$ is obtained from $L_\sigma = (m, D)$ as follows:
 - $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$,
 - D' obtained by:
 - (a) The above firability constraints (ii) for t from L_σ are added to system D ;
 - (b) For each k enabled at m' , a new variable $\underline{\phi}'_k$ is introduced, subject to:

$$\underline{\phi}'_k = \phi_k - \phi_t, \text{ if } k \neq t \text{ et } m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k),$$

$$Eft_s(k) \leq \underline{\phi}'_k \leq Lft_s(k), \text{ otherwise;}$$
 - (c) The variables $\underline{\phi}$ are eliminated.
-

The pair L_σ computed in Algorithm 30.1 represents the equivalence class by \cong of set C_σ . It is proven in Ref. 5 that the number of firing domains one can compute with Algorithm 30.1 is finite, whether the net is bounded or not. It follows that the graph of state classes of a TPN is finite iff the net is bounded.

The systems representing firing domains D are difference systems, checking equivalence \cong can be done by putting these systems into canonical form, which can be done with complexity $O(n^3)$ in time and $O(n^2)$ in space, where n is the number of variables, using, for example, the Floyd/Warshall algorithm for computing all-pair shortest paths. Implementations of Algorithm 30.1 are discussed, for example, in Refs. 3, 5, 10, and 29 for TPNs and in Ref. 26 for a model closely related to TPNs. It was observed in Ref. 26 that, when applying Algorithm 30.1 to state classes in canonical forms, canonization could be incrementally done in time complexity $O(n^2)$; the implementations described in Refs. 10 and 29 for TPNs follow from the same observation.

30.3.2 Illustration

As an illustration, let us build some state classes for the net shown in Figure 30.1. The initial class c_0 is described exactly like the initial state s_0 in Section 30.2.3. Firing t_1 from class c_0 leads to a class c_1 described exactly like state s_1 in Section 30.2.3. Firing t_2 from c_1 leads to class $c_2 = (m_2, D_2)$, with $m_2 = (p_2, p_3, p_5)$ and D_2 obtained in three steps:

- (a) First, the firability constraints for t_2 from c_1 are added to system D_1 , these are

$$\phi_{t_2} \leq \phi_{t_3}$$

$$\phi_{t_2} \leq \phi_{t_4}$$

$$\phi_{t_2} \leq \phi_{t_5}$$

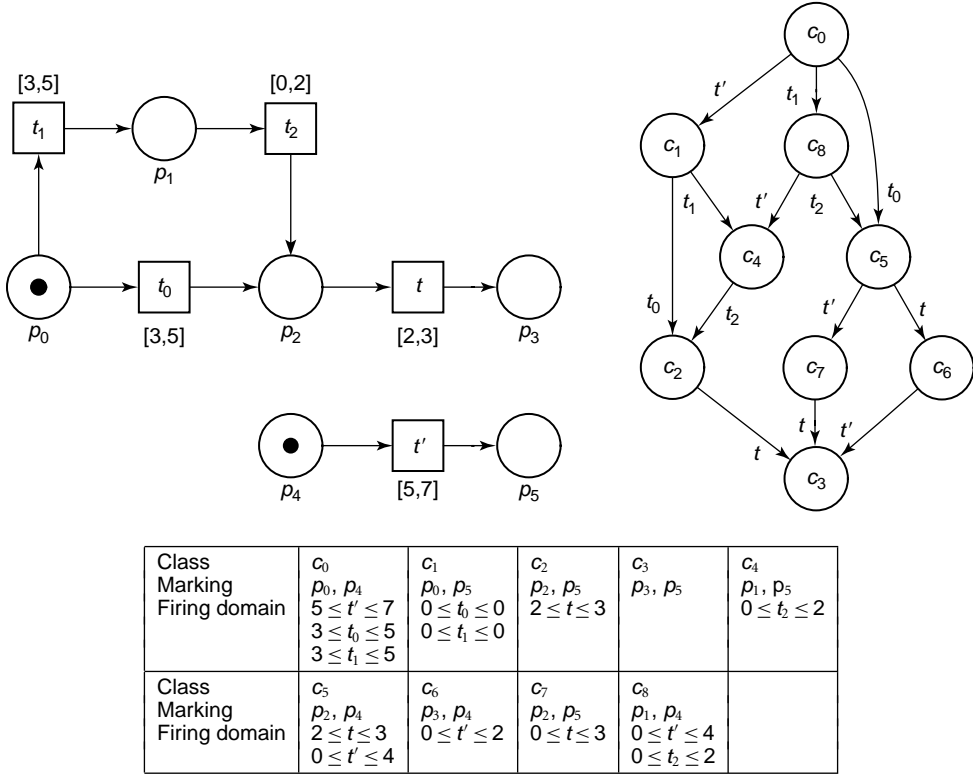


FIGURE 30.2 A TPN and its SCG. Variable ϕ_t is written as t .

(b) No transition is newly enabled by firing t_2 , and transitions t_3, t_4 , and t_5 are not in conflict with t_2 .

We thus simply add the equations $\phi'_{t_i} = \phi_{t_i} - \phi_{t_2}$, for $i \in \{3, 4, 5\}$

(c) Variables ϕ_{t_i} are eliminated (e.g., by the classical Fourier/Motzkin elimination), D_2 is the system:

$$\begin{aligned}
 0 &\leq \phi'_{t_3} \leq 3 & \phi'_{t_4} - \phi'_{t_3} &\leq 1 \\
 0 &\leq \phi'_{t_4} \leq 2 & \phi'_{t_5} - \phi'_{t_3} &\leq 2 \\
 0 &\leq \phi'_{t_5} \leq 3 & &
 \end{aligned}$$

The graph of state classes of the net shown in Figure 30.1 admits 12 classes and 29 transitions. Figure 30.2 shows another TPN and its SCG. This example will be used to compare the SCG construction with the constructions presented in the forthcoming sections.

30.3.3 Properties Preserved by the SCG

The sets C_σ defined in Section 30.3.1 may be presented as a tree rooted at C_ϵ . The SCG is the folding of this tree by relation \cong . By Theorem 30.3, and the fact that all states in any C_σ have the same marking, we have that the DSG and the SCG are complete-trace equivalent. Hence:

Theorem 30.4

For any TPN, its DSG and SCG have the same markings and obey the same LTL properties.

Interpreting LTL formulas over timed systems requires to consider time divergence: the case of states at which time can continuously elapse without any transition being taken. Such states are either (1) states at which no transition is enabled (deadlock states) or (2) states at which all enabled transitions have unbounded firing intervals.

For case 1, existence of deadlocks in the *DSG* or the state graph *SG* is preserved by the *SCG* since the deadlocked property for a state of a *TPN* only depends on its marking, and all states in a class have the same marking. Existence of deadlock states implies existence of classes without successor classes.

For case 2, existence of such states cannot be told from the structure of the *SCG* or *DSG*, while in the state graph *SG*, for any such diverging state s , there are always some s' , $d \geq 0$, and $d' > 0$ such that $s \xrightarrow{d} s' \xrightarrow{d'} s'$. To make this second kind of divergence observable from the structure of the *SCG*, a solution is to add silent transitions $c \rightarrow c$ at all classes $c = (m, I)$ such that all transitions enabled at m have unbounded static intervals. An alternative is to assume for *TPNs* the “fairness” hypothesis that no transition can be continuously enabled without being taken, in which case the *SCG* as built by Algorithm 30.1 is adequate for *LTL* model checking.

30.3.4 Variations

Checking the boundedness property on the fly: It was recalled in Section 30.2.4 that boundedness is undecidable for *TPNs*. There are, however, a number of decidable sufficient conditions for this property, one, for instance, is that the underlying Petri net is bounded. Some such sufficient conditions can be checked on the fly while building the *SCG* [3,5], adapting Algorithm 30.1. Structural analysis offers alternative sufficient conditions. Also, there are useful subclasses of *TPNs* bounded “by construction.”

Handling multienabledness: A transition t is multienabled at m if there is some $k > 1$ such that $m \geq k \cdot \mathbf{Pre}(t)$. The states defined in Section 30.2 associate exactly one interval with every transition enabled, whether that transition is multienabled or not; the different enabling times for t being collectively represented by the latest. In some applications, it might be relevant to distinguish the different enabling instances of a multienabled transition. These aspects are addressed in Ref. 2, where the notion of state of Section 30.2 is extended so that a transition which is enabled k times in some state is associated with k firing intervals. The different enabling instances may be assumed to be independent, or ordered according to their creation dates. Algorithm 30.1 can be adapted to support that extended notion of state and the details can be found in Ref. 2.

Preserving markings but not firing sequences: A state space abstraction preserving only the reachable markings is easily obtained by a variant of Algorithm 30.1. The idea is to identify a class with any class including it. That is, one computes a set C of classes such that, whenever $L_\sigma \in C$ and $\sigma.t$ is fireable, we have $L_{\sigma.t} \lesssim X$ for some $X \in C$ (rather than $L_{\sigma.t} \cong X$), where $(m, D) \lesssim (m, D')$ iff domain D' includes D . Intuitively, if such class X exists, then any schedule fireable from a state captured by $L_{\sigma.t}$ is fireable from a state captured by X , thus one will not find new markings by storing class $L_{\sigma.t}$. Obviously, this construction no more preserves the firing sequences of the *SG* and thus its *LTL* properties, it only preserves markings, but the graph built may be smaller than the *SCG* by a large factor.

30.4 State Space Abstractions Preserving States and Traces

Two sets C_σ and $C_{\sigma'}$ (see Definition 30.6) may be equivalent by \cong while having different contents in terms of states. The notation used for state classes in Algorithm 30.1 canonically identifies equivalence classes by \cong , but not the sets C_σ themselves. Reachability of a state cannot be proved or disproved from the *SCG* classes, in general, the *SCG* is too coarse an abstraction for that purpose.

The *strong state classes* (also called *state zones* by some authors) reviewed in this section exactly coincide with the sets of states C_σ of Definition 30.6. The graph of strong state classes preserves the *LTL* properties of the *DSG* of the net, but also its states, in a sense we will make precise. For building the strong state class graph (*SSCG*), we first need a means of canonically representing the sets C_σ . Clock domains serve this purpose.

30.4.1 Clock Domains

With every reachable state, one may associate a *clock function* γ , defined as follows: with each transition enabled at the state, function γ associates the time elapsed since it was last enabled. The clock function may also be seen as a vector $\underline{\gamma}$, indexed over the transitions enabled.

In the SSCG construction, a class is represented by a marking and a clock system. Let $\langle Q \rangle$ denote the solution set of inequation system Q . The set of states denoted by a marking m and a clock system $Q = \{G\underline{\gamma} \leq \underline{g}\}$ is the set $\{(m, \Phi(\underline{\gamma})) \mid \underline{\gamma} \in \langle Q \rangle\}$, where firing domain $\Phi(\underline{\gamma})$ is the solution set in $\underline{\phi}$ of:

$$\underline{0} \leq \underline{\phi}, \underline{e} \leq \underline{\phi} + \underline{\gamma} \leq \underline{l},$$

where $\underline{e}_k = Eft_s(k)$ and $\underline{l}_k = Lft_s(k)$

Each clock vector denotes a state, but different clock vectors may denote the same state, and clock systems with different solution sets may denote the same set of states. For this reason we introduce equivalence \equiv :

Definition 30.8

Given two pairs $c = (m, Q = \{G\underline{\gamma} \leq \underline{g}\})$ and $c' = (m', Q' = \{G'\underline{\gamma}' \leq \underline{g}'\})$, we write $c \equiv c'$ iff $m = m'$ and clock systems Q and Q' denote the same sets of states.

Equivalence \equiv is clearly decidable, methods for checking \equiv will be discussed in Section 30.4.3. Assuming such a method is available, we now give an algorithm for building the graph of strong state classes.

30.4.2 Construction of the SSCG

Strong classes are represented by pairs (m, Q) , where m is a marking and Q a clock system $G\underline{\gamma} \leq \underline{g}$. Clock variables $\underline{\gamma}$ are bijectively associated with the transitions enabled at m .

Algorithm 30.2 (computing strong state classes)

For each firable firing sequence σ , a pair R_σ can be computed as follows. Compute the smallest set C including R_ϵ and such that, whenever $R_\sigma \in C$ and $\sigma.t$ is firable, we have $X \equiv R_{\sigma.t}$ for some $X \in C$.

- The initial pair is $R_\epsilon = (m_0, \{0 \leq \underline{\gamma}_t \leq 0 \mid \mathbf{Pre}(t) \leq m_0\})$
 - If σ is firable and $R_\sigma = (m, Q)$, then $\sigma.t$ is firable iff:
 - (i) t is enabled at m , that is: $m \geq \mathbf{Pre}(t)$
 - (ii) Q augmented with the following constraints is consistent:

$$0 \leq \theta, Eft_s(t) - \underline{\gamma}_t \leq \theta, \{\theta \leq Lft_s(i) - \underline{\gamma}_i \mid m \geq \mathbf{Pre}(i)\}$$
 - If $\sigma.t$ is firable, then $R_{\sigma.t} = (m', Q')$ is computed from $R_\sigma = (m, Q)$ by:
 - $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
 - Q' obtained by:
 1. A new variable is introduced, θ , constrained by condition (ii);
 2. For each i enabled at m' , a new variable $\underline{\gamma}'_i$ is introduced, obeying:

$$\underline{\gamma}'_i = \underline{\gamma}_i + \theta, \text{ if } i \neq t \text{ and } m - \mathbf{Pre}(t) \geq \mathbf{Pre}(i)$$

$$0 \leq \underline{\gamma}'_i \leq 0, \text{ otherwise}$$
 - 3. Variables $\underline{\gamma}$ and θ are eliminated.
-

The temporary variable θ stands for the possible firing times of transition t . There is an arc labeled as t between R_σ and c iff $c \equiv R_{\sigma.t}$. R_σ denotes the set C_σ of states reachable from s_0 by firing schedules over σ .

As for the firing domains computed by Algorithm 30.1, the set of clock systems Q with distinct solution sets one can build by Algorithm 30.2 is finite, whether the *TPN* is bounded or not, so the SSCG is finite iff the *TPN* is bounded. As for Algorithm 30.1, one may easily add to Algorithm 30.2 on the fly boundedness

checks (see Section 30.3.4). Like the firing domains in the state classes of the SCG, the clock domains of the SSCG computed by Algorithm 30.2 are difference systems, for which canonical forms can be computed in polynomial time and space.

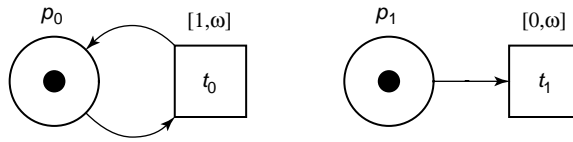
30.4.3 Checking Clock Domain Equivalence \equiv

First, equivalence \equiv is easily decided in a particular case:

Theorem 30.5 (\equiv , bounded static interval's case) (Ref. 8)

Consider two strong state classes $c = (m, Q = \{G\underline{\gamma} \leq \underline{g}\})$ and $c' = (m', Q' = \{G'\underline{\gamma}' \leq \underline{g}'\})$. Then if all transitions enabled at m or m' have bounded static intervals, we have $c \equiv c'$ iff $m = m' \wedge \langle Q \rangle = \langle Q' \rangle$.

In the general case, infinitely many distinct clock systems may denote the same set of states. An example of a net for which Algorithm 30.2 would not terminate with \equiv implemented as in Theorem 30.5 is the following:



Its initial strong class is $R_e = (m_0 = \{p_0, p_1\}, \{0 = \underline{\gamma}_{t_0}, 0 = \underline{\gamma}_{t_1}\})$. Using Algorithm 30.2, firing k ($k > 0$) times t_0 from it leads to $R_{t^k} = (\{p_0, p_1\}, \{0 = \underline{\gamma}_{t_0}, k \leq \underline{\gamma}_{t_1}\})$. The clock systems in these classes have different solution sets, but it should be observed that they all denote the same set of states: the singleton set containing the initial state. According to Definition 30.8, R_e and all R_{t^k} are equivalent by \equiv .

Since clock domains can be ordered by inclusion, a possible canonical representant for all clock domains denoting some set of states is the largest such domain. For our above example, this set can be described by the clock system $\{0 = \underline{\gamma}_{t_0}, 0 \leq \underline{\gamma}_{t_1}\}$. Such “largest” clock domains can be computed as follows:

Definition 30.9 (relaxation of clock systems) (Ref. 8)

Let (m, Q) represent some strong state class, and E^∞ be the transitions enabled at m with unbounded static intervals. The relaxation of Q is the disjunction of systems $\widehat{Q} = \bigvee \{Q_e | e \subseteq E^\infty\}$, with Q_e obtained by:

(i) First, Q is augmented with constraints:

$$\begin{aligned} \gamma_t &< Eft_s(t), \forall t \in e \\ \gamma_t &\geq Eft_s(t), \forall t \in E^\infty \setminus e \end{aligned}$$

(ii) Then all variables $t \in E^\infty \setminus e$ are eliminated

(iii) Finally, add constraints $\gamma_t \geq Eft_s(t), \forall t \in E^\infty \setminus e$.

Clock space Q is recursively split according to whether $\underline{\gamma}_k \geq Eft_s(k)$ or not, with $k \in E^\infty$. Then, in the half space in which $\underline{\gamma}_k \geq Eft_s(k)$, the upper bound of $\underline{\gamma}_k$ is relaxed. The solution set $\langle \widehat{Q} \rangle$ of \widehat{Q} is the union of the solution sets of its components. Relation \equiv can now be decided in the general case:

Theorem 30.6 (\equiv in arbitrary TPNs) (Ref. 8)

Let $c = (m, Q = \{G\underline{\gamma} \leq \underline{g}\})$ and $c' = (m', Q' = \{G'\underline{\gamma}' \leq \underline{g}'\})$. Then $c \equiv c' \Leftrightarrow m = m' \wedge \langle \widehat{Q} \rangle = \langle \widehat{Q}' \rangle$.

It remains to show how to update the construction algorithm for the SSCG to handle arbitrary static intervals. Two solutions were proposed in Ref. 8:

- The first is to implement \equiv in Algorithm 30.2 as in Theorem 30.6. This can be done by maintaining two representations for clock systems: that computed by Algorithm 30.2 for computing successor classes, and its relaxation used for comparing the class with those classes already computed.

- The second is to implement \equiv in Algorithm 30.2 as in Theorem 30.5 throughout, but integrate relaxation into Algorithm 30.2 as follows: after computation of $R_{\sigma,t} = (m', Q')$, \hat{Q}' is computed, and R_σ is assigned as many successor classes by t as \hat{Q}' has components, each pairing m' with a single component of \hat{Q}' .

Both solutions are adequate when there are few concurrent transitions with unbounded static intervals, but the second may produce much larger graphs when there are many of them. A third alternative was recently proposed in Ref. 15, based on what can be seen as a specialization of the above relaxation operation, called *normalization*. Normalization preserves convexity of clock systems and is computationally cheaper than relaxation. Geometrically, it computes the largest clock set that preserves the state contents of a class and that can be described by a difference system. Normalization is applied to all classes computed by Algorithm 30.2, after canonization. This solution yields the same graph as the above first alternative.

Definition 30.10 (normalization of clock systems) (Ref. 15)

Let (m, Q) represent some strong state class, and E^∞ be the set of transitions enabled at m that have unbounded static interval. The normalization of system Q is the system \tilde{Q} obtained from Q as follows (Q is assumed to be in the canonical form, all of its constraints are tight):

For each $t \in E^\infty$

- if $\gamma_t \geq \text{Eft}_s(t)$ is redundant, then relax variable γ_t (eliminate it then add constraint $\gamma_t \geq \text{Eft}_s(t)$)
- else, and if $\gamma_t \geq \text{Eft}_s(t)$ is satisfiable, remove any constraint $\gamma_t \leq c$ and, in addition, any constraint $\gamma_t - \gamma_{t'} \leq c''$ (or $\gamma_t - \gamma_{t'} < c''$) such that $\gamma_t \geq c'$ (or $\gamma_t > c'$) and $c' + c'' \geq \text{Eft}_s(t)$.

Theorem 30.7 (\equiv in arbitrary TPNs) (Ref. 15)

Let $c = (m, Q = \{G\underline{\gamma} \leq \underline{g}\})$ and $c' = (m', Q' = \{G'\underline{\gamma}' \leq \underline{g}'\})$. Then $c \equiv c' \Leftrightarrow m = m' \wedge \langle \tilde{Q} \rangle = \langle \tilde{Q}' \rangle$.

30.4.4 Illustration

As an illustration for the construction of the SSCG, let us build some state classes for the net shown in Figure 30.1. The initial class c_0 is the pair $(m_0, K_0 = \{\gamma_{t_1} = 0\})$.

t_1 is the sole transition enabled at m_0 and all transitions enabled after firing t_1 are newly enabled, firing t_1 from c_0 leads to $c_1 = (m_1 = (p_3, p_4, p_5), K_1 = \{\gamma_{t_2} = 0, \gamma_{t_3} = 0, \gamma_{t_4} = 0, \gamma_{t_5} = 0\})$ (renaming γ'_{t_i} into γ_{t_i}):

Firing t_2 from c_1 leads to a class $c_2 = (m_2, D_2)$, with $m_2 = (p_2, p_3, p_5)$ and K_2 obtained in three steps:

- (a) First, the firability constraints for t_2 from c_1 are added to system K_1 , these are the following:

$$\begin{aligned} 0 &\leq \theta \\ 0 &\leq \gamma_{t_2} + \theta \leq 2 \\ 1 &\leq \gamma_{t_3} + \theta \leq 3 \\ 0 &\leq \gamma_{t_4} + \theta \leq 2 \\ 0 &\leq \gamma_{t_5} + \theta \leq 3 \end{aligned}$$

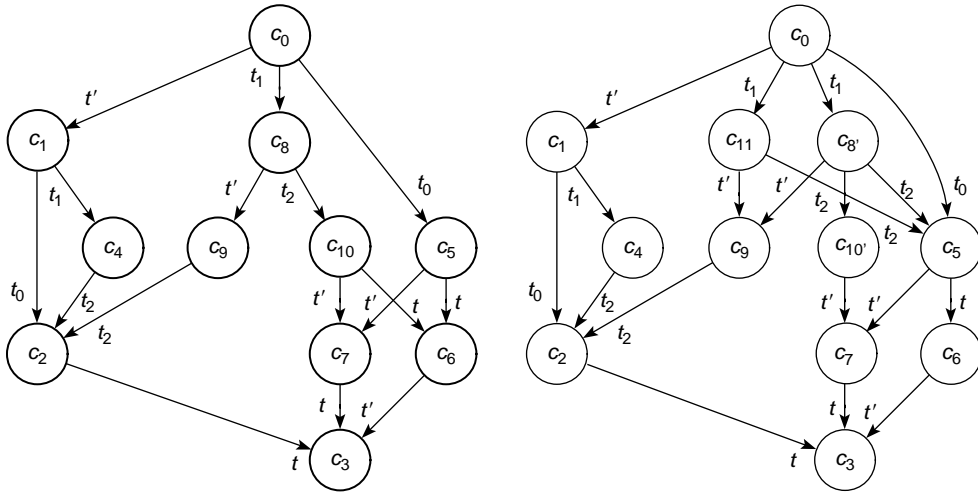
- (b) No transition is newly enabled by firing t_2 , and transitions t_3, t_4 , and t_5 are not in conflict with t_2 .

We thus simply add the equations $\gamma'_{t_i} = \gamma_{t_i} + \theta$, for $i \in \{3, 4, 5\}$

- (c) Finally, variables θ and γ_{t_i} are eliminated, yielding system K_2 :

$$\begin{aligned} 0 &\leq \gamma'_{t_3} \leq 2 & \gamma'_{t_3} &= \gamma'_{t_4} \\ 0 &\leq \gamma'_{t_4} \leq 2 & \gamma'_{t_3} &= \gamma'_{t_5} \\ 0 &\leq \gamma'_{t_5} \leq 2 & \gamma'_{t_4} &= \gamma'_{t_5} \end{aligned}$$

The graph of strong state classes of the net shown in Figure 30.1 admits 18 classes and 48 transitions. The SSCG of the TPN in Figure 30.2 admits 11 classes and 16 transitions, which is represented in Figure 30.3 (upper left). Compared to the SCG of the same net, represented in Figure 30.2, one will notice that the state sets $C_{t'.t_1}$ and $C_{t_1.t'}$ are distinguished in the SSCG (represented by strong classes c_4 and c_9 , respectively), while they were equivalent by \cong in the SCG, and similarly for the sets $C_{t_1.t_2}$ (c_{10}) and C_{t_0} (c_5).



Class	c_0	c_1	c_2	c_3
Marking	p_0, p_4	p_0, p_5	p_2, p_5	p_3, p_5
Clock domain	$0 \leq t' \leq 0$ $0 \leq t_0 \leq 0$ $0 \leq t_1 \leq 0$	$5 \leq t_0 \leq 5$ $5 \leq t_1 \leq 5$	$0 \leq t \leq 0$	
Class	c_4	c_5	c_6	c_7
Marking	p_1, p_5	p_2, p_4	p_3, p_4	p_2, p_5
Clock domain	$0 \leq t_2 \leq 0$	$0 \leq t \leq 0$ $3 \leq t' \leq 5$	$5 \leq t' \leq 7$	$0 \leq t \leq 3$
Class	c_8	c_9	c_{10}	c_{10}'
Marking	p_1, p_4	p_1, p_5	p_2, p_4	p_1, p_4
Clock domain	$3 \leq t' \leq 5$ $0 \leq t_2 \leq 0$	$0 \leq t_2 \leq 2$	$0 \leq t \leq 0$ $3 \leq t' \leq 7$	$3 < t' \leq 5$ $0 \leq t_2 \leq 0$
Class	c_{10}'	c_{11}		
Marking	p_2, p_4	p_1, p_4		
Clock domain	$0 \leq t \leq 0$ $5 < t' \leq 7$	$3 \leq t' \leq 3$ $0 \leq t_2 \leq 0$		

FIGURE 30.3 SSCG and ASFG of the net shown in Figure 30.2. Variable γ_i is denoted as t .

30.4.5 Properties Preserved

Theorem 30.4 also holds for the SSCG for the same reasons. The SSCG preserves the properties of the DSG one can express in linear time temporal logics like *LTL*. In addition, the SSCG allows one to check reachability of some state of the DSG, as expressed by the following theorem.

Theorem 30.8

Given a state $s = (m, I)$, one can always compute a clock vector $v = \underline{\gamma}$ such that, for any transition k enabled at m : $I(k) = I_s(k) \dot{-} \underline{\gamma}_k$. If there are several such clock vectors, choose any one. Then a state s belongs to the discrete state graph DSG iff, for some class (m', Q) of the SSCG, we have $m = m'$ and $v \in \hat{Q}$, where \hat{Q} is the relaxation of system Q , as computed in Section 30.4.3.

Theorem 30.9

For any TPN, its DSG and SSCG have the same states and obey the same *LTL* properties.

Theorem 30.8 is easily adapted to check membership of a state s in the state graph SG. A state $s = (m, I)$ belongs to the SG iff it is reachable from some $s' = (m', I')$ of the DSG by a continuous transition. Hence,

$s = (m, I)$ belongs to SG iff, for some class (m', Q) of the $SSCG$, we have $m = m'$ and $v - d \in \widehat{Q}$ for some $d \in \mathbf{R}^+$ (v is the clock vector computed in Theorem 30.8). Finally, as for the SCG in Section 30.3.3, the $SSCG$ can be enriched to preserve temporal divergence resulting from transitions with unbounded firing intervals.

30.4.6 Variations

As for the SCG (see Section 30.3.4), on the fly detection of sufficient conditions for the boundedness property can easily be added to Algorithm 30.2.

Finally, as for the SCG again, and as observed in Refs. 9 and 15, one can build a variant of the $SSCG$ preserving states, but not traces, by not memorizing a class when its clock domain is included in the clock domain of some already computed class. In most cases, this construction yields graphs smaller than the $SSCG$.

30.5 Abstractions Preserving States and Branching Properties

30.5.1 Preserving Branching Properties

The branching properties are those expressible in branching time temporal logics like CTL , or modal logics like HML or the μ -calculus. Neither the SCG nor the $SSCG$ preserves these properties. Consider, for instance, the net represented in Figure 30.2 with its SCG . An example of HML formula whose truth value is not preserved by this graph is $\langle t_1 \rangle \langle t_2 \rangle [t]F$, expressing that it is possible to reach by a schedule of support $t_1.t_2$ a state from which t is not firable. This property is false on the SCG but true on the SG of the net since, after firing schedule $5.t_1.2.t_2$, one reaches the state $(\{p_2, p_4\}, I)$, with $I(t) = [2, 3]$ and $I(t') = [0, 0]$, from which t is clearly not firable (t' must fire first).

Bisimilarity is known to preserve branching properties, we thus seek a state space abstraction bisimilar with the SG of the TPN . A first such construction for $TPNs$ was proposed in Ref. 31, called the *atomic state class graph*, for the subclass of $TPNs$ in which all transitions have bounded static intervals. A specialization of this construction, only preserving formulas of the $ACTL$ logic, was proposed in Ref. 22. Rather than these constructions, we will recall that of Ref. 8, which is applicable to any TPN and typically yields smaller graphs. The graph built is obtained by a refinement of the $SSCG$ of Section 30.4, it is also called the *atomic state class graph* ($ASCG$ for short) since it serves the same purposes as that in Ref. 31.

Bisimulations can be computed with an algorithm initially aimed at solving the *relational coarsest partition* problem [21]. Let \rightarrow be a binary relation over a finite set U , and for any $S \subseteq U$, let $S^{-1} = \{x | (\exists y \in S)(x \rightarrow y)\}$. A partition P of U is *stable* if, for any pair (A, B) of blocks of P , either $A \subseteq B^{-1}$ or $A \cap B^{-1} = \emptyset$ (A is said *stable wrt* B). Computing a bisimulation, starting from an initial partition P of states, is computing a stable refinement Q of P [17,1]. An algorithm is the following:

```
Initially :  $Q = P$ 
while there exists  $A, B \in Q$  such that  $\emptyset \subsetneq A \cap B^{-1} \subsetneq A$  do
  replace  $A$  by  $A_1 = A \cap B^{-1}$  and  $A_2 = A \setminus B^{-1}$  in  $Q$ 
```

In our case, a suitable starting partition is the graph of strong state classes of Section 30.4, or, alternatively [9], its variant only preserving states, discussed in Section 30.4.6. Computing the $ASCG$ is computing a stable refinement of this graph. Our algorithm is based on the above, with two differences implying that our partition will not generally be the coarsest possible: (1) The states, typically infinite in number, are handled as convex sets, but the coarsest bisimulation does not necessarily yield convex blocks. (2) Our refinement process starts from a cover of the set of states, rather than a partition (states may belong to several classes).

30.5.2 Partitioning a Strong Class

Let $c = (m, Q)$ be some class and assume $c \xrightarrow{t} c'$ with $c' = (m', Q')$. c is stable wrt c' by t iff all states of c have a successor in c' by t . If this is not the case, then we must compute a partition of c in which each block is stable wrt c' by t . This is done by computing the predecessor class $c'' = (m'', Q'')$ of c' by t , that is, the largest set of states that have a successor by t in c' (not all these states may be reachable). Then, the subset c^t of states of c that have a successor in c' by t is exactly determined by $c^t = (m, Q^t = Q \cap Q'')$, and those not having one is $c^{-t} = (m, Q^{-t} = Q \setminus Q'')$, c^t and c^{-t} constitute the partition of c we are looking for (we say that c has been split). Since Q and Q'' are convex, Q^t is convex too, but Q^{-t} is not in general convex, it is a finite union of convex sets.

Computing the predecessor class c'' of c' by t is done by applying the method to obtain successor classes in Algorithm 30.2 backward. Starting from the clock system Q of c , Algorithm 30.2 first adds the firability conditions for t from c (subsystem (F) below). Next, new variables γ'_i are introduced for the persistent transitions (subsystem (N)) and for the newly enabled transitions. Finally, variables $\underline{\gamma}$ and θ are eliminated.

$$(Q) \ G\underline{\gamma} \leq \underline{g} \quad (F) \ A(\underline{\gamma}|\theta) \leq \underline{b} \quad (N) \ \underline{\gamma}'_i = \underline{\gamma}_i + \theta$$

For computing Q'' , newly enabled transitions may be omitted, let Q^- be Q' with these variables eliminated. System Q'' is then $F \wedge N \wedge Q^-$, after elimination of variables $\underline{\gamma}'$.

30.5.3 Building the Atomic State Class Graph

Algorithm 30.3 (computing atomic state classes) (Ref. 8)

Start from the SSCG (built by Algorithm 30.2 or as in Section 30.4.6)
while some class c is unstable wrt one of its successors c' by some t **do**
 split c wrt c' by t
 Collect all classes reachable from the initial class.

Splitting c replaces it by the set of classes resulting from the partition of c , as explained in Section 30.5.2. Only one of the resulting classes has a successor by t in c' , otherwise each subclass in the partition inherits the predecessors and successors of c , including itself and excluding c if c was a successor of itself. Details and optimizations are left out. The classes of the ASCG are considered modulo \equiv . Termination of Algorithm 30.3 follows from finiteness of the SSCG and of the partitions of classes by the stability condition.

30.5.4 Illustration

The ASCG of the net shown in Figure 30.2 admits 12 classes and 19 transitions, which is shown in Figure 30.3 (upper right). The sole nonstable class of the SSCG (Figure 30.3 (upper left)) is c_{10} , that was split into c'_{10} and a class equivalent by \equiv to c_5 . Class c_8 then became nonstable, and was in turn split into c'_8 and c_{11} .

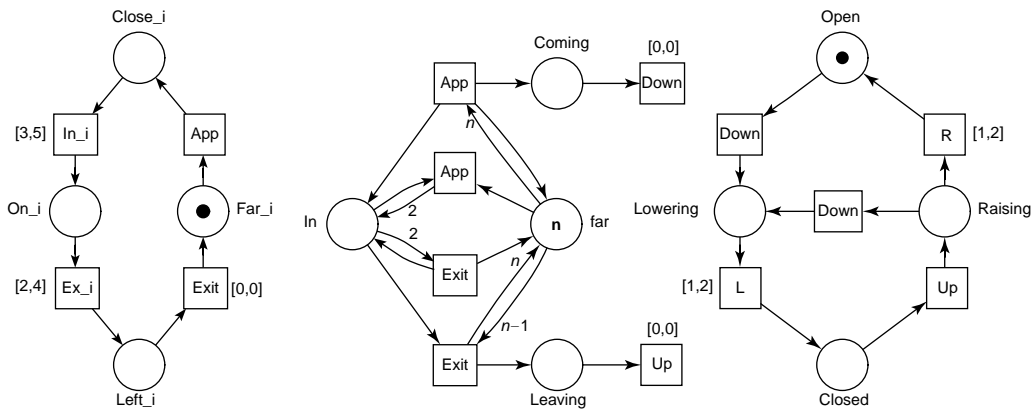
30.5.5 Properties Preserved

The ASCG is bisimilar with the discrete state graph DSG of the TPN , it thus obeys the same branching properties. In particular, it can be checked that the truth value of our example HML formula $\langle t_1 \rangle \langle t_2 \rangle [t]F$ is preserved by the ASCG (consider the path $c_0 \xrightarrow{t_1} c'_8 \xrightarrow{t_2} c'_{10}$).

Theorem 30.10

For any TPN , its DSG and $ASCG$ have the same states and are bisimilar.

Theorem 30.10 does not apply to the state graph SG . Owing to the continuous transitions appearing in the SG but not in the DSG , the SG and DSG are not in general bisimilar, even considering continuous



		SCG_{\lesssim}	SCG_{\cong}	$SSCG_{\leq}$	$SSCG_{\equiv}$	$ASCG(1)$	$ASCG(2)$
(1 train)	Classes	10	11	10	11	12	11
	Edges	13	14	13	14	16	15
	CPU(s)	0.00	0.00	0.00	0.00	0.00	0.00
(2 trains)	Classes	37	123	41	141	195	192
	Edges	74	218	82	254	849	844
	CPU(s)	0.00	0.00	0.00	0.00	0.01	0.01
(3 trains)	Classes	172	3101	232	5051	6973	6966
	Edges	492	7754	672	13019	49818	49802
	CPU(s)	0.00	0.07	0.01	0.15	3.50	2.28
(4 trains)	Classes	1175	134501	1807	351271	356959	356940
	Edges	4534	436896	7062	1193376	3447835	3447624
	CPU(s)	0.08	6.44	0.20	20.81	1264.89	320.58
(5 trains)	Classes	10972	—	18052	—	—	—
	Edges	53766	—	89166	—	—	—
	CPU(s)	1.93	—	5.40	—	—	—
(6 trains)	Classes	128115	—	217647	—	—	—
	Edges	760538	—	1297730	—	—	—
	CPU(s)	88.11	—	301.67	—	—	—

FIGURE 30.4 Level-crossing example.

transitions as silent transitions. One will notice that the *SG* is deterministic while the *DSG* is not. Concerning their branching structure, the difference between the *DSG* and the *SG* is in fact that the *DSG* interprets time elapsing as nondeterminism: the different states reachable from a state by letting time elapse are represented in the *SG*, ordered by continuous transitions, while they are considered unordered transient states in the *DSG* and not represented.

Concerning state reachability in the *SG* or *DSG*, it can be checked on the *ASCG* as it was checked on the *SSCG* in Section 30.4.5.

30.6 Computing Experiments

All SCG constructions presented in the previous sections have been implemented in a tool named Tina (TIme petri Net Analyzer), described in Ref. 7.

The example shown in Figure 30.4 is a *TPN* version of the classical level-crossing example. The net modeling the level crossing with n trains is obtained by parallel composition of n copies of the train model (lower left), synchronized with a controller model (upper left, n instantiated), and a barrier model (upper right). The specifications mix timed and untimed transitions (those labeled *App*).

For each n , we built with Tina for the level-crossing model with n trains:

- Its SCG, using either class equality as in Algorithm 30.2 (column SCG_{\cong} in Figure 30.4) or its variant in Section 30.3.4 using class inclusion (column SCG_{\subseteq})
- Its SSCG, using either class equality as in Algorithm 30.2 (column $SSCG_{\cong}$) or its variant in Section 30.4.6 using class inclusion (column $SSCG_{\subseteq}$)
- Its ASCG, using as starting partition either the SSCG built under the class equality rule (column $ASCG(1)$) or the SSCG built under the class inclusion rule (column $ASCG(2)$)

The results are shown in Figure 30.4, in terms of sizes and computing times on a typical desktop computer.

Safety properties reducing to marking reachability like “the barrier is closed when a train crosses the road” can be checked on any graph. Inevitability properties like “when all trains are far, the barrier eventually opens” can be expressed in *LTL* and may be checked on the SCG (SCG_{\cong}) or SSCG ($SSCG_{\cong}$). Potentiality properties like “at any time, any train which is far may approach” or the liveness property for the net (see Section 30.2.4) must be checked on the ASCG. Temporal properties like “when a train approaches, the barrier closes within some given delay” generally translate to properties of the net composed with “observer nets” deduced from the properties.

As can be expected, state space abstractions have sizes and costs that grow with the amount of information they preserve, from the SCG built under the class inclusion rule (only preserving markings) to the ASCG (preserving states, *LTL* properties, and branching properties). It can be noticed that both techniques for obtaining the ASCG yield comparable graphs, but that computing it from the SSCG built under the class inclusion rule is faster: The partition-refinement algorithm tends to generate many temporary classes, starting from a smaller graph typically produces less of them.

One will notice the fast increase in number of classes with the number of trains, for all constructions. For this particular example, this number could be greatly reduced by exploiting the symmetries of the state space resulting from replication of the train model.

30.7 Conclusion and Further Issues

30.7.1 On State Space Abstractions

Among those presented, the SCG construction of Refs. 3 and 5 presented in Section 30.3 should be adequate for most practical verification problems. It preserves markings and *LTL* properties, and is reasonably efficient in practice. *LTL* properties can be checked on the abstraction produced, they could also be checked on the fly while building the SCG. The SCG construction is supported by a number of tools, including Tina [7] and Romeo [14]. The strong state class (SSCG) and atomic state class (ASCG) graphs of Ref. 8, presented in Sections 30.4 and 30.5, bring additional verification capabilities, they are supported by the Tina tool. Both allow to decide state reachability, the first preserves the linear properties of the net and the second its branching properties (bisimilarity). The SSCG and ASCG constructions are closely related to the zone graph constructions for Timed Automata [1], another widely used model for real-time systems.

An alternative to state classes methods must be mentioned: It is shown in Ref. 24 that the behavior of a bounded *TPN* can be finitely represented by a subgraph of the *DSG* defined in Section 30.2.2. Intuitively, assuming the endpoints of static intervals are integers and that all intervals are closed, or unbounded and left closed, the set of states reachable from the initial state by continuous transitions with integer delays and then a discrete transition is finite. The abstraction obtained, called the *essential states* graph, preserves the *LTL* properties of the *DSG*, like the SCG, it is supported by the tool Ina [28]. For nets with intervals of small width and small endpoints, the essential states method may yield state space abstractions smaller than state class graphs, because it produces nondeterministic graphs, but state class graphs are typically smaller. In particular, scaling the static intervals of a *TPN* does not affect its SCGs (the timing information

in classes is just scaled), while its essential SG might grow by a large factor. Implementing the essential states method is much simpler, however.

Another group of methods for analyzing *TPNs* rely on the fact that they can be encoded into Timed Automata, preserving weak timed bisimilarity [13]. The translation makes possible to use for *TPNs* the analysis methods developed for Timed Automata [14]. Also, since Timed Automata and *TPNs* share the same semantic model of timed transition systems, most techniques for analysis of Timed Automata can be adapted to Petri nets. This is done in Ref. 30, for instance, for checking *TCTL* properties of *TPNs* (*TCTL* is an extension of *CTL* with modalities annotated by time constraints). The relationships between Timed Automata and *TPNs* are also discussed in Ref. 23, together with a number of model-checking issues.

Finally, an alternative to temporal logics to check real-time properties is to analyze the firing schedules, also called paths, of the state space. A variant of Algorithm 30.2, for instance, can be used to compute the dates at which transitions can fire along some firing sequence: It is just necessary for this to keep the auxiliary variables θ in the systems computed, instead of eliminating it. The resulting inequality system can be used to check time constraints on all schedules over some firing sequence. Such methods, called *path analysis*, are discussed in Refs. 10, 25, and 29.

30.7.2 Extensions of *TPNs*

TPNs have been extended in various ways, increasing or not their expressiveness, and state class style methods have been developed to analyze nets with these extensions.

Extensions like inhibitor arcs and read arcs, special arcs that test Boolean conditions on markings but do not transfer tokens, may significantly compact some specifications, they are supported by, e.g., *Tina*. These devices extend expressiveness of bounded *TPNs* in terms of timed bisimulation, but preserve their decidability. State class methods are easily extended to accommodate these extensions, as they mostly impact enabledness conditions and computation of markings.

Other extensions more deeply impact expressiveness of *TPNs*. The scheduling-extended *TPNs* of Ref. 18, the preemptive *TPNs* of Ref. 11, and the stopwatch *TPNs* of Ref. 4, for instance, supported respectively by the tools *Romeo*, *Oris*, and an extension of *Tina*, allow suspension and resumption of transitions. They allow one to model and analyze real-time systems with preemptive scheduling constraints. State class-based analysis methods extend to these models, but yield partial algorithms (termination is not guaranteed). In fact, it is shown in Ref. 4 that state reachability in these models is undecidable, even for bounded nets. Finite state space abstractions can be obtained by over-approximation methods, yielding abstractions capturing the exact behavior of nets, but possibly a larger set of states or firing schedules.

Finally, priorities are another extension of *TPNs* of practical and theoretical value. It is shown in Ref. 6 that static priorities strictly increase the expressiveness of *TPNs*, and that *TPNs* extended with priorities can simulate a large subclass of Timed Automata. Extension of state class methods to *TPNs* with priorities is being investigated.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *CONCUR 92: Theories of Concurrency, LNCS 630*, Springer, Berlin, pp. 340–354, 1996.
2. B. Berthomieu. La méthode des classes d'états pour l'analyse des réseaux temporels—mise en œuvre, extension la multi-sensibilisation. In *Modélisation des Systèmes Réactifs*, Hermes, France, 2001.
3. B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3): 259–273, 1991.
4. B. Berthomieu, D. Lime, O.H. Roux, and F. Vernadat. Reachability problems and abstract state spaces for time Petri nets with stopwatches. *Journal of Discrete Event Dynamic Systems*, 2007 (to appear).

5. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. *IFIP Congress Series*, 9: 41–46, 1983.
6. B. Berthomieu, F. Peres, and F. Vernadat. Bridging the gap between timed automata and bounded time Petri nets. In *4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, LNCS 4202, Springer, Berlin, 2006.
7. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA—construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14): 2741–2756, 2004.
8. B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, Warsaw, Poland, LNCS 2619, Springer, Berlin, pp. 442–457, 2003.
9. H. Boucheneb and R. Hadjidj. Towards optimal CTL* model checking of time Petri nets. In *Proceedings of 7th Workshop on Discrete Events Systems*, Reims, France, September 2004.
10. H. Boucheneb and J. Mullins. Analyse des réseaux temporels: Calcul des classes en $O(n[2])$ et des temps de chemin en $O(mn)$. *Technique et Science Informatiques*, 22: 435–459, 2003.
11. G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Timed state space analysis of real-time preemptive systems. *IEEE Transactions on Software Engineering*, 30(2): 97–111, 2004.
12. G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Transactions on Software Engineering*, 21(12): 969–992, 1995.
13. F. Cassez and O.H. Roux. Structural translation from time Petri nets to timed automata. *Journal of Systems and Software*, 2006. (forthcoming).
14. G. Gardey, D. Lime, M. Magnin, and O.(H.) Roux. Roméo: A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification*, LNCS 3576, Springer, Berlin, July 2005.
15. R. Hadjidj. *Analyse et validation formelle des systèmes temps réel*. PhD Thesis, Ecole Polytechnique de Montréal, Université de Montréal, February 2006.
16. N.D. Jones, L.H. Landweber, and Y.E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4, 277–299, 1977.
17. P.K. Kanellakis and S.A. Smolka. CCSexpressions, finite state processes, and three problems of equivalence newblock. *Information and Computation*, 86: 43–68, 1990.
18. D. Lime and O.H. Roux. Expressiveness and analysis of scheduling extended time Petri nets. In *5th IFAC International Conference on Fieldbus Systems and their Applications*, Aveiro, Portugal, Elsevier Science, pp. 193–202, July 2003.
19. P.M. Merlin. *A Study of the Recoverability of Computing Systems*. PhD Thesis, University of California, Irvine, 1974.
20. P.M. Merlin and D.J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Trans. Communication*, 24(9): 1036–1043, 1976.
21. P. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6): 973–989, 1987.
22. W. Penczek and A. Pórlola. Abstraction and partial order reductions for checking branching properties of time Petri nets. In *Proceedings of the 22st International Conference on Application and Theory of Petri Nets (ICATPN 2001)*, LNCS 2075, Springer, Berlin, pp. 323–342, 2001.
23. W. Penczek and A. Pórlola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach*. Studies in Computational Intelligence, Vol. 20, Springer, Berlin, 2006.
24. L. Popova. On time Petri nets. *Journal of Information Processing and Cybernetics EIK*, 27(4): 227–244, 1991.
25. L. Popova and D. Schlatter. Analyzing paths in time Petri nets. *Fundamenta Informaticae*, 37(3): 311–327, 1999.
26. T.G. Rokicki. *Representing and Modeling Circuits*. PhD Thesis, Stanford University, Stanford, CA, 1993.

27. T.G. Rokicki and C.J. Myers. Automatic verification of timed circuits. In *6th Conference Computer Aided Verification, CAV'94, LNCS 818*, Springer, Berlin, pp. 468–480, June 1994.
28. P.H. Starke. Ina—Integrated Net Analyzer, Reference Manual. Technical report, Humboldt Institute of Informatics, Berlin, 1997.
29. E. Vicario. Static analysis and dynamic steering of time-dependent systems. *IEEE Transactions on Software Engineering*, 27(8): 728–748, 2001.
30. I. Virbitskaite and E. Pokozy. A partial order method for the verification of time petri nets. In *Fundamentals of Computation Theory: 12th International Symposium (FCT'99)*, Iasi, Romania, LNCS 1684, Springer, Berlin, pp. 547–558, 1999.
31. T. Yoneda and H. Ryuba. CTL model checking of time Petri nets using geometric regions. *IEEE Transactions on Information and Systems*, E99-D(3): 1–10, 1998.
32. T. Yoneda, A. Shibayama, B.-H. Schlingloff, and E.M. Clarke. Efficient verification of parallel real-time systems. In *5th Conference on Computer Aided Verification, LNCS 697*, Springer, Berlin, pp. 321–332, June 1993.

31

Process-Algebraic Analysis of Timing and Schedulability Properties

Anna Philippou

University of Cyprus

Oleg Sokolsky

University of Pennsylvania

31.1	Introduction	31-1
	Conventional Process Algebras	
31.2	Modeling of Time-Sensitive Systems	31-4
	Discrete-Time Modeling • Comparative Example	
31.3	Modeling of Resource-Sensitive Systems	31-9
	Syntax and Semantics • Schedulability Analysis with ACSR	
31.4	Conclusions	31-19

31.1 Introduction

Process algebras, such as CCS [13], CSP [9], and ACP [3], are a well-established class of modeling and analysis formalisms for concurrent systems. They can be considered as high-level description languages consisting of a number of operators for building processes including constructs for defining recursive behaviors. They are accompanied by semantic theories that give precise meaning to processes, translating each process into a mathematical object on which rigorous analysis can be performed. In addition, they are associated with axiom systems that prescribe the relations between the various constructs and can be used for reasoning about process behavior. During the last two decades, they have been extensively studied and proved quite successful in the modeling and reasoning about various aspects of system correctness.

In a process algebra, there exist a number of elementary processes as well as operators for composing processes into more complex ones. A system is then modeled as a term in this algebra. The rules in the operational semantics of an algebra define the steps that a process may take in terms of the steps of the subprocesses. When a process takes a step, it evolves into another process. The set of processes that a given process P can evolve into by performing a sequence of steps defines the state space of P . The state space is represented as a labeled transition system (LTS) with steps serving as labels. The notation for a process P performing a step a and evolving into a process P' is $P \xrightarrow{a} P'$.

State space exploration techniques allow us to analyze properties of processes, for example, to identify deadlocks that the system may have.

31.1.1 Conventional Process Algebras

Instead of giving a formal definition for some process algebra, we will begin by supplying intuition for the design of a process-algebraic formalism. Without being specific to any particular process algebra, we will introduce basic concepts that are present in a typical process algebra and highlight the relationship

between them. This exposition will help us illustrate how the introduction of timing into a process algebra, discussed in later sections, affects the design of a formalism.

Syntax and semantics. The syntax of a process algebra is defined by a collection of operators that give rise to a set of processes. Each operator has a nonnegative arity. Operators of arity 0 represent elementary processes. Operators of arity n take n processes and form a new composite process from them. A semantic definition for such an algebra is formalized by means of structural operational semantics: for each operator a set of rules is supplied that prescribes how the steps of the constituent processes are transformed into a step of a composite process. Examples of such rules are given later in the section.

Basic execution steps. Basic execution steps of a process P are taken from a set called the *alphabet* of P , which we denote as $\mathcal{A}(P)$. Since execution steps serve as transition labels in the LTS of P , we generically refer to the elements of the alphabet as labels. Labels in the alphabet of conventional process algebras are often called *events* or *actions*. Here, we will refer to them as *events*, reserving the term *actions* to a special kind of steps which will be introduced in Section 31.3. Real-time extensions of process algebras also introduce additional kinds of labels to account for time, as discussed in Section 31.2. The nature of events depends on the modeling domain: it can correspond, for example, to a method call in the model of a software system or to receiving a packet from the network in the model of a communication protocol.

Common operators and rules. The syntax of a typical process algebra is given by the following BNF definition

$$P ::= \text{NIL} \mid e.P \mid P + P \mid P \parallel P \mid P \setminus A \mid C$$

where e and C range over a set \mathcal{A} of events and a set \mathcal{C} of process constants associated with the process algebra, respectively, and $A \subset \mathcal{A}$. Each operator is given precise meaning via a set of rules that, given a process P , prescribe the possible transitions of P , where a transition of P has the form $P \xrightarrow{e} P'$, specifying that process P can perform event e and evolve into process P' . The rules themselves have the form

$$\frac{T_1, \dots, T_n}{T} \phi$$

which is interpreted as follows: if transitions T_1, \dots, T_n can be derived and condition ϕ holds, then we may conclude transition T . Conditions T_1, \dots, T_n describe possible transitions of the components of a system and may be absent for some rules. We proceed to discuss each of the above operators and ways in which they are interpreted in different process-algebraic formalisms.

- **NIL.** The *inactive process*, represented by the operator NIL, does not have any rules associated with it and thus cannot perform any steps.
- **$e.P$.** The most basic operator of a process algebra is the *prefix* operator. If P is a process then $e.P$ is also a process, for every event e . This process performs event e and then behaves as P . The rule associated with a prefix operator usually has the form

$$(\text{ACT}) \quad \frac{-}{e.P \xrightarrow{e} P}$$

An alternative to the prefix operator is a more general *sequential composition* operator $P_1 \cdot P_2$ for arbitrary processes P_1 and P_2 . We refer the reader to Ref. 3 for details.

- **$P_1 + P_2$.** Another commonly used operator is the choice operator, $P_1 + P_2$, which yields a process that spontaneously evolves into P_1 or P_2 . The rules for the choice operator are

$$(\text{SUM}) \quad \frac{P_1 \xrightarrow{e} P'_1}{P_1 + P_2 \xrightarrow{e} P'_1}$$

and a symmetric rule for P_2 . We say that the first event performed by P_1 or P_2 resolves the choice.

- $P_1 \parallel P_2$. One of the main points of distinction between different process algebras is the *parallel composition* operator, which we denote here as $P_1 \parallel P_2$. Details of the definition for this operator specify whether processes in the model execute synchronously or asynchronously and describe the means of communication and synchronization between concurrent processes. In the case of asynchronous execution, processes can take steps independently. This case is expressed by the rule

$$(\text{PAR}_A) \quad \frac{P_1 \xrightarrow{e} P'_1}{P_1 \parallel P_2 \xrightarrow{e} P'_1 \parallel P_2}$$

Note that here, unlike the choice operator, P'_1 remains in the scope of the parallel operator. P_2 does not make a step in this case. There is a symmetric rule for P_2 .

In the case of a synchronous execution, all concurrent processes are required to take a step. A generic rule for the synchronous case can be expressed as

$$(\text{PAR}_S) \quad \frac{P_1 \xrightarrow{e_1} P'_1 \quad P_2 \xrightarrow{e_2} P'_2}{P_1 \parallel P_2 \xrightarrow{e} P'_1 \parallel P'_2}$$

The relationship between e and e_1, e_2 captures the synchronization details. In what is known as *CCS-style* synchronization, the alphabet \mathcal{A} is partitioned into the sets of input events, denoted as $e?$, output events, denoted as $e!$, and a distinguished internal event τ . A handshake synchronization between two processes turns an input event and a matching output event into an internal event, represented by the rule

$$(\text{PAR}_{\text{CCS}}) \quad \frac{P_1 \xrightarrow{e?} P'_1 \quad P_2 \xrightarrow{e!} P'_2}{P_1 \parallel P_2 \xrightarrow{\tau} P'_1 \parallel P'_2}$$

In *CSP-style* synchronization, components of a process P must synchronize on a predefined set $A \subseteq \mathcal{A}$ and we have that $e = e_1 = e_2 \in A$. In other formalisms, the result of a synchronization is a composite event, for example, $e_1 \cdot e_2$. The general case is called the *ACP-style* synchronization, which employs a *communication function* f . The communication function is a partial function that maps pairs of events to events. The rule

$$(\text{PAR}_{\text{ACP}}) \quad \frac{P_1 \xrightarrow{e_1} P'_1 \quad P_2 \xrightarrow{e_2} P'_2}{P_1 \parallel P_2 \xrightarrow{f(e_1, e_2)} P'_1 \parallel P'_2} \quad f(e_1, e_2) \neq \perp$$

is equipped with a side condition that makes the rule applicable only if the communication function is defined for e_1 and e_2 .

- $P \backslash A$. This process represents the *hiding* or *restriction* operator representing the process P when events from set A are hidden. This is another operator interpreted differently in distinct process algebras. These interpretations are closely related to the communication style adopted. In particular, in the ACP style of communication, hiding of an action prohibits the action from taking place altogether. However, in CCS, where the set of actions contains input and output actions along channels, hiding of a channel e within a system P prevents actions $e?$ and $e!$ from taking place on the interface of P with the environment, but not within the components of P .
- C . A common mechanism for providing recursion in the process algebra is *process constants*. A set of process constants, \mathcal{C} , is assumed. Each $C \in \mathcal{C}$ is associated with a definition $C \stackrel{\text{def}}{=} P$, where the process P may contain occurrences of C , as well as other constants. The semantics of process constants is given by the following rule:

$$(\text{CON}) \quad \frac{P \xrightarrow{e} P'}{C \xrightarrow{e} P'} \quad C \stackrel{\text{def}}{=} P$$

TABLE 31.1 Axioms of a Typical Process Algebra

(Sum1)	$P + \text{NIL} = P$	(Par1)	$P \parallel \text{NIL} = P$
(Sum2)	$P + Q = Q + P$	(Par2)	$P \parallel Q = Q \parallel P$
(Sum3)	$(P + Q) + R = P + (Q + R)$	(Par3)	$(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
(Hide1)	$\text{NIL} \setminus A = \text{NIL}$	(Hide2)	$(a.P) \setminus A = a.(P \setminus A), \text{ if } a \notin A$

Equational theory. A significant component of a process-algebraic framework is its equational theory, a set of *axioms* or *equations* that equates process expressions of the language. Some process algebras, such as those in Ref. 3 and its derivatives, are even defined equationally since their designers consider this to be the most intuitive starting point for understanding the algebra and its operators. In the case that an algebra's semantics are given operationally, as discussed above, this theory is produced on the basis of an *equivalence relation*, which deems two processes to be related if and only if their respective LTSs are “similar,” according to a specific notion of similarity. Based on such an equivalence relation, an equational theory is *sound* if all of its axioms equate equivalent processes and, it is *complete*, if any pair of equivalent processes can be shown to be equal by the equational theory.

The importance of equational theories is twofold. On the one hand, they shed light on the subtleties of a language's operators and the interactions that exist between them. As such, they can be useful for revealing differences between different algebras, their operators, and their associated equivalences. On the other hand, they support system verification by equational reasoning. Note that, to support *compositional* reasoning, they are typically defined over *congruence* relations, that is, relations which are preserved by all of a language's operators. One of the most common such relations is *strong bisimulation* [13,17]. Strong bisimulation is an equivalence relation that respects the branching structure of processes. Intuitively, two processes are bisimilar if they can simulate each other's behavior step by step. In Table 31.1, we present a subset of a basic process algebra's equational theory.

We note that the inactive process serves as the *unit process* for the parallel composition and choice operators (Axioms: (Sum1) and (Par1)).

An important axiom in the equational theory of most conventional process algebras is the *expansion theorem*, which allows to rewrite a process containing the parallel composition operator to an equivalent process without parallel composition. The expansion theorem allows simpler analysis algorithms that operate on the process LTS. Application of the expansion theorem, in CCS, results in the following equation

$$e?.P \parallel e!.Q = e?.(P \parallel e!.Q) + e!.(e?.P \parallel Q) + \tau.(P \parallel Q)$$

whereas, in CSP, it yields

$$e.P \parallel e.Q = e.(P \parallel Q)$$

It is interesting to note that the interplay between the parallel composition and the restriction operator gives rise to $(e?.P \parallel e!.Q) \setminus \{e\} = \tau.(P \parallel Q)$ in CCS and $(e.P \parallel e.Q) \setminus \{e\} = \text{NIL}$ in CSP. This highlights the fact that, while in CCS hiding of a channel within a system does not preclude a handshake communication between components of the system, in CSP the action is prevented from arising altogether.

31.2 Modeling of Time-Sensitive Systems

Conventional process algebras offer a way of modeling and analyzing qualitative, functional aspects of system behavior. Several *real-time process algebras* extend conventional process algebras to add the quantitative timing aspect to the system model. In this section, we look at the possible ways to add timing information to a process-algebraic model and discuss the effect of such extensions on the rules of the algebra and the interactions between them. Classification ideas may be found in Ref. 15.

Two-phase execution. The approach taken by most real-time process algebras is to introduce a new kind of labels into the process alphabet to represent the progress of time. Labels of this kind can either be real

or rational numbers, or new symbols that are not present in the alphabet of the untimed process algebra. The former approach is known as a *continuous time* extension, and a step labeled by a number represents advancing the time by the specified amount. The latter approach is called *discrete time*, and a step labeled by a label of the new kind represents time advancing by a fixed amount. We refer to such steps as *ticks*. All other steps of a process are assumed to be instantaneous.

Execution of a process proceeds in two phases. Initially, a sequence of instantaneous steps are executed at the same time instance, followed by a time step that advances time. An execution that contains an infinite sequence of instantaneous steps without allowing a time step to take place is called *Zeno*. Clearly, such executions do not have a physical meaning and need to be ruled out either by the semantics of the formalism or by means of analysis.

Progress. Conversely, a process can choose to pass time indefinitely even though it is capable of performing instantaneous steps. While not as harmful as *Zeno* behaviors, such idling behaviors are also undesirable and should be avoided. Therefore, most formalisms include some means of ensuring progress in an execution. Many real-time process algebras employ the notion of *maximal progress*, which forces synchronization between concurrent processes, whenever it can occur, to occur before the next time step. Some process algebras, such as CSA [7] and ACSR [11], employ a more restricted notion of progress, as described below.

Global versus local time. Most real-time process algebras employ an implicit global clock that is shared by all processes in the system. This means that time progress is *synchronous*, that is, time can advance only if all processes in the system agree to pass time together. This is in contrast to the execution of instantaneous steps of different processes, which may be interleaved even though they are logically simultaneous.

The assumption of uniform global time may be restrictive when modeling distributed systems, where each node may have its own clock that advances at its own rate. To address this problem, CSA [7] considers a multiclock process algebra. In this algebra, steps can have different time progress labels that correspond to a tick of a particular clock, and only the processes that are within the scope of that clock are required to participate in that step.

31.2.1 Discrete-Time Modeling

In this chapter, we concentrate on discrete-time modeling. For details on continuous-time process algebras, the interested reader is referred to Refs. 1, 21, and 25. We consider real-time process algebras that assume a global clock, shared by all concurrent processes, which advances in discrete increments. A tick of that clock is represented by a special label ϵ , which is added to the process alphabet. Events inherited from the untimed versions are considered to be instantaneous, thus are kept independent from the passage of time. Process algebras that fall into this category are ATP [16], TPL [8], TCCS [14], and several ACP flavors [1,2]. The process algebra ACSR, discussed in Section 31.3, also assumes a global clock but has a richer structure for time progress labels.

Syntactic extensions. One of the first steps in designing a real-time process algebra is to decide whether time progress steps are included into the syntax of the process terms. In a conventional process algebra, every event e can appear in the event-prefix operator $e.P$. TCCS and TPL treat time steps in the same way by introducing a *time-prefix* operator $\epsilon.P$, with the obvious rule:

$$(TACT) \quad \frac{}{\epsilon.P \xrightarrow{\epsilon} P}$$

Other process algebras, such as ATP, do not have time-prefix operators and extend the semantics for other operators with rules about time steps.

Extending operators with rules for time progress. Time progress rules can be added to any operator in the algebra. However, implementation of the rules for different operators is not done in isolation and allow the algebra designers to offer interesting modeling devices. Below, we consider different choices for rule design and look at the interplay between them.

Conventional process algebras usually have one kind of inactive process, which we have denoted by NIL. In real-time process algebras, it is useful to distinguish between a completed process, which we will call δ , and a deadlocked process, which we will call NIL. The semantic difference is, of course, that the completed process cannot perform any actions but allows time progress. That is, it is equipped with the rule

$$(IDLE) \quad \frac{-}{\delta \xrightarrow{\epsilon} \delta}$$

On the contrary, the deadlocked process does not have any rules associated with it, thus preventing time from progressing any further. Process algebras featuring this time-blocking deadlocked process include TCCS, ACSR, and ACP_ρ [2]. By this process, it is possible to model abnormal conditions that require attention, and can be employed for detecting anomalies in system models (e.g., in ACSR, the presence of timelocks can be used to test for schedulability of a system model).

It is worth noting here that if the algebra has the time-prefix operator, then the completed process can be represented by a secondary operator, defined by the recursive definition $\delta \stackrel{\text{def}}{=} \epsilon.\delta$.

Next, we consider event prefix. Here, we may distinguish two different approaches on the treatment of event execution.

- A first approach, adopted by the process algebra TPL, allows events to be so-called lazy or patient with respect to time passage. To implement this, the prefix operator is accompanied by the rule

$$(WAIT) \quad \frac{-}{a.P \xrightarrow{\epsilon} a.P}$$

signifying that the process $a.P$ can idle indefinitely. The motivation behind this law is the intention for all processes to be patient until the communications in which they can participate become enabled. Then, progress in the system is assured by appropriate treatment of the parallel composition operator as we discuss below. Note that, if events are lazy, there is no need for the time-prefix operator.

- Alternatively, the algebra designed has an option of making events *urgent*. In this case, the event-prefix operator cannot idle and progress is ensured. This approach is adopted in ATP and TCCS. In the case of TCCS, time passage is enabled (solely) by the aid of a time-prefix operator, whereas ATP features a unit-delay operator. In some process algebras, such as TPL, urgency of events only applies to the internal action.

However, modelers using a process algebra with urgent events can inadvertently block time progress completely, introducing unwanted Zeno behavior in the model. In ATP, the term *well-timed* systems has been coined to refer to systems that contain no Zeno behavior. In general, there is no syntactic characterization of well-timed systems. It is, however, possible to discover Zeno behavior by state space exploration.

Continuing with the choice construct, we may see that the majority of timed process algebras proposed in the literature implement *time determinism*. That is, they define the choice construct so that it postpones the resolution of the choice between its summands in the case of system idling. This is described by the following rule:

$$(TSUM) \quad \frac{P_1 \xrightarrow{\epsilon} P'_1, P_2 \xrightarrow{\epsilon} P'_2}{P_1 + P_2 \xrightarrow{\epsilon} P'_1 + P'_2}$$

Note, however, that in an algebra with urgent events a process may not be able to pass time and the choice may have to be resolved before time advances.

As far as parallel composition is concerned, we first observe that time progresses synchronously in all parallel components, that is, timed actions can be performed only if all the system components

agree to do so:

$$(TPAR) \quad \frac{P_1 \xrightarrow{\epsilon} P'_1, P_2 \xrightarrow{\epsilon} P'_2}{P_1 \parallel P_2 \xrightarrow{\epsilon} P'_1 \parallel P'_2}$$

Specifically, in the presence of a communication function f , we have that $f(e_1, e_2) = \perp$ if one of e_1, e_2 is ϵ but not both.

Further details concerning the interaction of time passage and the parallel composition construct depend on the synchronization style featured in an algebra as well as the nature of events. In the case of CCS-style communication and patient events, we may see maximal progress implemented by the following rule:

$$(TPAR_{MP}) \quad \frac{P_1 \xrightarrow{\epsilon} P'_1, P_2 \xrightarrow{\epsilon} P'_2, P_1 \parallel P_2 \not\xrightarrow{\tau}}{P_1 \parallel P_2 \xrightarrow{\epsilon} P'_1 \parallel P'_2}$$

This specifies that $P \parallel Q$ may delay if both components may delay and no internal communication between the two is possible. It can be encountered in TPL, where, for example, in the presence of patient actions,

$$a!.P + b?.Q \xrightarrow{\epsilon} a!.P + b?.Q$$

allows time to pass; whereas, the composition $(a!.P + b?.Q) \parallel a? \cdot \text{NIL}$ does not. We should also note that under this rule, the parallel operator construct does not preserve Zenoness: $P \stackrel{\text{def}}{=} a?.P$ and $Q \stackrel{\text{def}}{=} a!.Q$, are non-Zeno processes, but $P \parallel Q$ is not.

The rule for maximal progress is not featured in algebras such as TCCS and ATP. It becomes unnecessary due to the fact that insistent events (recall that the event-prefix operator cannot idle) force progress to be made. CSA replaces maximal progress with locally maximal progress appropriate for distributed systems with multiple clocks. As we will see in the next section, ACSR replaces maximal progress with resource-constrained progress.

We continue by reviewing some time-specific constructs introduced in process algebras.

- In addition to the time-prefix operator, TCCS contains an unbounded delay operator $\delta.P$ (note that this is different to the completed process δ), which may delay indefinitely before proceeding with the computation. This resembles the willingness of patient events to wait for an environment ready to participate. Its behavior is specified by the following two rules:

$$\frac{-}{\delta.P \xrightarrow{\epsilon} \delta.P} \quad \frac{P \xrightarrow{e} P'}{\delta.P \xrightarrow{e} P'}$$

- In ATP, the basic time-passing construct is the binary *unit-delay operator*, $[P](Q)$. This process behaves like P if the execution of P starts before the first tick of time takes place. Otherwise, control is passed to process Q . Thus, we may consider the operator as a timeout construct with delay 1. Variants of this operator, including unbounded delay, and bounded but nonunit delay, are encountered in ATP and other algebras.
- Time interrupt constructs, of the form $\lceil P \rceil^d(Q)$, have been defined in ATP and TCSP, and describe a process where computation proceeds as for process P for the first d time units, and then Q is started.
- Another useful construct, encountered in TCSP, ATP, and ACSR, is the timeout operator. In TCSP and ATP, this is modeled as a binary operator, $P \triangleright^d Q$, where d is a positive integer, P is called the body, and Q the exception of the timeout. The intuitive meaning of this process is that P is given permission to start execution for d time units, but, if it fails to do so, control is passed to process Q . In ACSR, as we will see below, the timeout operator is more complex and combines the notions of successful termination and interrupts.

Let us now look into the interplay between the above-mentioned operators by studying the resulting axioms in various process algebras. First, consider rules (Sum1) and (Par1) of Table 31.1. We observe that, in the presence of axioms (TSUM) and (TPAR), respectively, they do not hold in any of the above-mentioned approaches. However, in the presence of the completed process δ , we have

$$(\text{CSum1}) \quad P + \delta = P \quad (\text{CPar1}) \quad P \parallel \delta = P$$

In contrast, the blocked process NIL satisfies the laws

$$\begin{array}{ll} (\text{BSum1}) \quad e.P + \text{NIL} = e.P & (\text{BPar1}) \quad e.P \parallel \text{NIL} = e.P \\ (\text{BSum2}) \quad \epsilon.P + \text{NIL} = \text{NIL} & (\text{BPar2}) \quad \epsilon.P \parallel \text{NIL} = \text{NIL} \end{array}$$

When restricting attention to regular processes; that is, processes using finitely many guarded recursive equations, an expansion theorem can be deduced for all of the mentioned process algebras.

31.2.2 Comparative Example

To tie together the concepts introduced in the previous sections, and to illustrate the effect of design decisions within process-algebraic formalisms on the modeling style they impose, we show two models of the same system in the algebras ATP and TPL. We begin by summarizing the syntax of the two algebras and their associated semantical meaning. To keep presentation simple, we unify the notations of the two formalisms for operators representing the same concept. However, semantic differences between respective constructs may be present and we highlight them by mentioning the semantic rules which apply in each case.

ATP. The syntax of the process algebra ATP is given as follows:

$$P ::= \text{NIL} \mid C \mid a.P \mid P + P \mid \lfloor P \rfloor(P) \mid P \parallel P \mid P \setminus A \mid \lceil P \rceil^d(P)$$

The operators of the language satisfy, among others, the rules (ACT) (and not (WAIT)), (SUM), (TSUM), and (TPAR) (and not (PAR_{MP})). The terminated process δ can be derived in ATP as follows: $\delta \stackrel{\text{def}}{=} \lfloor \text{NIL} \rfloor(\delta)$.

TPL. The syntax of the process algebra TPL is given as follows:

$$P ::= \text{NIL} \mid C \mid a.P \mid \epsilon.P \mid P + P \mid \lfloor P \rfloor(P) \mid P \parallel P \mid P \setminus A$$

The operators of the language satisfy, among others, the rules (ACT), (WAIT), (SUM), (TSUM), and (PAR_{MP}).

Example

We now proceed to model a simple system in the two algebras. We will show that, despite their syntactic closeness, subtle semantic differences in the definitions of the operators will require us to model the same system differently to achieve the same behavior in both languages.

The system under consideration represents two real-time tasks. Task T_1 is a periodic task, which is released every 10 time units and nondeterministically takes either 1 or 2 time units to complete. Once task T_1 completes its execution, it sends a message to release the task T_2 , which takes 3 time units to complete. This example has several features that highlight the difference between different process algebras: the treatment of the timeout used in the periodic release of the first task, the choice between time progress and sending an event, necessary to implement the nondeterministic completion time of the first task, and implementation of a lazy event in the second task to receive the release signal from the first task.

We first consider the representation of this system in ATP.

$$\begin{aligned}
 T_1 &\stackrel{\text{def}}{=} [\lfloor \delta \rfloor (start.\delta + \lfloor \delta \rfloor (start.\delta))]^{10}(T_1) \\
 T_2 &\stackrel{\text{def}}{=} start.\lfloor \delta \rfloor (\lfloor \delta \rfloor (\lfloor \delta \rfloor (T_2))) + \lfloor \delta \rfloor (T_2) \\
 System &\stackrel{\text{def}}{=} (T_1 \parallel T_2) \setminus \{start\}
 \end{aligned}$$

Note that the modeling style of ATP is to describe the events that the process can perform during a given time unit, each within the scope of a unit-delay operator. Consider T_1 . In the first time unit, the process is engaged in an internal computation and cannot perform any events. In the second time unit, the process has two choices of behavior. If the internal computation has been completed in the first time unit, the process can perform the event *start* and then idle until the timeout of 10 time units. Otherwise, it idles for another time unit and then has no other choice but to perform the event *start* and wait for the timeout. The process T_2 begins execution when event *start* arrives, with the summand $\lfloor \delta \rfloor (T_2)$ allowing it to idle as needed. Finally, the system is the parallel composition of the two task processes, with the *start* event restricted to ensure CSP-style synchronization.

Consider now the same system expressed in TPL. Modeling is distinguished by the ATP description due to ATP's patient events, the explicit time-prefix operator, and the absence of a timeout operator. Furthermore, recall that TPL adopts the CCS style of communication, which results in the use of input and output events for the synchronization of the two tasks.

$$\begin{aligned}
 T_1 &\stackrel{\text{def}}{=} \epsilon.(start!. \epsilon^9.T_1 + \tau.\epsilon.start!. \epsilon^8.T_1) \\
 T_2 &\stackrel{\text{def}}{=} start?. \epsilon.\epsilon.\epsilon.T_2 \\
 System &\stackrel{\text{def}}{=} (T_1 \parallel T_2) \setminus \{start\}
 \end{aligned}$$

Here, for simplicity, we employ the notation $\epsilon^n.P$ for the process that makes n consecutive executions of event ϵ before proceeding to process P . Thus, task T_1 initially performs some internal computation during the first time unit, and then it evolves into a choice process. This presents the two possibilities for the duration of the execution of the task. The first summand corresponds to the case the task is completed after a single time unit, in which case a *start* message is emitted. Alternatively, an internal action is performed by the task and, after a second time unit elapses, task T_2 is notified and the process idles for a further 8 time units. Note that, unlike the ATP model, here it is not necessary to model explicitly the waiting of task T_2 : the process waits indefinitely until the action becomes enabled. However, owing to the maximal progress, once the communication on the channel *start* becomes enabled, it cannot be further delayed by time passage. Consequently, the τ action included in the description of T_1 is crucial for the correct description of the model. Had it been absent, maximal progress would enforce the communication on *start* to take place immediately in a process such as $(start!.P + \epsilon.Q \parallel start?.R)$.

31.3 Modeling of Resource-Sensitive Systems

The notion of maximal progress employed by many real-time process algebras is a useful modeling concept. Maximal progress makes the model engage in useful computation whenever it is able to do so. However, there are many cases when maximal progress does not fit well with the application domain. Many systems that operate under real-time constraints are also constrained by available computational resources. This is especially true of embedded systems that have to operate on processors with limited computing power and small memory. Clearly, satisfaction of timing constraints by such systems depends on timely availability of resources needed by each process. Therefore, modeling of resource-constrained systems replaces the notion of maximal progress with the notion of resource-constrained progress.

Process algebra CCSR [11] implements this notion of resource-constrained progress by associating resource consumption with time passage. Specifically, it replaces the single timed event ϵ , discussed in the

previous section, by a structured set of labels: whenever a process wants to perform a step that advances time, it declares the set of resources that are needed for this step, along with the priority of access for each resource. This resource declaration is called an *action*. The semantic rules of the parallel composition operator ensure that only actions using disjoint resources are allowed to proceed simultaneously. Thus, while progress in the system is enforced wherever possible, for two processes to consume the same resource two distinct phases (time units) must elapse. We will build upon the discussion in Section 31.2 to highlight similarities and differences between ACSR and other real-time process algebras.

31.3.1 Syntax and Semantics

Execution steps. ACSR employs a different execution model compared to other real-time process algebras. We have previously seen that processes in most process algebras execute events that are considered instantaneous and, in between events, time can advance while processes are “doing nothing.” In ACSR, processes can also engage in instantaneous events. The CCS-style of communications is adopted for such events which can be observable, such as sending or receiving a message, or the internal τ action. However, it assumes that processes can engage in internal computation that takes nontrivial amount of time relative to the size of the clock tick. These timed actions involve the consumption of resources, or explicit idling. Consequently, ACSR distinguishes between events and actions. Just like the events that we have considered above, ACSR events are logically instantaneous and do not require computational resources to execute. An action, by contrast, takes one unit of time and requires a set of resources to complete. We consider the details below.

Timed actions. We consider a system to be composed of a finite set of serially reusable resources denoted by \mathcal{R} . An action that consumes one “tick” of time is a set of pairs of the form (r, p) , where r is a resource and p , a natural number, is the priority of the resource use, with the restriction that each resource be represented at most once. As an example, the singleton action, $\{(cpu, 2)\}$, denotes the use of some resource $cpu \in \mathcal{R}$ running at priority level 2. The action \emptyset represents idling for 1 time unit, since no resource is consumed.

We use \mathcal{D}_R to denote the domain of timed actions, and we let A, B range over \mathcal{D}_R . We define $\rho(A)$ to be the set of resources used in the action A ; e.g., $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$.

Instantaneous events. Instantaneous events provide the basic synchronization mechanism in ACSR. We assume a set of channels L . An event is denoted by a pair (a, p) , where a is the *label* of the event and p its *priority*. Labels are drawn from the set $L \cup \bar{L} \cup \{\tau\}$, where for all $a \in L$, $a^\# \in \bar{L}$, and $a! \in \bar{L}$. We say that $a^\#$ and $a!$ are *inverse* labels. As in CCS, the special identity label τ arises when two events with inverse labels are executed in parallel.

We use \mathcal{D}_E to denote the domain of events, and let e, f range over \mathcal{D}_E . We use $l(e)$ to represent the label of event e . The entire domain of actions is $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α and β range over \mathcal{D} .

ACSR operators. The following grammar describes the syntax of PACSR processes:

$$\begin{aligned} P ::= & \text{NIL} \mid A : P \mid e \cdot P \mid P + P \mid P \parallel P \mid \\ & P \Delta_t^a(P, P, P) \mid P \setminus F \mid [P]_I \mid P \setminus\!\!\setminus I \mid C \end{aligned}$$

Semantics for ACSR is set up in two steps. First, the *unprioritized* semantics define steps of a process by means of structural rules, as we have seen with the other process algebras above. Second, the *prioritized* semantics interpret priorities within events and actions by applying a *preemption relation*, disabling low-priority events and actions in the presence of higher-priority ones. The rules of the unprioritized semantics are given in Table 31.2.

Process NIL is a deadlocked process that blocks the progress of time. Event prefix $e.P$ is similar to the event prefix of ATP and cannot pass time. Action prefix $A : P$ is similar to the time prefix of TPL, except that a distinct action A is executed instead of ϵ . Process $P + Q$ offers the nondeterministic choice between P and Q . This choice is resolved with the first action taken by one of the two processes.

The process $P \parallel Q$ describes the concurrent composition of P and Q : the component processes may proceed independently or interact with one another while executing events, and they synchronize on

TABLE 31.2 The Nonprioritized Relation

(Act1)	$e.P \xrightarrow{e} P$	(Act2)	$A : P \xrightarrow{A} P$
(Sum1)	$\frac{P_1 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$	(Sum2)	$\frac{P_2 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$
(Par1)	$\frac{P_1 \xrightarrow{e} P'_1}{P_1 \parallel P_2 \xrightarrow{e} P'_1 \parallel P_2}$	(Par2)	$\frac{P_2 \xrightarrow{e} P'_2}{P_1 \parallel P_2 \xrightarrow{e} P_1 \parallel P'_2}$
(Par3)	$\frac{P_1 \xrightarrow{(a!,n)} P'_1, P_2 \xrightarrow{(a!,n)} P'_2}{P_1 \parallel P_2 \xrightarrow{(\tau, n+m)} P'_1 \parallel P'_2}$		
(Par4)	$\frac{P_1 \xrightarrow{A_1} P'_1, P_2 \xrightarrow{A_2} P'_2, \rho(A_1) \cap \rho(A_2) = \emptyset}{P_1 \parallel P_2 \xrightarrow{A_1 \cup A_2} P'_1 \parallel P'_2}$		
(Res1)	$\frac{P \xrightarrow{e} P', l(e) \notin F}{P \setminus F \xrightarrow{e} P' \setminus F}$	(Res2)	$\frac{P \xrightarrow{A} P'}{P \setminus F \xrightarrow{A} P' \setminus F}$
(Cl1)	$\frac{P \xrightarrow{A_1} P', A_2 = \{(r, 0) \mid r \in I - \rho(A_1)\}}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I}$	(Cl2)	$\frac{P \xrightarrow{e} P'}{[P]_I \xrightarrow{e} [P']_I}$
(Hide1)	$\frac{P \xrightarrow{A} P', A' = \{(r, n) \in A \mid r \notin I\}}{P \setminus\!\!\setminus I \xrightarrow{A'} P' \setminus\!\!\setminus I}$	(Hide2)	$\frac{P \xrightarrow{e} P'}{P \setminus\!\!\setminus I \xrightarrow{e} P' \setminus\!\!\setminus I}$
(Sc1)	$\frac{P \xrightarrow{e} P', l(e) \neq b!, t > 0}{P \triangle_t^b (Q, R, S) \xrightarrow{e} P' \triangle_t^b (Q, R, S)}$	(Sc2)	$\frac{P \xrightarrow{(b!,n)} P', t > 0}{P \triangle_t^b (Q, R, S) \xrightarrow{(\tau, n)} Q}$
(Sc3)	$\frac{P \xrightarrow{A} P', t > 0}{P \triangle_t^b (Q, R, S) \xrightarrow{A} P' \triangle_{t-1}^b (Q, R, S)}$	(Sc4)	$\frac{R \xrightarrow{\alpha} R', t = 0}{P \triangle_t^b (Q, R, S) \xrightarrow{\alpha} R'}$
(Sc5)	$\frac{S \xrightarrow{\alpha} S', t > 0}{P \triangle_t^b (Q, R, S) \xrightarrow{\alpha} S'}$	(Rec)	$\frac{P \xrightarrow{\alpha} P', C \stackrel{\text{def}}{=} P}{C \xrightarrow{\alpha} P'}$

timed actions. Specifically, rules (Par1) through (Par3) implement CCS-style communication in the model, with the first two rules describing the asynchronous execution of events, and the latter describing synchronization on matching events. Of particular interest to the algebra is rule (Par4). We first observe that time progresses synchronously in a complex system; that is, for time to pass all of the concurrent components must be willing to perform a timed action. However, note that such progress is resource constrained: as expressed in the side condition of the rule, at most one process may use a given resource in any time unit. A consequence of this side condition is that whenever two (or more) concurrent processes are competing for the use of the same resource and neither is willing to engage in alternative behavior, then the system is deadlocked. This fact plays a significant role in the algebra and it is exploited for performing schedulability analysis. For example, process $\{(cpu, 1), (mem, 2)\} : P \parallel \{(cpu, 2)\} : Q$ has no outgoing transitions since $\{(cpu, 1), (mem, 2)\} \cap \{(cpu, 2)\} \neq \emptyset$. On the contrary,

$$\{(cpu_1, 1), (mem, 2)\} : P \parallel (\{(cpu_1, 2)\} : Q_1 + \{cpu_2, 1\} : Q_2) \xrightarrow{\{(cpu_1, 1), (mem, 2), (cpu_2, 1)\}} P \parallel Q_2$$

The scope construct, $P \triangle_t^a (Q, R, S)$, binds the process P by a temporal scope and incorporates the notions of timeout and interrupts. We call t the *time bound*, where $t \in \mathbf{N} \cup \{\infty\}$ and require that P may execute for a maximum of t time units. The scope may be exited in one of the three ways: first, if P terminates successfully within t time units by executing an event labeled $a!$, where $a \in L$, then control is

delegated to Q , the success handler. Else, if P fails to terminate within time t , then control proceeds to R . Finally, throughout the execution of this process, P may be interrupted by process S .

As an example, consider the task specification $T \stackrel{\text{def}}{=} R \Delta_{10}^a (SH, EH, IN)$, where

$$\begin{aligned} R &\stackrel{\text{def}}{=} (in?, 1).(a!, 2).NIL + \emptyset : R \\ SH &\stackrel{\text{def}}{=} (ack!, 1).T \\ EH &\stackrel{\text{def}}{=} (nack!, 1).T \\ IN &\stackrel{\text{def}}{=} (kill?, 3).NIL \end{aligned}$$

This task awaits an input request to arrive for a 10 time-unit period. If such an event takes place, it signals on channel a the arrival and the success handler process, SH , acknowledges the event. If the deadline elapses without the appearance of the event, the task signals the lack of input on channel $nack$. Finally, at any point during its computation, the task may receive a signal on channel $kill$ and halt its computation. According to the rules for scope, process $R \Delta_{10}^a (SH, EH, IN)$ may engage in the following actions:

$$\begin{aligned} R \Delta_{10}^a (SH, EH, IN) &\xrightarrow{\emptyset} R \Delta_9^a (SH, EH, IN) \\ R \Delta_{10}^a (SH, EH, IN) &\xrightarrow{(in?, 1)} ((a!, 2).NIL) \Delta_{10}^a (SH, EH, IN) \\ R \Delta_{10}^a (SH, EH, IN) &\xrightarrow{(kill?, 3)} NIL \end{aligned}$$

Furthermore, note that

$$\begin{aligned} R \Delta_0^a (SH, EH, IN) &\xrightarrow{(nack!, 1)} T \\ ((a!, 2).NIL) \Delta_{10}^a (SH, EH, IN) &\xrightarrow{(\tau, 2)} SH \end{aligned}$$

The Close operator, $[P]_I$, produces a process P that monopolizes the resources in $I \subseteq \mathcal{R}$. Rules (Cl1) and (Cl2) describe the behavior of the close operator. When a process P is embedded in a closed context such as $[P]_I$, we ensure that there is no further sharing of the resources in I . Assume that P executes a time-consuming action A . If A utilizes less than the full resource set I , the action is augmented with $(r, 0)$ pairs for each unused resource $r \in I - \rho(A)$. The way to interpret Close is as follows. A process may idle in two ways: it may either release its resources during the idle time (represented by \emptyset), or it may hold them. Close ensures that the resources are held. Instantaneous events are not affected. Thus,

$$\begin{aligned} [\emptyset : P_1 + \{(cpu, 1)\} : P_2]_{cpu} &\xrightarrow{\{(cpu, 0)\}} P_1 \\ [\emptyset : P_1 + \{(cpu, 1)\} : P_2]_{cpu} &\xrightarrow{\{(cpu, 1)\}} P_2 \end{aligned}$$

The resource hiding operator, $P \setminus I$, internalizes the set of resources I within the process P . As shown in (Hide1) and (Hide2), information regarding usage of resources in I is removed from timed actions of P while instantaneous events remain unaffected. Finally, the restriction operator $P \setminus F$ and process constants C are defined in the usual way.

In subsequent sections, we will assume the presence of a special constant $IDLE \in \mathcal{C}$, representing the process that idles forever, defined by $IDLE \stackrel{\text{def}}{=} \emptyset : IDLE$. Further, we will be using the shorthand notation $P \Delta_t (Q)$ for $P \Delta_t^a (NIL, Q, NIL)$ to denote the process that executes as P for the first t time units and then proceeds as Q . Finally, we write $\Pi_{i \in I} P_i$ for the parallel composition of processes P_i , $i \in I$.

An important observation to make about the semantic rules is that the only “source” of time progress is the action-prefix operator. That is, time progress has to be explicitly encoded into the model by the designer. This feature forces the designer to think carefully about time progress in the model and results in fewer unexpected behaviors in a complex model.

Another important observation is that time determinism, which is one of the main underlying principles for most real-time process algebras, loses its importance in ACSR. Specifically, time determinism requires that if $P \xrightarrow{\epsilon} P_1$ and $P \xrightarrow{\epsilon} P_2$, then $P_1 = P_2$. That is, because a process is, effectively, idle while time advances, one tick should be indistinguishable from another. However, in ACSR, a process may be able to perform different timed actions and evolve to different states: $P \xrightarrow{A_1} P_1$ and $P \xrightarrow{A_2} P_2$, with $P_1 \neq P_2$. Since actions may be blocked by other processes with higher resource-access priorities, it is important to develop models in such a way that a process offers alternative behaviors in case the main one is blocked. For example, a process P that wants to execute an action A but is willing to be preempted can be expressed as $P \stackrel{\text{def}}{=} A : P' + \emptyset : P$. The action \emptyset , which does not require any resources, allows the process P to idle for 1 time unit and then retry to execute A .

Preemption and prioritized transitions. The prioritized transition system is based on *preemption*, which incorporates the ACSR treatment of priority. This is based on a transitive, irreflexive, binary relation on actions, $<$, called the *preemption relation*. If $\alpha < \beta$, for two actions α and β , we say that α is *preempted by* β . Then, in any process, if there is a choice between executing either α or β , β will always be executed. We refer to Ref. 11 for the precise definition of $<$. Here, we briefly describe the three cases for which $\alpha < \beta$ is deemed to be true by the definition.

- The first case is for two timed actions α and β which compete for common resources. Here, it must be that the preempting action β employs all its resources at priority level at least the same as α . Also, β must use at least one resource at a higher level. It is still permitted for α to contain resources not in β but all such resources must be employed at priority level 0. Otherwise, the two timed actions are incomparable. Note that β cannot preempt an action α consuming a *strict subset* of its resources at the same or lower level. This is necessary for preserving the compositionality of the parallel operator. For instance, $\{(r_1, 2), (r_2, 0)\} < \{(r_1, 7)\}$ but $\{(r_1, 2), (r_2, 1)\} \not< \{(r_1, 7)\}$.
- The second case is for two events with the same label. Here, an event may be preempted by another event with the same label but a higher priority. For example, $(\tau, 1) < (\tau, 2)$, $(a, 2) < (a, 5)$, and $(a, 1) \not< (b, 2)$ if $a \neq b$.
- The third case is when an event and a timed action are comparable under “ $<$.” Here, if $n > 0$ in an event (τ, n) , we let the event preempt any timed action. For instance, $\{(r_1, 2), (r_2, 5)\} < (\tau, 2)$, but $\{(r_1, 2), (r_2, 5)\} \not< (\tau, 0)$.

We define the prioritized transition system “ $\xrightarrow{\pi}$,” which simply refines “ $\xrightarrow{\epsilon}$ ” to account for preemption as follows: $P \xrightarrow{\alpha}_{\pi} P'$ if and only if (1) $P \xrightarrow{\alpha} P'$ is an unprioritized transition, and (2) there is no unprioritized transition $P \xrightarrow{\beta} P''$ such that $\alpha < \beta$.

We note that the close operator has a special significance in relation to the preemption relation and the prioritized transition system. In particular, we may see that, by closing a system by the set of its resources, we enforce progress to be made: for example, consider the case when an idling action and a resource-consuming action are simultaneously enabled from the initial state of a system. By applying the close operator, the idling action is transformed into an action that consumes all resources of the system at priority level 0. Then, in the prioritized transition relation this action will become preempted by the nonidling action, and the system will be forced to progress.

31.3.2 Schedulability Analysis with ACSR

In this section, we discuss how realistic task sets can be modeled in ACSR and how to perform schedulability analysis of such task sets. A *real-time task* is a building block for a real-time system. A task is invoked in

response to an event in the system and needs to perform its computation by a certain deadline. A task invocation, then, is characterized by two parameters that are constant across all invocations of the same task: a relative deadline d and the range of execution times it takes to complete, $[c_{\min}, c_{\max}]$. The task itself can be considered as a sequence of invocations and is characterized by a *dispatch policy*, which can be periodic, aperiodic, or sporadic. A periodic task with a period p is dispatched by a timer event every p time units. Aperiodic and sporadic tasks are dispatched by events that can be raised by tasks within the system or originate from the system's environment. Sporadic tasks are characterized by the minimum separation between task invocations. For more information the reader is referred to Ref. 12. A real-time system consists of a fixed set of tasks and a set of resources that tasks share. The most important kind of resources are processors that run tasks. Although ACSR can model migration of tasks between processors, in this chapter we assume that tasks are statically allocated to processors. Other resource types include buses, shared memory blocks, etc.

In Ref. 5, modeling and analysis of several scheduling problems was introduced. Here, we apply a similar approach to a somewhat different scheduling problem. We consider real-time systems that contain periodic tasks with precedence constraints, executing on a set of processor resources without task migration. These tasks are competing between them for the use of the system's resources, access to which is resolved by a scheduler according to the task priorities. These priorities can be static or dynamic. We assume that the systems we consider are *closed*, that is, there is no competition for system resources from outside sources. Similar task models have been considered in Refs. 18 and 20. We use the following notation: for a task T_i , let the integer constants d_i , $[c_{\min,i}, c_{\max,i}]$, and p_i denote the deadline, execution time range, and period of the task, respectively. Resource cpu_i denotes the processor resource to which T_i is allocated. Note that, if more than one tasks are allocated to a processor, then $cpu_i = cpu_j$ for some i and j .

We assume a set of precedence constraints between the system tasks representing data or control dependencies that exist between them. These constraints are of the form

$$w_{ij} : T_i \xrightarrow{f_{ij}} T_j$$

where f_{ij} is an integer constant representing a propagation delay. We require that the precedence graph be acyclic to avoid deadlocks during execution. We also use the simplifying assumption that for every constraint w_{ij} tasks T_i and T_j have harmonic periods; that is, either p_i is a multiple of p_j or vice versa. Let $t_{j,n}$ be the n th invocation of task T_j . Then, a constraint $w_{ij} : T_i \xrightarrow{f_{ij}} T_j$ with $p_j = k \cdot p_i$ (i.e., T_i executes more often than T_j) is interpreted as follows: $t_{j,n}$ cannot begin execution until f_{ij} time units later than $t_{i,(n-1) \cdot k + 1}$ completes its execution. However, if $w_{ij} : T_i \xrightarrow{f_{ij}} T_j$ with $p_i = k \cdot p_j$, then $t_{j,n}$ cannot begin execution until f_{ij} time units later than $t_{j,n \bmod k}$ completes its execution.

As we present the modeling approach, it will become clear that this task model can be easily generalized in several ways, at the cost of a slightly more complex ACSR encoding. More general constraints can be used, the requirement for harmonic periods for interacting tasks can be eliminated, and other dispatch policies can be represented. We also note that the encoding of tasks and constraints can be automatically generated from higher-level architectural representations such as AADL, as reported in Ref. 23.

Example

To illustrate the scheduling problem described above, consider the example shown in Figure 31.1a. The system consists of four tasks with constant execution times ($c = c_{\min} = c_{\max}$) and deadlines equal to task periods. Constraints are represented as a graph, with tasks as graph nodes and constraints as directed edges. Figure 31.1b presents the schedule of a system execution with the rate-monotonic priority assignment. (Tasks are assigned static priorities according to their periods, the shorter the period the higher the priority, and scheduling is preemptive.) Note that even though task T_4 has the highest priority, it barely meets its deadline owing to a precedence dependency on a lower-priority T_2 , which is preempted by T_1 . Note also that only the first invocation of T_4 is affected by T_2 , and only the first invocation of T_1 affects T_3 .

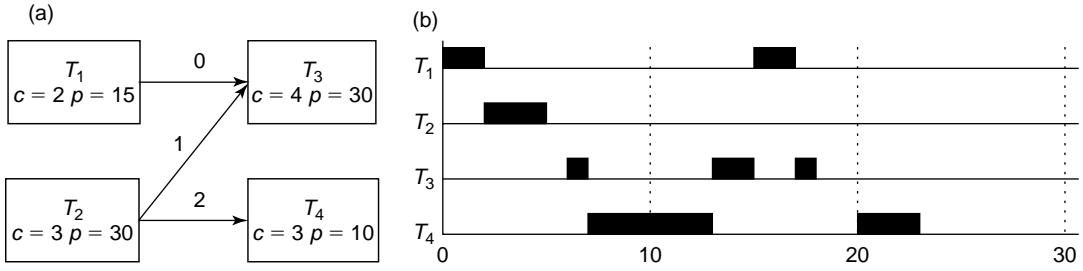


FIGURE 31.1 Task set with precedence constraints.

We now proceed with the model of the scheduling problem. We do this in a bottom-up approach, beginning with the modeling of tasks and building up to the modeling of an entire system.

Encoding of task execution. The execution of a task T_i goes through several states. Initially, T_i is sleeping, represented by the ACSR process $Sleep_i$. This process can idle indefinitely, waiting for the environment to offer the $dispatch_i$ event. When this event occurs, the process proceeds to the next state $Resolve_i$, where it waits until all precedence constraints are met. When that happens, the environment offers the $csat_i$ event, which will allow the task to begin its execution. Execution is modeled by a parameterized collection of ACSR processes $Exec_{i,c}$. Intuitively, process $Exec_{i,c}$ represents the execution when the task invocation has accumulated c time units of computation. While $c < c_{i,max}$, $Exec_{i,c}$ can gain access to the resource cpu_i , and by doing this it advances to $Exec_{i,c+1}$. Alternatively, the task may be preempted, in which case $Exec_{i,c}$ follows the self-loop labeled with the idling action. In addition, if $c \geq c_{i,min}$, $Exec_{i,c}$ can send event $done_i$ and return to state $Sleep_i$. Once $c = c_{i,max}$, the process may only offer the event $done_i$ until it becomes accepted. The ACSR processes for the task execution are shown below:

$$\begin{aligned}
 Sleep_i &\stackrel{\text{def}}{=} \emptyset : Sleep_i + (dispatch_i?, 0).Resolve_i \\
 Resolve_i &\stackrel{\text{def}}{=} \emptyset : Resolve_i + (csat_i?, 0).Exec_{i,0} \\
 Exec_{i,c} &\stackrel{\text{def}}{=} \emptyset : Exec_{i,c} + \{(cpu_i, pr_i)\} : Exec_{i,c+1} && c < c_{i,min} \\
 Exec_{i,c} &\stackrel{\text{def}}{=} \emptyset : Exec_{i,c} + \{(cpu_i, pr_i)\} : Exec_{i,c+1} + (done_i!, pr_i).Sleep_i && c_{i,min} \leq c < c_{i,max} \\
 Exec_{i,c} &\stackrel{\text{def}}{=} \emptyset : Exec_{i,c} + (done_i!, pr_i).Sleep_i && c = c_{i,max}
 \end{aligned}$$

Task activators. For each task T_i , the event $dispatch_i$ is supplied by a process acting as a *task activator*. The activator process serves two purposes: it handles thread dispatch and keeps track of the deadline of the current thread invocation. Here we show a simple activator process for a periodic thread T_i whose deadline is equal to its period ($d_i = p_i$).

$$\begin{aligned}
 Activator_i &\stackrel{\text{def}}{=} (dispatch_i!, pr_i).Wait \Delta_{p_i}(Activator_i) \\
 Wait &\stackrel{\text{def}}{=} (done?, pr_i).Signal_i + \emptyset : Wait
 \end{aligned}$$

The process sends the dispatch event to the task process, initiating execution of the task invocation. Then, the activator awaits p_i time units for the task to signal its completion on channel $done$ in which case it continues as process $Signal_i$, explained in the next paragraph. Note that, as soon as the task is ready to emit an event labeled with $done$ this will be immediately received by $Wait$. This is ensured by the preemption relation that enforces maximal progress in this sense by preempting timed actions over internal communications. Once the p_i time units elapse, the $Activator_i$ resends the dispatch signal. By that time, the thread should have completed its activation and be able to accept the dispatch signal. Otherwise,

a deadline violation is detected and the activator process is blocked since, in its initial state, it cannot pass time. The resulting deadlocked state in the state space of the model allows us to detect the missed deadline.

Encoding of constraints. The set of precedence constraints of the system is handled by associating two more processes with each task in the model. The first is responsible for *releasing* all tasks that are waiting for the task to complete the execution, and the latter is for detecting the release of all constraints on which its own invocation depends. So, suppose that task T_i is involved in a set of constraints $W_1 = \{w_{i,j} \mid j \in J\}$, and a set of constraints $W_2 = \{w_{m,i} \mid m \in M\}$. Let us begin by considering constraints W_1 . For each such constraint, one should detect the completion of task T_i and send appropriate messages to release the relevant task. The detection of the completion is implemented by process $Signal_i$, the component of $Activator_i$ mentioned above, as follows:

$$Signal_i \stackrel{\text{def}}{=} (fin_i!, pr_i).Signal_i + \emptyset : Signal_i$$

This process is enabled until the next invocation of $Task_i$ and emits the event fin_i signaling the completion of an invocation. Such a completion may have different importance for difference constraints. To capture this, for each constraint $w_{i,j} \in W_1$, we define a process $Release_{i,j}$ that determines whether the constraint should be released and emits a message to task j . Representation of $Release_{i,j}$ depends on the relationship between the periods of T_i and T_j . In the case that $p_j = k \cdot p_i$, $Release_{i,j}$ awaits the completion of task T_i and emits a message to task T_j by offering $done_{i,j}$. It repeats this behavior every p_j time units and ignores intermediate completions of invocations:

$$\begin{aligned} Release_{i,j} &\stackrel{\text{def}}{=} Rel_{i,j} \triangle_{p_j} (Release_{i,j}) \\ Rel_{i,j} &\stackrel{\text{def}}{=} (fin_i?, pr_i).Send_{i,j} + \emptyset : Rel_{i,j} \\ Send_{i,j} &\stackrel{\text{def}}{=} (done_{i,j}!, pr_i).IDLE + \emptyset : Send_{i,j} \end{aligned}$$

The case of $p_i = k \cdot p_j$ is very similar with the exception that $Release_{i,j}$ repeats this behavior every p_i time units.

Let us now consider constraints W_2 for a task T_i . These are handled by an ACSR process $Constr_i$. The process itself consists of a number of concurrent processes: a set of processes $Prop_{j,i}$, for each $w_{j,i} \in W_2$, which detect the release of the constraint $w_{j,i}$, and a process $Poll_i$ that queries all the $Prop_{j,i}$ processes and sends the $csat_i$ event if all constraints are released. Communication between these concurrent processes is internal to $Constr_i$

$$Constr_i \stackrel{\text{def}}{=} (Poll_i \parallel \Pi_j Prop_{j,i}) \setminus \{ready_i\}$$

The process $Poll_i$ runs every time unit and sequentially tries to synchronize with every process $Prop_{j,i}$ on event $ready_i$. If any of the processes cannot offer $ready_i$, this indicates that the constraint $w_{j,i}$ is not satisfied. Let us assume that $|W_2| = m$. We have

$$\begin{aligned} Poll_i &\stackrel{\text{def}}{=} Poll_{i,0} \\ Poll_{i,j} &\stackrel{\text{def}}{=} (ready_i?, 0).Poll_{i,j+1} + \emptyset : Poll_i \quad j < m \\ Poll_{i,m} &\stackrel{\text{def}}{=} (csat_i!, pr_i).\emptyset : Poll_i + \emptyset : Poll_i \end{aligned}$$

Representation of $Prop_{j,i}$ depends on the relationship between the periods of T_i and T_j . Let us begin with the case that $p_i = k \cdot p_j$. In this case, $Prop_{j,i}$ awaits a signal from process $Release_{j,i}$ and, after $f_{j,i}$ time

units, it declares the release of the task to the polling process by sending the event $ready_i$. It repeats this behavior every p_i time units:

$$\begin{aligned} Prop_{i,j} &\stackrel{\text{def}}{=} P_{i,j} \triangle_{p_i} (Prop_{i,j}) \\ P_{i,j} &\stackrel{\text{def}}{=} (done_{i,j}?, pr_i).IDLE \triangle_{f_{j,i}} (Ready_{i,j}) + \emptyset : P_{i,j} \\ Ready_{i,j} &\stackrel{\text{def}}{=} \emptyset : Ready_{i,j} + (ready_{j,l}!, pr_{i,l}).Ready_{i,j} \end{aligned}$$

The case when $p_j = k \cdot p_i$ is handled in a similar way, only the behavior of $Prop_{i,j}$ is repeated every p_j time units.

Priority assignments and final composition. We are now ready to put together the processes for tasks and constraints into a coherent model. We first form the “task cluster” that brings together the processes that relate to the single task, and make communication between them internal to the cluster. The four processes involved in a cluster for the task T_i are $Sleep_i$, $Activator_i$, $Release_i$, and $Constr_i$. They communicate by events $dispatch_i$, $done_i$, fin_i , and $csat_i$. We thus get the following expression for the task cluster process

$$Task_i \stackrel{\text{def}}{=} (Sleep_i \| Activator_i \| Constr_i \| Release_i) \setminus \{dispatch_i, done_i, csat_i, fin_i\}$$

It remains to compose task clusters together, and make all communication on events $done_{i,j}$ internal. In addition, we need to close all processor resources so that no other tasks can use them and to enforce progress in the system.

$$RTSystem \stackrel{\text{def}}{=} [(\Pi_{i=1, \dots, n} Task_i) \setminus \{done_{i,j} | i, j = 1, \dots, n\}]_{\{cpu_i | i=1, \dots, n\}}$$

In order for this model to execute correctly, we need to carefully assign priorities in actions and events used in the model. Priorities of resource access represent the scheduling policy used to schedule processors in the system. In the case of static priority scheduling, all actions in all processes $Exec_{i,c}$ have the same priority, which is the priority statically assigned to thread T_i . Dynamic-priority scheduling policies, such as earliest deadline first (EDF) policy, makes the resource priority in $Exec_{i,c}$ a function of d_i and c .

On the Contrary, event priorities do not play an important role in the specific model. We may observe that any process may engage in at most one event at any given point in time. Thus, although the execution of more than one internal event may be possible in any time-phase (between two timed actions) in the global system, if a number of internal events are simultaneously enabled in some state of the system, execution of one event does not affect the execution of another, since, by construction, this must be taking place in a separate component of the system. In addition, note that all such events will take place confluenty *before* the next timed action due to the preemption relation preempting timed events by internal actions.

Schedulability analysis. Within the ACSR formalism we can conduct two types of analysis for real-time scheduling: validation and schedulability analysis. Validation shows that a given specification correctly models the required real-time scheduling discipline, such as Rate Monotonic and EDF. Schedulability analysis determines whether or not a real-time system with a particular scheduling discipline misses any of its deadlines. In the ACSR modeling approach, a missed deadline induces a deadlock in the system. The correctness criterion, then, can be stated as the absence of deadlocks in the model of the system. This can be checked by using state-space exploration techniques. Specifically, we can show that an instance of the problem is schedulable if and only if its associated $RTSystem$ process contains no deadlocks. The following theorem pinpoints the source of deadlocks in a certain class of systems, including process $RTSystem$ above, and enables it to arrive at the desired conclusion.

First, let us introduce some useful notation. In the sequel, we will write $P \xrightarrow{\alpha}_{\pi}$ if there is some process P' such that $P \xrightarrow{\alpha} P'$, and $P \not\xrightarrow{\alpha}_{\pi}$ if there is no process P' such that $P \xrightarrow{\alpha}_{\pi} P'$. Further, we will

write $P \Longrightarrow_{\pi} P'$ if there is a sequence of transitions of the form $P \xrightarrow{\alpha_1}_{\pi} P_1 \xrightarrow{\alpha_2}_{\pi} \dots \xrightarrow{\alpha_n}_{\pi} P_n = P'$, in which case we say that P' is a *derivative* of P and that the *length* of the transition $P \Longrightarrow_{\pi} P'$ is n .

Theorem 31.1

Consider a process P such that

$$P \stackrel{\text{def}}{=} ([(\Pi_{i \in I} P_i) \setminus F]_U) \setminus V$$

where $F \subseteq L$, $U, V \subseteq \mathcal{R}$, for all i , P_i contains no parallel composition operator and $I = I_1 \cup I_2$, where

- for all $i \in I_1$ and for all derivatives Q_i of P_i , $Q_i \xrightarrow{\emptyset}_{\pi}$, and
- for all $i \in I_2$ and for all derivatives Q_i of P_i , either (1) $Q_i \xrightarrow{\emptyset}_{\pi}$ or (2) $Q_i \xrightarrow{\alpha_i}_{\pi}$, $\ell(\alpha_i) \in F$, and $Q_i \not\xrightarrow{\beta}_{\pi}$ for all $\beta \in \mathcal{D} - \{\alpha_i\}$.

Then, for all derivatives Q of P , if $Q \not\xrightarrow{}_{\pi}$ then $Q = ([(\Pi_{i \in I} Q_i) \setminus F]_U) \setminus V$ and $Q_i \xrightarrow{\alpha_i}_{\pi}$ for some $i \in I_2$.

Proof

Consider a process $P \stackrel{\text{def}}{=} ([(\Pi_{i \in I} P_i) \setminus F]_U) \setminus V$ satisfying the conditions of the theorem. We will prove that any derivative Q of P is such that $Q = ([(\Pi_{i \in I} Q_i) \setminus F]_U) \setminus V$, where no Q_i contains a parallel composition operator, and I can be partitioned into sets I_1 and I_2 satisfying the conditions of the theorem. The proof will be carried out by induction on the length, n , of the transition $P \Longrightarrow_{\pi} Q$.

Clearly, the claim holds for $n = 0$. Suppose that it holds for $n = k - 1$ and that $P \Longrightarrow_{\pi} Q' \xrightarrow{\alpha}_{\pi} Q$ is a transition of size n . By the induction hypothesis, $Q' = ([(\Pi_{i \in I} Q'_i) \setminus F]_U) \setminus V$ satisfies the conditions of the theorem. Consider the transition $Q' \xrightarrow{\alpha}_{\pi} Q$. Three cases exist:

- $\alpha \in \mathcal{D}_R$. This implies that for all $i \in I$, $Q'_i \xrightarrow{A_i}_{\pi} Q_i$, for some Q_i , $Q = ([(\Pi_{i \in I} Q_i) \setminus F]_U) \setminus V$ and $\alpha = \bigcup_{i \in I} A_i$. It is straightforward to see that no Q_i contains a parallel composition operator and that, since each Q_i is a derivative of Q'_i , the conditions of the theorem are satisfied.
- $\alpha = \tau$. This implies that there exist $j, k \in I$, such that $Q'_j \xrightarrow{\alpha_j}_{\pi} Q_j$ and $Q'_k \xrightarrow{\alpha_k}_{\pi} Q_k$, where $\ell(\alpha_j)$ and $\ell(\alpha_k)$ are inverse labels, and $Q = ([(\Pi_{i \in I - \{j, k\}} Q'_i \parallel Q_j \parallel Q_k) \setminus F]_U) \setminus V$. It is straightforward to see that no Q'_i , Q_i , contains a parallel composition operator and check that the conditions of the theorem are satisfied.
- $\alpha \in \mathcal{D}_E$. This implies that there exists $j \in I$, such that $Q'_j \xrightarrow{\alpha_j}_{\pi} Q_j$, $Q = ([(\Pi_{i \in I - \{j\}} Q'_i \parallel Q_j) \setminus F]_U) \setminus V$ and the proof follows easily.

So consider an arbitrary derivative Q of P and suppose that $Q \not\xrightarrow{}_{\pi}$. Since $Q = ([(\Pi_{i \in I} Q_i) \setminus F]_U) \setminus V$ satisfies the conditions of the theorem, and further $Q \not\xrightarrow{\emptyset}_{\pi}$, it must be that some $Q_i \not\xrightarrow{\emptyset}_{\pi}$, $i \in I_2$. This implies that $Q_i \xrightarrow{\alpha_i}_{\pi}$ and the result follows.

As a corollary we obtain the desired result.

Proposition 31.1

RTSystem is schedulable if and only if it contains no deadlocks.

Proof

First, we observe that if *RTSystem* contains no deadlocks then the associated real-time system is schedulable: task activations take place as planned and no deadlines are missed.

To prove the opposite direction, we show that if the system contains a deadlock then *RTSystem* is not schedulable. Consider system *RTSystem*. By using the ACSR axiom system, we may easily rewrite this process to an equivalent process which has the form of P in Theorem 31.1, that is,

$$RTSystem = [(\Pi_{i=1 \dots n} (Sleep_i \parallel Activator_i \parallel (Poll_i \parallel \Pi_j Prop_i) \parallel Release_i)) \setminus F]_U$$

where $F = \{done_{i,j} \mid i, j = 1, \dots, n\} \cup \{dispatch_i, done_i, csat_i, ready_i, fin_i \mid i = 1, \dots, n\}$, $U = \{cpu_i \mid i = 1, \dots, n\}$.

It is straightforward to verify that the above process satisfies the conditions of Theorem 31.1, and, further, I_2 contains all processes $Activator_i$, with $\alpha_i = (dispatch_i, pr_i)$. Consequently, by the same theorem, if a deadlock arises in $RTSystem$, the event $dispatch_i$ is enabled in some process $Activator_i$ but not in the respective $Sleep_i$ process. This implies that the task has not yet completed execution of its previous activation and has missed its deadline. Thus, the system is not schedulable. This completes the proof.

Thus, the model we have described can be instantiated to a specific task set with a specific scheduling algorithm, and its schedulability can be decided by searching for deadlocks in the resulting model.

31.4 Conclusions

In this chapter, we have given an overview of how timing information can be embedded in process-algebraic frameworks. We have concentrated on illustrating how time and time passage are modeled in different process algebras, and how the resulting mechanisms interact with each other and with standard, untimed operators. We have proceeded to review the ACSR process algebra, which extends the ideas of timed process algebras to the field of timed, resource-constrained systems. We have observed that ACSR shares a number of features with other timed process algebras. For example, all the mentioned formalisms share the view that a timed system is the composition of cooperating sequential processes that synchronize on a shared global clock and operate in a sequence of two-phase steps alternating between time progress and instantaneous steps. Further, in all the formalisms the designer is called to ensure that nonZenoness and progress are preserved in the system, with respect to the idiosyncracies of the formalism.

ACSR stands out from the remainder of the process algebras considered in that it is geared toward the modeling of resource-sensitive, timed systems. In doing this, it adopts the notion of a *resource* as a first-class entity, and it replaces maximal progress, employed by other process algebras, by the notion of *resource-constrained* progress. Thus, it associates resource-usage with time passage, and it implements appropriate rules to ensure that progress in the system is enforced as far as possible while simultaneous usage of a resource by distinct processes is excluded. In addition, ACSR employs the notions of priorities to arbitrate access to resources by competing processes.

Furthermore, we have illustrated the use of ACSR for the schedulability analysis of a realistic real-time system problem. The systems in question are composed of a set of periodic tasks with precedence constraints, competing for the use of a set of processors, access to which is resolved by a scheduler according to the task priorities. Schedulability analysis of this, and other schedulability problems, is translated in the formalism into the absence of deadlocks in the ACSR process modeling the system. We have made this concrete by providing a compositional result that characterizes the source of deadlocks in a specific set of concurrent components. We have shown that, in our model, this source is associated with tasks missing their deadlines.

ACSR has been extended into a family of process algebras. Extensions and variations include GCSR [4], which allows the visual representation of ACSR processes; Dense-time ACSR [6], which includes a more general notion of time; ACSR-VP [10], which includes a value-passing capability; PACSR [19], a probabilistic formalism for quantitative reasoning about fault-tolerant properties of resource-bound systems; and P²ACSR [24], which allows to specify power-constrained systems. The PARAGON toolset [22] provides tool support for system modeling and analysis using these formalisms.

References

1. J. Baeten and C. Middelburg. Process algebra with timing: Real time and discrete time. In *Handbook of Process Algebra*, pp. 627–684. Elsevier, Amsterdam, 2001.
2. J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2): 481–529, 1991.

3. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1990.
4. H. Ben-Abdallah. *GCSR: A Graphical Language for the Specification, Refinement and Analysis of Real-Time Systems*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, 1996.
5. H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie. A process algebraic approach to the schedulability analysis of real-time systems. *Real-Time Systems*, 15(3): 189–219, 1998.
6. P. Brémont-Grégoire and I. Lee. Process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189: 179–219, 1997.
7. R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *Proceedings of CONCUR'97*, LNCS 1243, pp. 166–180, 1997.
8. M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2): 221–239, 1995.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, New York, 1985.
10. H. Kwak, I. Lee, A. Philippou, J. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proceedings of RTSS'98*, pp. 409–419, 1998.
11. I. Lee, P. Brémont-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, pp. 158–171, 1994.
12. J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, New York, 2000.
13. R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.
14. F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of CONCUR'90*, LNCS 458, pp. 401–415, 1990.
15. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of CAV'91*, LNCS 575, pp. 376–398, 1991.
16. X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1): 131–178, 1994.
17. D. Park. Concurrency and automata on infinite sequences. In *Proceedings of 5th GI Conference*, LNCS 104, pp. 167–183, 1981.
18. D. Peng, K. Shin, and T. Abdelzaher. Assignment and scheduling of communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12), 1997.
19. A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. In *Proceedings of CONCUR'98*, LNCS 1466, pp. 389–404, 1998.
20. K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4): 412–420, 1995.
21. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58: 249–261, 1988.
22. O. Sokolsky, I. Lee, and H. Ben-Abdallah. Specification and analysis of real-time systems with PARAGON. *Annals of Software Engineering*, 7: 211–234, 1999.
23. O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *Proceedings of WP-DRTS'06*, 2006.
24. O. Sokolsky, A. Philippou, I. Lee, and K. Christou. Modeling and analysis of power-aware systems. In *Proceedings of TACAS'03*, LNCS 2619, pp. 409–425, 2003.
25. W. Yi. Real time behaviour of asynchronous agents. In *Proceedings of CONCUR'90*, LNCS 458, pp. 502–520, 1990.

32

Modular Hierarchies of Models for Embedded Systems

32.1	Motivation	32-1
32.2	Comprehensive System Modeling Theory	32-3
	The Data Model: Algebras • Syntactic Interfaces of Systems and Their Components • State View: State Machines • The Interface Model • The Distributed System Model: Composed Systems	
32.3	Structuring Interfaces	32-10
	Structuring Multifunctional Systems • Relating Services	
32.4	Refinement	32-15
	Property Refinement • Compositionality of Property Refinement • Granularity Refinement: Changing Levels of Abstraction	
32.5	Composition and Combination	32-18
	Functional Enhancement: Combining Services • Architecture Design: Composition of Interfaces	
32.6	Modeling Time	32-20
	Changing the Timescale • Rules for Timescale Refinement	
32.7	Perspective, Related Work, Summary, and Outlook	32-23
	Perspective • Related Work • Summary and Outlook	

Manfred Broy

Technische Universität München

32.1 Motivation

Development of software in general and, in particular, of embedded software is today one of the most complex but at the same time most effective tasks in the engineering of innovative applications. Software drives innovation in many application domains. Modern software systems typically are embedded in technical or organizational processes, distributed, dynamic, and accessed concurrently by a variety of independent user interfaces. Just by constructing the fitting software we can provide engineering artifacts that can calculate results, communicate messages, control devices, and illustrate and animate all kinds of information (Figure 32.1).

Making embedded systems and their development more accessible and reducing their complexity both for their development, operation, and maintenance we use classical concepts from engineering, namely abstraction, separation of concerns, and appropriate ways of structuring software. Well-chosen models and their theories support such concepts. Also software development can be based on models of system behavior and since well-chosen models are a successful way to understand software and hardware development, modeling is an essential and crucial issue in software construction.

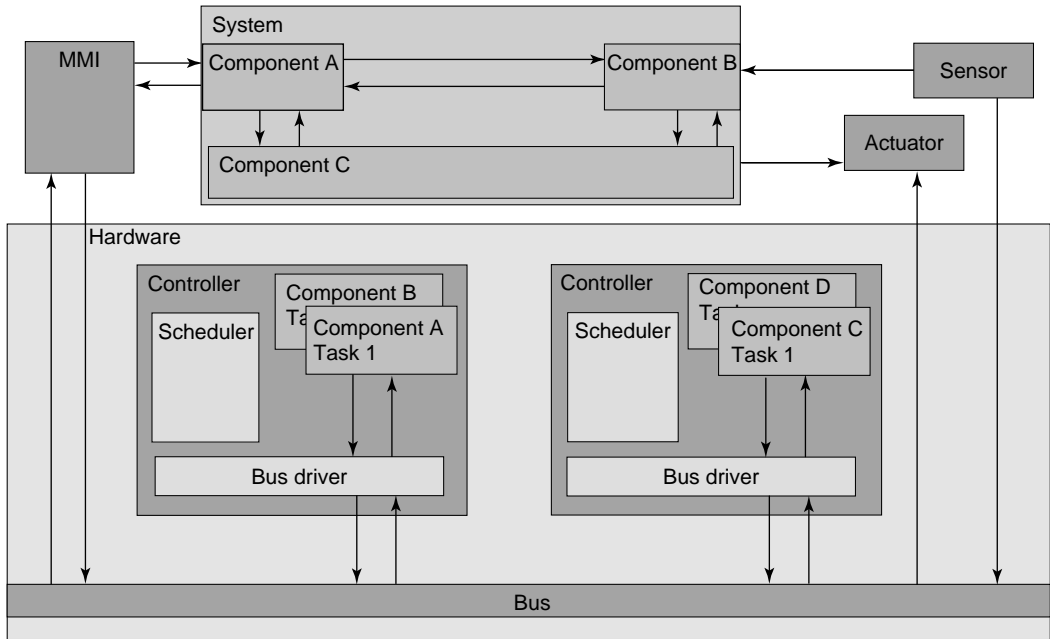


FIGURE 32.1 Schematic architecture of an embedded system. Top: an abstract view; bottom: a concrete view.

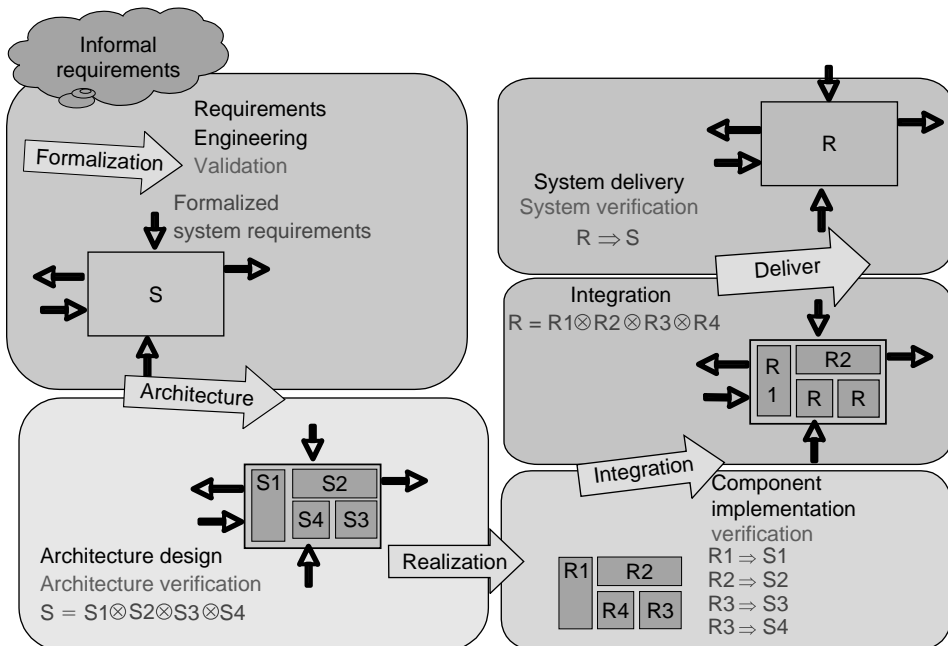


FIGURE 32.2 Idealized modular development.

In the development of large complex software systems it is simply impossible to provide one comprehensive model for the system in only one step. Rather we

- Specify a system or subsystem first in terms of its functionality modeled by its interface, given in a structured way in terms of a function hierarchy.

- Add stepwise details by property refinement.
- Provide several views onto the system in terms of states, architecture, and processes.
- Decompose the hierarchical system into components.
- Construct a sequence of such models on different levels of abstraction.

Each step in these activities introduces models, refines them, verifies, or integrates them. Figure 32.2 shows an idealized process based on modeling.

The most remarkable issue is the independence of the architecture verification from the implementation, the modularity (mutual independency) of the component implementation, and the guaranteed correctness of the integration, which follows as a theorem from the correctness of the architecture and the correctness of each of the realizations of the components.

32.2 Comprehensive System Modeling Theory

In this section we introduce a comprehensive system model. It aims at distributed systems that interact by asynchronous message exchange in a time frame. It introduces the following views onto systems:

- The *data* view that introduces the fundamental data types.
- The *syntactic interface* view that shows by which events a system may interact with its environment.
- The *state* view where systems are modeled by state machines with input and output.
- The *semantic interface* view where systems are modeled in terms of their interaction with their environment.
- The *architecture* view where systems are modeled in terms of their structuring into components.
- The *service* view where the overall functionality of a large system is structured into a hierarchy of services and subservices.

We introduce mathematical models to represent these views. In the following chapters we show how these views are related.

It is to be seen as the basis of a theory of modeling for the engineering of software-intensive systems. Figure 32.3 gives an overview on these views in terms of the models to represent them.

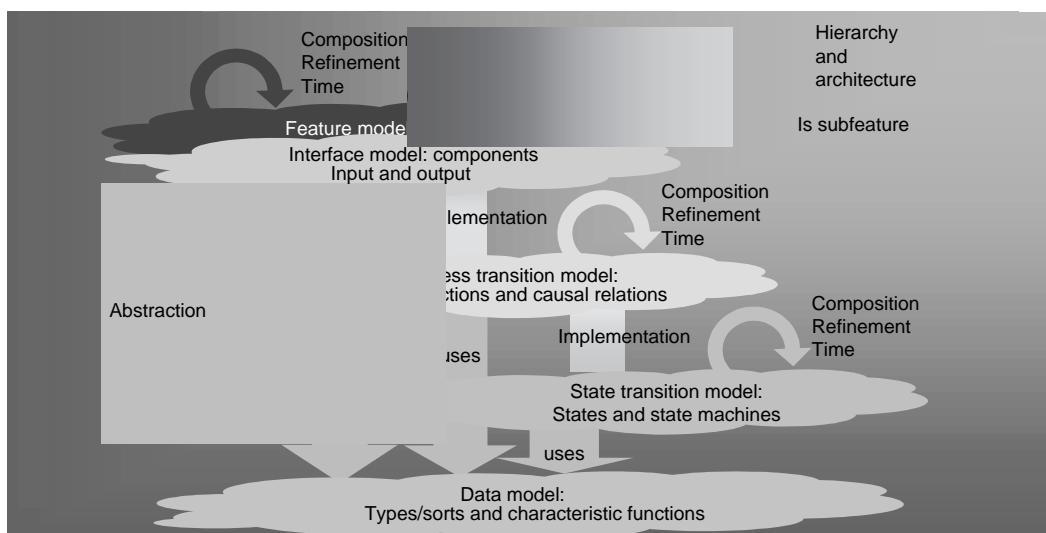


FIGURE 32.3 The structure of modeling elements.

32.2.1 The Data Model: Algebras

We believe, like many others, that data types (typing) are a very helpful concept in a structured modeling of application domains and software structures (for details, see Ref. 11). From a mathematical point of view, a data model consists of a *heterogeneous algebra*. Such algebras are given by families of *carrier sets* and families of *functions* (including predicates and thus relations). More technically, we assume a set TYPE of types (sometimes also called *sorts*).

Given types we consider a set FUNCT of function symbols with a predefined functionality (TYPE* stands for the set of finite sequences over the set of types TYPE)

$$\mathbf{fct} : \text{FUNCT} \rightarrow (\text{TYPE}^* \times \text{TYPE})$$

The function **fct** associates with every function symbol in FUNCT its domain types and its range types. Both the sets TYPE and FUNCT provide only names for sets and functions. The pair (TYPE, FUNCT) together with the type assignment **fct** is often called the *signature* of the algebra. The signature is the *static* (also called *syntactic*) *part* of a data model. Every algebra of signature (TYPE, FUNCT) provides a carrier set (a set of data elements) for every type and a function of the requested functionality for every function symbol. For each type $T \in \text{TYPE}$, we denote by $\text{CAR}(T)$ its carrier set. There are many ways to describe data models such as algebraic specifications, E/R diagrams (see Ref. 27), or class diagrams.

32.2.2 Syntactic Interfaces of Systems and Their Components

A system and also a system component is an active information processing unit that encapsulates a state and communicates asynchronously with its environment through its interface syntactically characterized by a set of input and output channels. This communication takes place within a global (discrete) time frame. In this section we introduce the notion of a syntactic interface of systems and system components. The syntactic interface models by which communication lines, which we call channels, the system or a system component is connected to the environment and which messages are communicated over the channels. We distinguish between input and output channels.

The channels and their messages determine the events of interactions that are possible for a system or a system component. In the following sections we introduce several views such as state machines, semantic interfaces, and architectures that all fit into the syntactic interface view. As we will see, each system can be used as a component in a larger system and each component of a system is a system by itself. As a result, there is no difference between the notion of a system and that of a system component.

32.2.2.1 Typed Channels

In this section we introduce the concept of a typed channel. A typed channel is a directed communication line over which messages of its specific type are communicated.

A *typed channel* c is a pair $c = (e, T)$ consisting of an identifier e , called the channel identifier, and the type T , called the channel type.

Let CID be a set of identifiers for channels. For a set

$$C \subseteq \text{CID} \times \text{TYPE}$$

of typed channels we denote by $\text{SET}(C) \subseteq \text{CID}$ its set of channel identifiers:

$$\text{SET}(C) = \{e \in \text{CID} : \exists T \in \text{TYPE} : (e, T) \in C\}$$

A set $C \subseteq \text{CID} \times \text{TYPE}$ of typed channels is called a *typed channel set*, if every channel identifier $e \in \text{SET}(C)$ has a unique type in C (in other words, we do not allow in typed channel sets the same channel identifier to occur twice with different types). Formally, we assume for a typed channel set:

$$(c, T_1) \in C \wedge (c, T_2) \in C \Rightarrow T_1 = T_2$$

By $\text{Type}_C(c)$ we denote for $c \in C$ with $c = (e, T)$ the type T .

A typed channel set $C1$ is called a *subtype* of a typed channel set $C2$ if the following formula holds:

$$(c, T1) \in C1 \Rightarrow \exists T2 \in \text{TYPE}: (c, T2) \in C2 \wedge \text{CAR}(T1) \subseteq \text{CAR}(T2)$$

Then $\text{SET}(C1) \subseteq \text{SET}(C2)$ holds. We then write

$C1$ subtype $C2$

Thus, a subtype $C1$ of a set of typed channels carries only a subset of the channel identifiers and each of the remaining channels only a subset of the messages. The idea of subtypes is essential for relating services (see later).

The **subtype**-relation is a partial order, since it is obviously reflexive, transitive, and antisymmetric. In fact, it is a complete partial order on the set of typed channel sets. For two sets of typed channels $C1$ and $C2$ there is always a least set of typed channels C such that both $C1$ **subtype** C and $C2$ **subtype** C hold. C is called the *least super type* of $C1$ and $C2$. Similarly, there exists always a *greatest subtype* for two sets $C1$ and $C2$ of typed channels that is the greatest set of typed channels that is a subtype of both $C1$ and $C2$. Actually, the set of the sets of typed channels forms a lattice. We denote by $\text{glb}\{C1, C2\}$ the greatest subtype and by $\text{lub}\{C1, C2\}$ the least upper bound (the least super type) of the sets $C1$ and $C2$ of typed channels.

32.2.2.2 Syntactic Interfaces

A syntactic interface of a system is defined by the set of messages that can occur as input and output. Since we assume that those messages are communication over channels, we introduce a syntactic interface of systems that consists of typed channels.

Let I be the set of input channels and O the set of output channels of the system F . With every channel in the set $I \cup O$, we associate a data type indicating the type of messages sent along that channel. Then by $(I \blacktriangleright O)$ the *syntactic interface* of a system is denoted. A graphical representation of a system with its syntactic interface and individual channel types is shown in Figure 32.4. It has the syntactic interface

$$(\{x_1 : S_1, \dots, x_n : S_n\} \blacktriangleright \{y_1 : T_1, \dots, y_m : T_m\})$$

By $(I \blacktriangleright O)$ and $\text{SIF}[I \blacktriangleright O]$, we denote the syntactic interface with input channels I and output channels O . By SIF we denote the set of all syntactic interfaces for arbitrary channel sets I and O .

In the following, we give three versions of representations of systems with such syntactic interfaces, namely state machines, stream functions, and architectures.

32.2.3 State View: State Machines

One common way to model a system and its behavior is to describe it by a state machine in terms of a *state space* and its *state transitions*. This leads to a *state view* of systems.

32.2.3.1 State Machine Model: State Transitions

Often systems can be modeled in a well-understandable way by a state transition machine with input and output. A state transition is one step of a system execution leading from a given state to a new state.

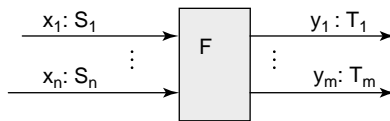


FIGURE 32.4 Graphical representation of a system as a data flow node with input channels x_1, \dots, x_n and output channels y_1, \dots, y_m and their respective types.

Let M be a set of messages. Given a state space Σ , a state machine (Δ, Λ) with input and output according to the syntactic interface $(I \blacktriangleright O)$ is given by a set $\Lambda \subseteq \Sigma$ of initial states as well as a state transition function

$$\Delta : (\Sigma \times (I \rightarrow M^*)) \rightarrow \wp(\Sigma \times (O \rightarrow M^*))$$

For each state $\sigma \in \Sigma$ and each valuation $u: I \rightarrow M^*$ of the input channels in I by sequences we obtain by every pair $(\sigma', s) \in \Delta(\sigma, u)$ a successor state σ' and a valuation $s: O \rightarrow M^*$ of the output channels consisting of the sequences produced by the state transition. If the output depends only on the state we speak of a Moore machine.

By $SM[I \blacktriangleright O]$ we denote the set of all Moore machines with input channels I and output channels O . By SM we denote the set of all Moore machines.

32.2.3.2 Computations of State Machines

In this section we define computations for state machines with input and output.

A computation for a state machine (Δ, Λ) and an input $x \in I \rightarrow (IN_+ \rightarrow M^*)$, with $IN_+ = IN \setminus \{0\}$, is given by a sequence of states

$$\{\sigma_t : t \in IN\}$$

and an output history $y \in O \rightarrow (IN_+ \rightarrow M^*)$ such that for all $t \in IN$ we have

$$(\sigma_{t+1}, y(t+1)) \in \Delta(\sigma_t, x(t+1)) \text{ and } \sigma_0 \in \Lambda$$

The history y is called an *output* of the computation of the state machine (Δ, Λ) for input x and initial state σ_0 . We also say that the machine computes the output y for the input x and initial state σ_0 .

The introduced state machines communicate a finite sequence of messages on each of their channels in each state transition. Thus, a state transition comprises a set of events. We call a state transition therefore a macro-step. We assume that each macro-step is carried within a fixed time interval. It subsumes all events of interaction in this time interval and produces the state that is the result of these events. Thus, the state machines model concurrency in the time intervals.

32.2.4 The Interface Model

In this section, we introduce an interface model for systems. It describes the behavior of a system in the most abstract way by the relation between its streams of input messages and output messages.

32.2.4.1 Streams

Let M be a set of messages, for instance, the carrier set of a given type. By M^* we denote the finite sequences of elements from M . By M^∞ we denote the set of infinite streams of elements of set M , which can be represented by functions $IN_+ \rightarrow M$, where $IN_+ = IN \setminus \{0\}$. By $M^\omega = M^* \cup M^\infty$ we denote the set of streams of elements from the set M , which are finite or infinite sequences of elements from M . A stream represents the sequence of messages sent over a channel during the lifetime of a system.

In fact, in concrete systems communication takes place in a time frame. Actually, we find it often convenient to be able to refer to this time. Therefore, we work with *timed streams*.

Our model of time is extremely simple. We assume that time is represented by an infinite sequence of time intervals of equal length. In each interval on each channel a finite, possibly empty sequence of messages is transmitted. By $(M^*)^\infty$ we denote the set of infinite streams of sequences of elements of set M . Mathematically, a timed stream in $(M^*)^\infty$ can also be understood as a function $IN_+ \rightarrow M^*$.

On each channel that connects two systems there flows a stream of messages in a time frame as shown in Figure 32.5.

Throughout this paper we work with a few simple notations for streams. We use, in particular, the following notations for a timed stream x :

- $z \hat{\ } x$ —concatenation of a sequence or stream z to a stream x .
- $x \cdot k$ — k th sequence in the stream x .

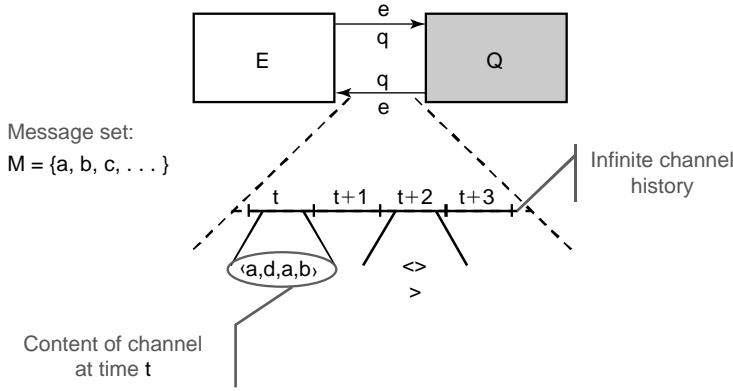


FIGURE 32.5 Streams as models of the interaction between systems.

$x \downarrow k$ —prefix of the first k sequences in the timed stream x .

$M \odot x$ —yields the stream obtained from the stream x by keeping only the messages in the set M (and deleting all messages not in M).

\bar{x} —finite or infinite (untimed) stream that is the result of concatenating all sequences in x .

Let $r \in (M^*)^\infty$; \bar{r} is called the *time abstraction* of the timed stream r .

One of the advantages of timed streams is that we can easily define a merge operator. Merging two streams is a very basic concept. First, we introduce a function to merge sequences

$$\text{merge} : M^* \times M^* \rightarrow \wp(M^*)$$

where (for $s, s1, s2 \in M^*, a1, a2 \in M$):

$$\begin{aligned} \text{merge}(\langle \rangle, s) &= \text{merge}(s, \langle \rangle) = \{s\} \\ \text{merge}(\langle a1 \rangle^{\wedge} s1, \langle a2 \rangle^{\wedge} s2) &= \{ \langle a1 \rangle^{\wedge} s : s \in \text{merge}(s1, \langle a2 \rangle^{\wedge} s2) \} \\ &\quad \cup \{ \langle a2 \rangle^{\wedge} s : s \in \text{merge}(\langle a1 \rangle^{\wedge} s1, s2) \} \end{aligned}$$

This function is easily extended to timed streams $x, y \in (M^*)^\infty$ as follows (let $t \in \mathbb{N}$)

$$\text{merge}(x, y)(t) = \text{merge}(x(t), y(t))$$

We generalize this idea of merging streams to valuations of channels by timed streams, which are introduced in the next section, and speak of the *direct sum* of two histories.

32.2.4.2 Channel Valuations

A typed channel c is given by a channel type T and a channel identifier e , which denotes a stream with stream type $\text{Stream } T$. A channel is basically a name for a stream. Let C be a set of typed channels. A channel valuation is a mapping

$$x : C \rightarrow (M^*)^\infty$$

that associates a timed stream $x(c)$ with each channel $c \in C$, where the timed stream $x(c)$ carries only messages of the type c . The set of valuations of the channels in C is denoted by \vec{C} and by $\text{IH}(C)$.

The operators introduced for streams easily generalize to sets of streams and valuations. Thus, we denote for a channel valuation $x \in \tilde{C}$ by \bar{x} its time abstraction, defined for each channel $c \in C$ by the equation

$$\bar{x}(c) = \overline{x(c)}$$

Note that \bar{x} defines a time abstraction for the channel valuation x .

We generalize the idea of merging streams to valuations of channels by timed streams to the *direct sum* of two histories.

Definition: Direct sum of histories

Given two sets C and C' of typed channels with consistent types (i.e., for joint channel identifiers their types coincide) and histories $z \in IH(C)$ and $z' \in IH(C')$ we define the *direct sum* of the histories z and z' by $(z \oplus z') \subseteq IH(C \cup C')$. It is specified as follows:

$$\begin{aligned} \{y(c) : y \in (z \oplus z')\} &= \{z(c)\} \Leftarrow c \in \text{SET}(C) \setminus \text{SET}(C') \\ \{y(c) : y \in (z \oplus z')\} &= \{z'(c)\} \Leftarrow c \in \text{SET}(C') \setminus \text{SET}(C) \\ (z \oplus z')(c) &= \text{merge}(z \cdot c, z'(c)) \Leftarrow c \in \text{SET}(C) \cap \text{SET}(C') \end{aligned}$$

This definition expresses that each history in the set $z \oplus z'$ carries all the messages the streams z and z' carry in the same time intervals and the same order.

The sum operator \oplus is commutative and associative. The proof of these properties is rather straightforward.

Based on the direct sum we can introduce the notion of a *subhistory ordering*. It expresses that a history contains only a selection of the messages of a given history.

Definition: Subhistory ordering

Given two histories $z \in IH(C)$ and $z' \in IH(C')$ where C **subtype** C' holds, we define the *subhistory ordering* \leq_{sub} as follows:

$$z \leq_{\text{sub}} z' \text{ iff } \exists z'' : z' \in z \oplus z''$$

In fact, the subhistory ordering relation between histories is a partial ordering on the set of channel histories. Again the proof is rather straightforward. The empty stream is the least element in this ordering.

The notions of a timed stream and that of a channel history are essential for defining the behavior of systems and services in the following.

32.2.4.3 Interface Behavior

In the following, let I and O be sets of typed channels.

We describe the *interface behavior* of a system by an *I/O-function* that defines a relation between the input and output streams of a system. An I/O-function is represented by a set-valued function on valuations of the input channels by timed streams.

$$F : \vec{I} \rightarrow \wp(\vec{O})$$

The function yields a set of valuations for the output channels for each valuation of the input channels.

If F is called *strongly causal* it fulfills the following *timing property*, which axiomatizes the time flow (let $x, z \in \vec{I}, y \in \vec{O}, t \in \mathbb{N}$):

$$x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t + 1 : y \in F(x)\} = \{y \downarrow t + 1 : y \in F(z)\}$$

Here $x \downarrow t$ denotes the stream that is the prefix of the stream x and contains t finite sequences. In other words, $x \downarrow t$ denotes the communication histories in the channel valuation x until time interval t . The

timing property expresses that the set of possible output histories for the first $t + 1$ time intervals only depends on the input histories for the first t time intervals. In other words, the processing of messages in a system takes at least one time tick.

Strong causality implies that for every interface behavior F either for all input histories x the sets $F(x)$ are not empty or that for all input histories x the sets $F(x)$ are empty. This is easily proved by choosing $t = 0$ in the formula defining strong causality. In the case where for all input histories x the sets $F(x)$ are empty we speak of a *paradoxical* interface behavior.

If F is called *weakly causal* it fulfills the following *timing property*, which axiomatizes the time flow (let $x, z \in \tilde{I}, y \in \tilde{O}, t \in \mathbb{N}$):

$$x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t : y \in F(x)\} = \{y \downarrow t : y \in F(z)\}$$

Weakly causal functions also show a proper time flow. A reaction to input must not occur before the input arrives. For them, however, there is no time delay required for output that reacts to input. As a consequence, there is no causality required within a time interval for the input and output. This may lead to causal loops. In the following, we rather insist on strong causality (which is the counterpart to Moore machines as introduced above). However, many of our concepts and theorems carry over to the case of weak causality.

By $\text{CIF}[I \blacktriangleright O]$, we denote the set of all I/O-functions with input channels I and output channels O . By CIF , we denote the set of all I/O-functions for arbitrary channel sets I and O . For any $F \in \text{CIF}$, we denote by $\text{In}(F)$ its set of input channels and by $\text{Out}(F)$ its set of output channels.

32.2.5 The Distributed System Model: Composed Systems

An interactive composed system consists of a family of interacting subsystems called *components* (in some approaches also called *agents* or *objects*). These components interact by exchanging messages via their channels, which connect them. A network of communicating components gives a structural system view, also called *system architecture*. Architectures can nicely be represented graphically by directed graphs. Their nodes represent components and their arcs communication lines (channels) on which streams of messages are sent. The nodes represent components, also called *data flow nodes*. The graph represents a net, also called *data flow net*.

32.2.5.1 Uninterpreted Architectures

We model architectures by data flow nets. Let IK be a set of identifiers for components and I and O be sets of input and output channels, respectively. An uninterpreted architecture (composed system, distributed system) (ν, O) with syntactic interface $(I \blacktriangleright O)$ is represented by the mapping

$$\nu : \text{IK} \rightarrow \text{SIF}$$

that associates with every node a syntactic interface. O denotes the output channels of the system.

As a well-formedness condition for forming a net from a set of component identifiers IK , we require that for all component identifiers $k, j \in \text{IK}$ (with $k \neq j$) the sets of output channels of the components $\nu(k)$ and $\nu(j)$ are disjoint. This is formally guaranteed by the condition

$$k \neq j \Rightarrow \text{SET}(\text{Out}(\nu(k))) \cap \text{SET}(\text{Out}(\nu(j))) = \emptyset$$

In other words, each channel has a uniquely specified component as its source.* We denote the set of all (internal and external) channels of the net by the equation

$$\text{Chan}((\nu, O)) = \{c \in \text{In}(\nu(k)) : k \in \text{IK}\} \cup \{c \in \text{Out}(\nu(k)) : k \in \text{IK}\}$$

*Channels that occur as input channels but not as output channels have the environment as their source.

The channel set O determines, which of the channels occur as output. We assume that $O \subseteq \{c \in \text{Out}(v(k)) : k \in \text{IK}\}$.

The set

$$I = \text{Chan}((v, O)) \setminus \{c \in \text{Out}(v(k)) : k \in \text{IK}\}$$

denotes the set of input channels of the net. The channels in the set

$$\text{Intern}((v, O)) = \{c \in \text{Out}(v(k)) : k \in \text{IK}\} \setminus O$$

are called *internal*. By $\text{CUS}[I \blacktriangleright O]$ we denote the set of all uninterpreted architectures with input channels I and output channels O . CUS denotes the set of all uninterpreted architectures.

32.2.5.2 Interpreted Architectures

Given an uninterpreted architecture (v, O) represented by the mapping

$$v : \text{IK} \rightarrow \text{SIF}$$

we get an interpreted architecture by a mapping

$$\eta : \text{IK} \rightarrow \text{CIF} \cup \text{SM}$$

where $\eta(k) \in \text{SM}[I' \blacktriangleright O']$ or $\eta(k) \in \text{CIF}[I' \blacktriangleright O']$ if $v(k) = (I' \blacktriangleright O')$. We also write (η, O) for the interpreted architecture.

By $\text{CIS}[I \blacktriangleright O]$ we denote the set of all interpreted architectures with input channels I and output channels O . CIS denotes the set of all interpreted architectures.

A fully interpreted hierarchical system is defined iteratively as follows: A hierarchical system of level 0 and syntactic interface $(I \blacktriangleright O)$ is denoted by $\text{HS}_0[I \blacktriangleright O]$ and defined by

$$\text{HS}_0[I \blacktriangleright O] = \text{SM}[I \blacktriangleright O] \cup \text{CIF}[I \blacktriangleright O] \cup \text{CIS}[I \blacktriangleright O]$$

By HS_0 we denote the set of all hierarchical system of level 0 with arbitrary syntactic interfaces. A hierarchical system of level $j+1$ and syntactic interface $(I \blacktriangleright O)$ is denoted by $\text{HS}_{j+1}[I \blacktriangleright O]$ and defined by an uninterpreted architecture represented by the mapping

$$v : \text{IK} \rightarrow \text{SIF}$$

a mapping

$$\eta : \text{IK} \rightarrow \text{HS}_j$$

where $\eta(k) \in \text{SM}[I' \blacktriangleright O'] \cup \text{CIF}[I' \blacktriangleright O'] \cup \text{HS}_j[I' \blacktriangleright O']$ if $v(k) = (I' \blacktriangleright O')$.

By $\text{HS}[I \blacktriangleright O]$ we denote the set of all hierarchical systems of arbitrary level with syntactic interface $(I \blacktriangleright O)$. By HS we denote the set of all hierarchical system of arbitrary level with arbitrary syntactic interfaces.

32.3 Structuring Interfaces

Modern software systems offer their users large varieties of different functions, in our terminology called *services* or *features*. We speak of *multifunctional distributed interactive systems*.

32.3.1 Structuring Multifunctional Systems

In this section, we concentrate on concepts supporting the structuring of the functionality of multifunctional systems. Our goal, in essence, is a scientific foundation for the structured presentation of system functions, organized in abstraction layers, forming hierarchies of functions and subfunctions (we speak of “services,” sometimes also of “features”).

Our vision addresses a service-oriented software engineering theory and methodology where services are basic system building blocks. In particular, services address *aspects* of interface behaviors and interface specifications. In fact, services also induce an *aspect-oriented view* onto the functionality of systems and their architectures (see Ref. 22).

Our approach aims at modeling two fundamental, complementary views onto multifunctional systems. These views address the two most significant, in principle, independent dimensions of a structured modeling of systems in the analysis and design phases of software and systems development:

- User functionality and feature hierarchy: Structured modeling of the user functionality of systems as done in requirements engineering and specification:
 - A multifunctional system offers many different functions (“features” or “services” often captured by “use cases”); we are interested in a structured view onto the family of these functions and their logical dependencies.
 - The main result is structured interface specifications of systems.
- Logical hierarchical component architecture: Decomposition of systems into components as done in the design of the architecture:
 - A system is decomposed into a family of components that mutually cooperate to generate the behavior that implements the specified user functionality of the system.
 - The main result is the logical component architecture of the system.

These two complementary tasks of structuring systems and their functionality are crucial in the early phases of software and systems development. The architectural view was introduced at the end of the previous section. The functional hierarchy view is introduced in this section.

32.3.1.1 Services

The notion of a service is the generalization of the notion of an interface behavior of a system. It has a syntactic interface just like a system. Its behavior, however, is “*partial*” in contrast to the totality of a nonparadoxical system interface. Partiality here means that a service is defined (has a nonempty set of output histories) only for a subset of its input histories. This subset is called the *service domain*.

Definition: Service interface

A service interface with the syntactic interface ($I \blacktriangleright O$) is modeled by a function

$$F : \vec{I} \rightarrow \wp(\vec{O})$$

that fulfills the *strong causality property* only for the input histories with a nonempty output set (let $x, z \in \vec{I}$, $y \in \vec{O}$, $t \in \mathbb{N}$):

$$F(x) \neq \emptyset \neq F(z) \wedge x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t + 1 : y \in F(x)\} = \{y \downarrow t + 1 : y \in F(z)\}$$

Such a partial behavior function that fulfills this property is called *strongly causal*, too. The set $\text{dom}(F) = \{x : F(x) \neq \emptyset\}$ is called the *service domain* of F . The set $\text{ran}(F) = \{y \in F(x) : x \in \text{dom}(F)\}$ is called the *service range* of F . By $\text{IF}[I \blacktriangleright O]$, we denote the set of all service interfaces with input channels I and output channels O . By IF , we denote the set of all interfaces for arbitrary channel sets I and O .

Obviously, we have $\text{CIF} \subseteq \text{IF}$ and $\text{CIF}[I \blacktriangleright O] \subseteq \text{IF}[I \blacktriangleright O]$. In contrast to a system, where the causality property implies that for a system F either all output sets $F(x)$ are empty for all input histories x or none, a service is a partial function, in general, with a nontrivial service domain (Figure 32.6).

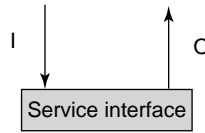


FIGURE 32.6 General representation of a service interface.

To get access to a service, typically, certain access conventions have to be observed. We speak of a *service protocol*. Input histories x that are not in the service domain do not fulfill the service access assumptions. This gives a clear view: a nonparadoxical system is total, while a service may be partial. In other words, a nonparadoxical system is a total service. For a nonparadoxical system there exist nonempty sets of possible behaviors for every input history.

From a methodological point of view, a service is closely related to the idea of a use case as found in object-oriented analysis. It can be seen as a formalization of this idea.

32.3.1.2 Structuring Functionality into Hierarchies of Services

Given a service, we can refine it by extending its domain (provided it is not total). This means that we allow for more input histories with specified output histories and thus enlarge its service domain. Extending service domains is an essential step in service development for instance in making services error tolerant against unexpected input (streams). It is also a step from services to systems if the behaviors are finally made total.

We are, however, not only interested in specifying services and their extensions in isolation, but also interested in being able to specify such extensions in a structured way on top of already specified services. We are, in particular, looking for helpful relationships between the different services, looking for relations that are methodologically useful such as a refinement relation for services. These relations form the arcs of the functional hierarchy.

32.3.1.3 Structuring the Functionality of Multifunctional Systems

In the following, we give a formal definition of the concept of services. This formalization is guided by our basic philosophy concerning services, which is shortly outlined as follows:

- Services are formalized and specified by patterns of interactions.
- Services are partial subbehaviors of (interface) behaviors of systems.
- Systems realize families of services.
- Services address aspects of user functionality

This philosophy of services is taken as a guideline for our way to build up a theory of services. We work out a formal approach to services. We specify services in terms of relations on interactions represented by streams. Consequences of our formal definitions are, in particular, as follows:

- A system is described by a total behavior.
- In contrast, a service is, in general, described by a partial behavior.
- A system is the special case of a total service.
- Services are related to systems—systems offer services.
- A multifunctional system/component is defined in terms of its family of services.

Our theory captures and covers all these notions. Open questions that remain are mainly of methodological and practical nature.

Multifunctional systems can incorporate large families of different, in principle, independent functions, which in our terminology offer different forms of services. So they can be seen as formal models of different use cases of a system.

Our basic methodological idea is that we should be able to reduce the complexity of the user functionality of a system by describing the functionality of systems as follows:

- First we describe each of its single use cases independently by simple services.
- Then we relate these services into a service hierarchy.
- Specify relationships between these individual services that show how the services influence or depend on each other.

Typically, some of the services are completely independent and are just grouped together into a system that offers a collection of functionalities. In contrast, some services may restrict others. There is quite a variety of relations between the individual services of a system. While some services may just have small, often not very essential side effects onto others, other services may heavily rely on other services that influence their behaviors in very essential ways [21].

32.3.1.4 Structuring Systems into a Hierarchy of Services

A system or a combined service may actually implement or offer many independent services. In fact, we can structure the overall functionality of a multifunctional system into the hierarchy of its subservices. We may decompose each system into a family of its subservices and each of these services again and again into families of their subservices.

Understanding the user functionality of a system requires not only the understanding of the single services, but also understanding how they are related and mutually dependent. Our vision here is that we can introduce a number of characteristic relations between the services of a system such that in the end we describe a system structure by just specifying which services are available and how they are related. Each of the individual services is then described in isolation. This can reduce the complexity of the task of specifying a multifunctional system considerably.

Today's information processing systems offer many services as part of their overall functionality. We speak of *multifunctional systems*. We illustrate this idea by an informal description of a simple example of a multifunctional system.

Example: Communication unit

We look at the example of a simple communication network. It has three subinterfaces and offers the following global services:

- User identification, authentication, and access control
- Communication
- User administration

At a finer grained service level we may distinguish subservices such as

- User log-in and identification
- Password change
- Sending message
- Receiving message
- Calibration of the quality of services
- User introduction
- User deletion
- Change of user rights

All these extended services can be described in isolation by specification techniques as the one introduced and demonstrated above. However, there are a number of dependencies between these services.

To obtain a comprehensive view onto the hierarchy of services, we introduce a notion of user roles as shown in our example:

- SAP A and B (SAP = Service Access Point)
- Administrator

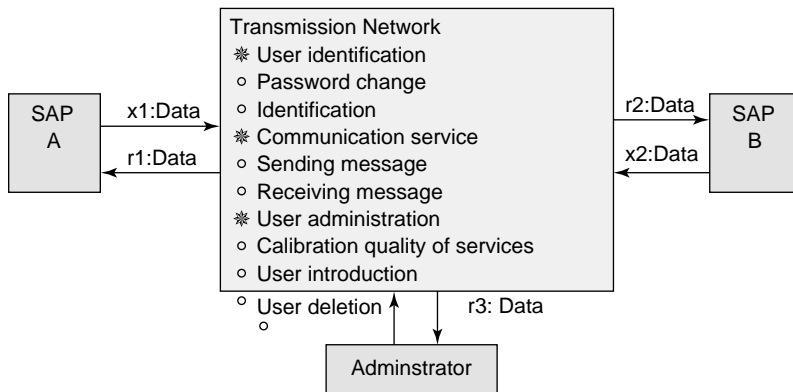


FIGURE 32.7 Service structure and interface.

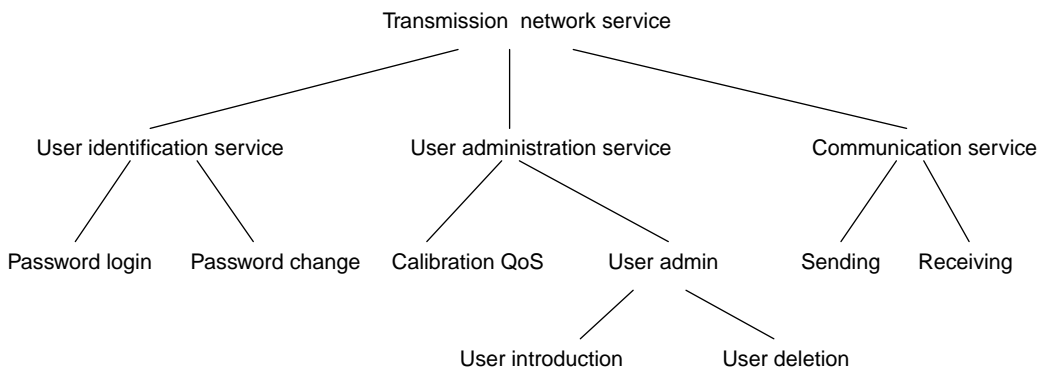


FIGURE 32.8 Service hierarchy—the lines represent the subservice relation.

The set of services offered by a system can be informally described as shown in Figure 32.7. In fact, it can be structured into a hierarchy of services and subservices as shown in Figure 32.8. By such a service hierarchy we get a structuring of a system into a hierarchy of services that are in the subservice relation.

In such a service hierarchy we can relate services by introducing appropriate more specific relations between services, converting the service hierarchy into a directed graph.

32.3.2 Relating Services

In this section we deal with a number of characteristic relations between services. The most significant one for requirements engineering is that of a subservice that will be formally introduced later. There are, of course, many more practically interesting relations between two services A and B besides the subservice relation as introduced in the previous section. Informal examples of such relations for services A and B are

- A and B are mutually independent
- A affects B
- A controls B
- A includes B

Just characterizing the relations by well-chosen names is helpful, but in the end not sufficient, since it does, of course, not fix their precise meaning. It is not difficult to dream up a number of further

methodologically useful relations. Typical further characteristic relations between services are listed in the following:

- Uses
- Enables
- Changes
- Interferes (feature interaction)
- Is combined of
- Is sequentially composed of

Of course, just giving names for these relations is not sufficient. Since we have a formal notion of service, we are actually able to give formal definitions for such concepts of relations between services.

32.4 Refinement

In requirements engineering, in the design and implementation phase of system development many issues have to be addressed such as requirements elicitation, conflict identification and resolution, information management, as well as the selection of a favorable software architecture. These activities are connected with development steps. Refinement relations are the medium to formalize development steps and in this way the development process.

We formalize the following basic ideas of refinement of interfaces:

- *Property refinement*—enhancing requirements—allows us to add properties to a specification.
- *Glass box refinement, implementation refinement*—designing implementations—allows us to decompose a system into a composed system or to give a state transition description for a system specification.
- *Interaction refinement*—relating *levels of abstraction*—allows us to change the representation of the communication histories, in particular, the granularity of the interaction as well as the number and types of the channels of a system.

In fact, these notions of refinement describe the steps needed in an idealistic view of a strictly hierarchical top-down system development. The two refinement concepts mentioned above are formally defined and explained in detail in the following.

32.4.1 Property Refinement

Property refinement is a classical concept in program development. It is based—just as deduction in logics—on logical implication. In our model this relates to set inclusion.

Property refinement allows us to replace an interface behavior or an interface with one having additional properties. This way interface behaviors are replaced by more restricted ones. An interface

$$F : \vec{I} \rightarrow \wp(\vec{O})$$

is refined by a behavior

$$\hat{F} : \vec{I} \rightarrow \wp(\vec{O})$$

if

$$F \approx_{\text{IF}} \hat{F}$$

This relation stands for the proposition

$$\forall x \in \vec{I} : \hat{F}(x) \subseteq F(x)$$

Obviously, property refinement is a partial order and therefore reflexive, asymmetric, and transitive. Note that the paradoxical system logically is a refinement for every system with the same syntactic interface.

A property refinement is a basic refinement step adding requirements as it is done step by step in requirements engineering. In the process of requirements engineering, typically, the overall services of a system are specified. Requiring more and more sophisticated properties for systems until a desired behavior is specified, in general, does this.

32.4.2 Compositionality of Property Refinement

In our case, the proof of the compositionality of property refinement is straightforward. This is a consequence of the simple definition of composition. Let (ν, O) be an uninterpreted architecture with interpretations η and η' with the set IK of components given by their interfaces. The rule of compositional property refinement reads as follows:

$$\frac{\forall k \in IK : \eta(k) \approx > \eta'(k)}{F_{(\eta, O)} \approx > F_{(\eta', O)}}$$

Compositionality is often called *modularity* in system development. Modularity allows for a separate development of components.

Modularity guarantees that separate refinements of the components of a system lead to a refinement of the composed system.

The property refinement of the components of composed systems leads to a property refinement for the composed system independently of the question whether the components are described by interfaces, state machines, or architectures.

32.4.3 Granularity Refinement: Changing Levels of Abstraction

In this section we show how to change the levels of abstractions by refinements of the interfaces, state machines, and processes. Changing the granularity of interaction and thus the level of abstraction is a classical technique in software system development. Interaction refinement is the refinement notion for modeling development steps between levels of abstraction. Interaction refinement allows us to change for a component

- The number and names of its input and output channels.
- The types of the messages on its channels determining the granularity of the communication.

An *interaction refinement* is described by a pair of two functions

$$A: \vec{C}' \rightarrow \wp(\vec{C}) \quad R: \vec{C} \rightarrow \wp(\vec{C}')$$

that relate the interaction on an abstract level with corresponding interaction on the more concrete level. This pair specifies a development step that is leading from one level of abstraction to the other one as illustrated by Figure 32.9. Given an abstract history $x \in \vec{C}$ each $y \in R(x)$ denotes a concrete history representing x . Calculating a representation for a given abstract history and then its abstraction yields the old abstract history again. Using sequential composition, this is expressed by the requirement:

$$R \circ A = \text{Id}$$

Let Id denote the identity relation and “ \circ ” the sequential composition defined as follows:

$$(R \circ A)(x) = \{y \in A(z) : z \in R(x)\}$$

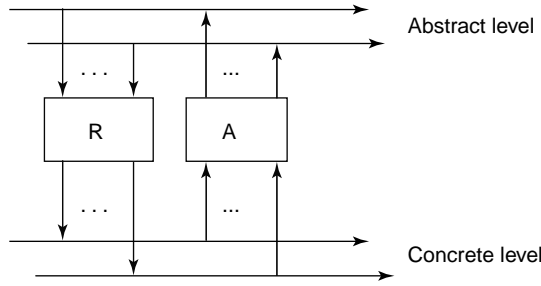
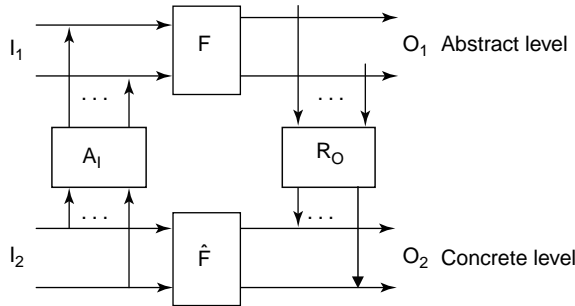


FIGURE 32.9 Communication history refinement.

FIGURE 32.10 Interaction refinement (U^{-1} -simulation).

A is called the *abstraction* and R the *representation*. R and A are called a *refinement pair*. For “untimed” systems we weaken this requirement by requiring $R \circ A$ to be a property refinement of the untimed identity, formally expressed by the following equation:

$$\overline{(R \circ A)(x)} = \{\bar{x}\}$$

This defines an identity under time abstraction.

Interaction refinement allows us to refine systems, given appropriate refinement pairs for their input and output channels. The idea of an interaction refinement is visualized in Figure 32.10 for the so-called U^{-1} -simulation. Note that here the components (boxes) A_I and A_O are no longer definitional in the sense of specifications, but rather methodological, since they relate two levels of abstraction.

Given refinement pairs

$$\begin{array}{ll} A_I: \vec{I}_2 \rightarrow \wp(\vec{I}_1) & R_I: \vec{I}_1 \rightarrow \wp(\vec{I}_2) \\ A_O: \vec{O}_2 \rightarrow \wp(\vec{O}_1) & R_O: \vec{O}_1 \rightarrow \wp(\vec{O}_2) \end{array}$$

for the input and output channels we are able to relate abstract to concrete channels for the input and for the output. We call the interface

$$\hat{F}: \vec{I}_2 \rightarrow \wp(\vec{O}_2)$$

an *interaction refinement* of the I/O-behavior

$$F: \vec{I}_1 \rightarrow \wp(\vec{O}_1)$$

if the following proposition holds: $\approx>$

$$A_I \circ F \circ R_O \approx> \hat{F} \quad U^{-1}\text{-simulation}$$

This formula essentially expresses that \hat{F} is a property refinement of the system $A_I \circ F \circ R_O$. Thus, for every “concrete” input history $\hat{x} \in \hat{I}_2$ every concrete output $\hat{y} \in \hat{O}_2$ can be also obtained by translating \hat{x} onto an abstract input history $x \in A_I(\hat{x})$ such that we can choose an abstract output history $y \in F(x)$ such that $\hat{y} \in R_O(y)$.

The idea of granularity refinement is of particular interest from a methodological point of view. It formalizes the idea of levels of abstraction as found in many approaches to software and systems engineering. This includes the classical ISO/OSI protocol models (see Ref. 26) as well as the idea of layered architectures (see Ref. 15).

32.5 Composition and Combination

In this section we study forms of composition for systems. To cope with large systems composition should always be hierarchical. This means that we can compose systems and the composition yields systems that are units of composition again. As a result we get trees or hierarchies of subsystems. We study two operations on systems and system components: composition and combination. In principle, we have introduced both operations already, when studying

- *Functional enhancement by service combination*: the notion of a subservice yields a notion of a decombination of the functionality of a larger multifunctional system into a set of subservices; this leads again to an operation to compose—we rather say to combine systems to larger systems.
- *System composition*: the relationship between architectures and interfaces as well as state machines; this yields a form of composition of systems.

The two operations on systems are fundamental in system development. Functional enhancement addresses requirements engineering while system composition addresses architecture design.

32.5.1 Functional Enhancement: Combining Services

In principle, there are several ways to combine services out of given services. First of all we can try to *combine* more elaborate services from given ones. In the simple case we just put together services, which are more or less independent within a family of services of a multifunctional system. In this section, we are interested in a combination of services that are not necessarily independent.

Definition: Service combination

The combination of the two services $F1 \in [I1 \blacktriangleright O1]$ and $F2 \in [I2 \blacktriangleright O2]$ is only defined, if they are combinable; then we denote them, w.r.t. the sub-type-relation by

$$F1 \oplus F2 \in [I \blacktriangleright O]$$

where I is the lub of $\{I1, I2\}$ and O is the lub of $\{O1, O2\}$. We define $F1 \oplus F2 = F$ the service F by the property

$$F(x) = \{y : y|O1 \in F1 \cdot (x|I1) \wedge y|O2 \in F2 \cdot (x|I2)\}$$

such that (here for $x \in \bar{C}$ by $x|C'$ with $C' \subseteq C$ we denote the restriction of x to C')

$$F1 \subseteq_{\text{sub}} F \wedge F2 \subseteq_{\text{sub}} F$$

$F1 \oplus F2$ is called the *service combination* of $F1$ and $F2$.

By service combination we can build up multifunctional systems from elementary services.

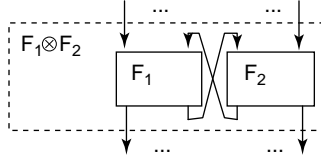


FIGURE 32.11 Parallel composition with feedback.

A service provides a partial view onto the interface behavior of a system. The characterization of the service domain can be specified and used in service specifications by formulating assumptions characterizing the input histories in the service domain.

In our interface model a system is a set-valued function on streams. As a consequence all operations on sets are available. The interface model forms a complete lattice. This way we can form the union and the disjunction of interfaces. In other words, we may join or intersect specifications.

32.5.2 Architecture Design: Composition of Interfaces

Architectures consist of sets of components that are connected via their channels. If these components work together in interpreted architectures according to their interface specifications architectures generate behaviors. In fact, this shows that in architecture design we have two concepts of composition: the “syntactic” composition of syntactic interfaces into uninterpreted architectures forming data flow nets and the “semantic” composition of semantic interfaces in the sense of interpreted architectures into interface behaviors of the interpreted architecture.

32.5.2.1 Composing Interfaces

Given I/O-behaviors with disjoint sets of output channels ($O_1 \cap O_2 = \emptyset$)

$$F_1 : \vec{I}_1 \rightarrow \wp(\vec{O}_1), \quad F_2 : \vec{I}_2 \rightarrow \wp(\vec{O}_2)$$

we define the parallel composition with feedback as it is illustrated in Figure 32.11 by the I/O-behavior

$$F_1 \otimes F_2 : \vec{I} \rightarrow \wp(\vec{O})$$

with a syntactic interface as specified by the equations:

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2), \quad O = (O_1 \cup O_2)$$

The resulting function is specified by the following equation (here we assume $y \in \vec{C}$ where the set of all channels C is given by $C = I_1 \cup I_2 \cup O_1 \cup O_2$):

$$(F_1 \otimes F_2)(x) = \{y|O : y|I = x \wedge y|O_1 \in F_1(y|I_1) \wedge y|O_2 \in F_2(y|I_2)\}$$

Here, for a channel set $C' \subseteq C$ we denote for $y \in \vec{C}$ by $y|C'$ the restriction of y to the channels in C' .

As long as F_1 and F_2 have disjoint sets of input and output channels the composition is simple. Given $x_1 \in \vec{I}_1$ and $x_2 \in \vec{I}_2$, we get

$$(F_1 \otimes F_2) \cdot (x_1 \oplus x_2) = \{y_1 \oplus y_2 : y_1 \in F_1(x_1) \wedge y_2 \in F_2(x_2)\}$$

Now assume

$$I_1 = O_1 \text{ and } I_2 = O_2 = \emptyset$$

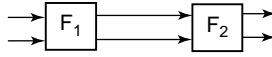


FIGURE 32.12 Pipelining.

We write $\mu.F_1$ for $F_1 \otimes F_2$ since F_2 is then the system without input and output. We get $I = \emptyset$ ($\mu.F_1$ has no input channels) and

$$\mu \cdot F_1 = \{y : y \in F_1(y)\}$$

This somewhat special construction shows that composition with feedback loops corresponds to a kind of fixpoint equation. We call $y \in F_1(y)$ a fixpoint of F_1 . Note that in case of a deterministic function $f_1: \vec{O}_1 \rightarrow \vec{O}_1$ we get $y = f(y)$.

The operator is a rather general composition operator that can be easily extended from two systems to a family of systems.

A more specific operation is sequential composition also called *pipelining*. It is a special case of the composition operator where $O_1 = I_2$ and the sets I_1 and O_2 are disjoint. In this case we define

$$F_1 \circ F_2 = F_1 \otimes F_2$$

where the composition is illustrated by Figure 32.12.

Pipelining is the special case of composition without feedback. It can easily be generalized to the case where the channel sets I_1 and O_2 are not disjoint. The generalized definition reads as follows:

$$(F_1 \circ F_2)(x) = \{z \in F_2(y) : y \in F_1(x)\}$$

This composition is also called *relational composition* if F_1 and F_2 are represented as relational or *functional composition* if F_1 and F_2 are deterministic and thus functions.

32.5.2.2 Composition and Architecture

Each interpreted architecture (η, O) describes an interface behavior $F_{(\eta, O)} \in \text{IF}[I \blacktriangleright O]$. This behavior is basically the composition of all interface behaviors F_1 of the systems $F_1, F_2, F_3 \dots$ of the architecture $F_1 \otimes F_2 \otimes F_3 \dots$. This shows that system composition is the basis for deriving interface behaviors of architectures from the interface behaviors of the components of architectures.

32.6 Modeling Time

In this section we show how to model and work with different time granularities and timescales. We show, in particular, how we can work with different timescales for the different components within architectures. This is of interest when running several functions in multiplexing mode on one hardware node (CPU) with different sample time requirements.

32.6.1 Changing the Timescale

We first show how to make the timescale coarser.

32.6.1.1 Coarsening the Timescale

Let $n \in \mathbb{N}_+$ and C be a set of typed channels; to make for a channel history (or a stream)

$$x \in \vec{C}$$

its timescale coarser by the factor n we introduce the coarsening function

$$\text{COA}(n) : \vec{C} \rightarrow \wp(\vec{C})$$

defined by (for all $t \in \mathbb{N}$):

$$\text{COA}(n)(x)(t+1) = x(t * n + 1) \wedge \dots \wedge x(t * n + n)$$

$\text{COA}(n)(x)$ yields a history from history x where for each stream associated with a channel n successive time intervals are concatenated (“abstracted”) into one. We forget about some of the time distribution. The timescale is made coarser that way.

Time coarsening obviously represents a form of abstraction. We forget some information about the timing this way. Distinct histories may be mapped onto the same history by timescale coarsening.

It is not difficult to allow even a coarsening factor $n = \infty$ in time coarsening. Then an infinite number of time intervals is mapped into one. Timed streams are abstracted into nontimed streams:

$$\text{COA}(\infty)(x) = \bar{x}$$

On histories, coarsening is a function that is not injective and thus there does not exist an inverse.

We generalize the coarsening of the timescale from channel histories to behaviors. To make a behavior

$$F : \vec{I} \rightarrow \wp(\vec{O})$$

coarser by the factor n , we define the coarsening operator that maps F onto

$$\text{COA}(F, n) : \vec{I} \rightarrow \wp(\vec{O})$$

which is defined as follows:

$$\text{COA}(F, n)(x) = \{\text{COA}(n)(y) : \exists x' : x = \text{COA}(n)(x') \wedge y \in F(x')\}$$

Coarsening maps behavior functions onto behavior functions. On one hand, coarsening may introduce some kind of further nondeterminism and underspecification into behaviors due to the coarser timescale of the input histories. Certain different input histories are mapped by the time coarsening onto the same coarser input histories. Then their sets of output histories are defined by the union of all their coarsened output histories. This way the nondeterminism may grow.

On the other hand, some nondeterminism and underspecification may be removed in behaviors by coarsening since different output histories may be mapped by the time coarsening on the same coarsened output history.

A special case is the coarsening $\text{COA}(\infty)$, which abstracts completely away all time information. If the output of F is depending on the timing of the input, then the coarsening $\text{COA}(F, \infty)$ introduces a lot of nondeterminism, in general. However, if the output produced by F does not depend on the timing of the input messages at all but only on their values and the order in which they arrive, $\text{COA}(F, \infty)$ will rather be more deterministic.

If F is weakly causal, the behavior $\text{COA}(F, n)$ is obviously weakly causal, too. However, strong causality is not maintained, in general. We will come back to more explicit rules of causality and coarsening later. Reactions to input at later time intervals may be mapped into one time interval.

32.6.1.2 Making the Timescale Finer

We can also map a history as well as a behavior onto finer time granularities. Let $n \in \mathbb{N}$; to make for a history (or a stream)

$$x \in \vec{C}$$

its timescale finer by the factor n we use the function

$$\text{FINE}(n) : \vec{C} \rightarrow \wp(\vec{C})$$

defined by the equation

$$\text{FINE}(n)(x)(t + 1) = \{x' \in \vec{C} : x(t) = x' \cdot (n * t + 1) \hat{\cdot} \dots \hat{\cdot} x' \cdot (n * t + n)\}$$

$\text{FINE}(n)(x)$ yields a set of histories where for each time interval the sequences of messages in this interval are arbitrarily subdivided into n sequences that are associated with n successive time intervals. Thus, the sequence on each time interval for each channel is nondeterministically divided into n sequences. The timescale is made finer that way.

Making the timescale finer is a form of concretization. Each history is mapped onto a number of histories by making its timescale finer. Each of these histories represents one version of the history with the finer time granularity.

Along this line of discussion another way to define the function FINE is given by the following formula

$$\text{FINE}(n)(x) = \{x' : \text{COA}(n)(x) = x'\}$$

This equation shows more explicitly the relationship between making the timescale coarser and making the timescale finer. They are “inverse” operations. Changing the timescale represents an abstraction, if we make the timescale coarser, and a concretization, if we make it finer.

The idea of making a timescale finer can also be applied to behaviors. We specify

$$\text{FINE}(F, n)(x) = \{\text{FINE}(n)(y) : \exists x' : x = \text{FINE}(n)(x') \wedge y \in F(x')\}$$

Owing to the nondeterminism in the way we make the timescale finer, there is no guarantee that we get a higher number of delays in the behaviors when moving to a finer timescale.

32.6.2 Rules for Timescale Refinement

Changing the timescale is an operation on histories and behaviors. In this section we study laws and rules for changing the timescale.

Our first rules of changing the timescale show that the functions $\text{COA}(n)$ and $\text{FINE}(n)$ form refinement pairs in the sense of granularity refinement:

$$\text{COA}(n)(\text{FINE}(n)(x)) = x$$

$$x \in \text{FINE}(n)(\text{COA}(n)(x))$$

In other words, coarsening is the inverse of making the timescale finer. We observe, in particular, the following equations:

$$\text{COA}(F, n) = \text{FINE}(n) \circ F \circ \text{COA}(n)$$

$$\text{FIN}(F, n) = \text{COA}(n) \circ F \circ \text{FINE}(n)$$

The proof is quite straightforward. The equations show that time refinement in fact is a special case of interaction granularity refinement (see Ref. 16).

Both abstractions and refinements by factors $n * m$ can be seen as two consecutive refinements by the factor n followed by m or *vice versa*.

We get the following obvious rules:

$$\text{FINE}(n * m) = \text{FINE}(n) \circ \text{FINE}(m)$$

$$\text{COA}(n * m) = \text{COA}(n) \circ \text{COA}(m)$$

32.7 Perspective, Related Work, Summary, and Outlook

In this final section, we put our modeling theory into perspective with related issues of software and systems engineering, refer to related work, and give a summary and an outlook.

32.7.1 Perspective

In this paper we concentrate on the modeling software-intensive systems and a theory of modeling. Of course, in engineering software-intensive systems we cannot work directly with the theory. We need well-chosen notations, using textual or graphical syntax or tables. In practical development it helps including notational conventions to make the presentations of the models better tractable. Furthermore, we need logical calculi to prove properties about relationships between model descriptions. And finally we need tools that support our modeling theory in terms of documentation, analysis, verification, and transformation (see Ref. 1).

Our theory is carefully designed to make such a support possible and some of what is needed is already available. There exists a well worked out proof theory (see Ref. 14) as well as tool support (see Refs. 2, 3, and 46) that includes a graphical notation for the model views. There is plenty of research to apply and detail the presented theory (see Refs. 9, 12, 13, 15, 17, 26, and 41).

32.7.2 Related Work

Modeling has been, is, and will be a key activity in software and systems engineering. Research in this area therefore has always been of major interest for informatics.

Early pioneering work in system and software modeling led to Petri nets (see Refs. 39 and 40) aiming at highly abstract models for distributed concurrent systems. Early work based on Petri nets led to data flow models (see Ref. 32). Another early line of research is denotational semantics (see Ref. 52) aiming at mathematical models of programs. Also the work on programming logics (see, for instance, Ref. 25) always had a sometimes rather implicit flavor of modeling issues. In fact, much of the early work on data structures was a search for the appropriate models (see Ref. 11).

Very much influenced by work on programming language semantics—and also by denotational semantics—is the work on VDM (see Ref. 30). Other work on formal specification such as Z (see Ref. 51) and B (see Ref. 1) has also a modeling flavor, although in this work often mathematical concepts are more explicitly emphasized than system modeling issues.

Very often in the history of software and systems engineering notations were suggested first—of course with some ideas in mind what to express by them—and only later it was found out that giving a consistent meaning to the notations was much less obvious and much harder than originally thought. Examples are CCS (see Ref. 38), CSP (see Ref. 28), where it took several years of research to come up with a reasonable semantical model, and statecharts. Recent examples along these lines are approaches like UML (see Ref. 18). This way plenty of interesting work on modeling was and still is triggered.

Much work in modeling and modeling theory was carried out by research on formal techniques for distributed systems. Typical examples are CSP (see Ref. 28), CCS (see Ref. 38), or more general process algebras (see Ref. 4), and later Unity (see Ref. 19) as well as TLA (see Ref. 34). In Unity, the underlying model is kept very implicit and everything is explicitly done by some abstract programming notation and a logical calculus. Nevertheless, there is a state machine model behind Unity. In TLA the underlying model is more explicitly explained. All the mentioned approaches do not define system views but rather one system model. Sometimes the system model is enhanced by a specification framework, which can be understood as a complementary view. Also general work on temporal logic (see Ref. 37) always included some modeling research. More recent developments are evolving algebras (see Ref. 23).

Other examples are the so-called synchronous languages such as Lustre and Signal (see Ref. 36) or Esterel (see Refs. 5 and 6). They all aim at programming notations for real-time computations and not so much at making their underlying models explicit.

A pioneer in modeling with a more practical flavor is Michael Jackson (see Ref. 31). He related modeling issues with engineering methods (see also Ref. 53).

Another line of research related to modeling is the field of architecture description languages (see Refs. 33 and 35). Also, there we find the discrepancy between a suggestive notation, an extensive terminology talking about components and connectors, for instance, and a poor, often rather implicit modeling theory.

A completely different line of research aims at visual modeling notations. Statecharts (see Ref. 24) were suggested as a graphical modeling language. It took a while until the difficulties have been recognized to give meaning to statecharts.

A quite different approach arose from more pragmatic work on system modeling. Graphical notations were suggested first control flow graphs, later in SA (see Ref. 20) and SADT (see Refs. 42 and 43). Early modeling languages were SDL (see Refs. 47, 48, 49, and 10). Then the object-oriented approaches (see Refs. 29 and 45) came such as OOD (see Ref. 7), OADT (see Ref. 44), ROOM (see Ref. 50) and many others. Today UML (see Ref. 8) is much advocated, although in many aspects it is just a graphical notation with many open problems concerning its modeling theory.

None of the approaches tried explicitly to bring together the different views and to formally relate them and to introduce, in addition, refinement as an additional relation between the views.

32.7.3 Summary and Outlook

Why did we present this setting of mathematical models and relations between them? First of all, we want to show how rich and flexible the tool kit of mathematical model is and has to be, and how far we are in integrating and relating them. Perhaps, it should be emphasized that we presented first of all an integrated system model very close to practical approaches by SDL or UML where a system is a set or hierarchy of components. In this tree of components the leaves are state machines. In our case, the usage of streams and stream processing function is the reason for the remarkable flexibility of our model toolkit and the simplicity of the integration.

There are many interesting directions of further research on the basis of the presented theory of modeling. One direction is to apply to it the specific domains. This may lead to domain-specific modeling languages.

Another interesting issue is the relationship to programming languages and standard software infrastructure such as operating systems. In such an approach it has to be worked out how the introduced models are mapped on programs that are executed in an infrastructure of operating systems and middleware. For an example in that direction, see Ref. 9.

Acknowledgments

It is a pleasure to thank Markus Pizka, Leonid Kof, Ingolf Krüger, and Bernhard Schätz for stimulating discussions and helpful remarks on draft versions of the manuscript. I thank Judith Hartmann for carefully proof reading the manuscript.

References

1. J.R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
2. Website of AutoFocus with documentation, screenshots, tutorials and download.
<http://autofocus.in.tum.de>.
3. Website AutoRAID, with documentation, screenshots and downloads.
<http://wwwbroy.in.tum.de/~autoraid/>.
4. J.C.M. Baeten, J. Bergstra. *Process Algebras with Signals and Conditions*. In: M. Broy (ed.), *Programming and Mathematical Method*. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 88, 1992, pp. 273–324.

5. G. Berry, G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. INRIA, Research Report 842, 1988.
6. G. Berry. The Foundations of Esterel. MIT Press, Cambridge, MA, USA, 2000.
7. G. Booch. Object Oriented Design with Applications. Benjamin Cummings, Redwood City, CA, 1991.
8. G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language for Object-Oriented Development, Version 1.0, RATIONAL Software Cooperation.
9. J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul, M. Spichkova. On the Correctness of Upper Layers of Automotive Systems. To appear.
10. M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing* 3: 21–57, 1991.
11. M. Broy, C. Facchi, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I/II Technische Universität München, Institut für Informatik, TUM-I9311 / TUM-I9312, May 1993.
12. M. Broy. Refinement of Time. In: M. Bertran, Th. Rus (eds), *Transformation-Based Reactive System Development. ARTS'97, Mallorca 1997. Lecture Notes in Computer Science* 1231, 1997, pp. 44–63. Springer, Berlin.
13. M. Broy, C. Hofmann, I. Krüger, M. Schmidt. A Graphical Description Technique for Communication in Software Architectures. Technische Universität München, Institut für Informatik, TUM-I9705, February 1997 URL: <http://www4.informatik.tu-muenchen.de/reports/TUM-I9705>, 1997. Also in: Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97).
14. M. Broy, K. Stølen. Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement. Springer, New York, 2001.
15. M. Broy. Modeling Services and Layered Architectures. In: H. König, M. Heiner, A. Wolisz (eds.), *Formal Techniques for Networked and Distributed Systems. Berlin 2003, Lecture Notes in Computer Science* 2767, Springer, Berlin, 2003, 48–61.
16. M. Broy. Time, Abstraction, Causality, and Modularity in Interactive Systems. FESCA 2004. Workshop at ETAPS 2004, 1–8.
17. M. Broy. The Semantic and Methodological Essence of Message Sequence Charts. *Science of Computer Programming*, SCP 54(2–3): 213–256, 2004.
18. M. Broy, M.V. Cengarle, B. Rumpe. Semantics of UML. Towards a System Model for UML. The Structural Data Model, Technische Universität München, Institut für Informatik, Report TUM-IO612, June 06.
19. K.M. Chandy, J. Misra. *Program Design. A Foundation*. Addison-Wesley, Boston, MA, USA, 1988.
20. T., DeMarco. *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1979.
21. M. Deubler. *Dienst-orientierte Softwaresysteme: Anforderungen und Entwurf*. Dissertation, To appear.
22. R. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, Boston, MA, USA, 2004.
23. Y. Gurevich: Evolving algebras. In: B. Pehrson, I. Simon (eds), *IFIP 1994 World Computer Congress, volume I: Technology and Foundations*, pp. 423–427. Elsevier Science Publishers, Amsterdam, 1994.
24. D. Harel: Statecharts. A Visual Formalism for Complex Systems. *Science of Computer Programming* 8: 231–274, 1987.
25. E.C.R. Hehner. *A Practical Theory of Programming*. Springer, Berlin, 1993.
26. D. Herzberg, M. Broy. Modeling layered distributed communication systems. *Applicable Formal Methods*. Springer, Berlin, Vol. 17, No. 1, May 2005.
27. R. Hettler. Zur Übersetzung von E/R-Schemata nach SPECTRUM. Technischer Bericht TUM-I9409, TU München, 1994.

28. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.
29. I. Jacobsen. *Object-Oriented Software Engineering*. Addison-Wesley, Boston, MA, USA, ACM Press, 1992.
30. C. Jones. *Systematic Program Development Using VDM*. Prentice-Hall, 1986.
31. M. A. Jackson. *System Development*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1983.
32. G. Kahn. The Semantics of a Simple Language for Parallel Processing. In: J.L. Rosenfeld (ed.): *Information Processing 74*. Proc. of the IFIP Congress 74, Amsterdam, North Holland, 1974, 471–475.
33. D. Garlan, R. Allen, J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, pp. 17–26, November 1995.
34. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16(3): 872–923, 1994.
35. D.C. Luckham, J.L. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering* 21(4): 336–355, 1995.
36. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. LeGuernic, R. De Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 2003.
37. Z. Manna, A. Pnueli. *A Temporal Logic of Reactive Systems and Concurrent Systems*. Springer, 1992.
38. R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer, Berlin, 1980.
39. C.A. Petri. *Kommunikation mit Automaten*. Technical Report RADCTR-65–377, Bonn, Institut für Instrumentelle Mathematik, 1962.
40. C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. of IFIP Congress 62*. North Holland Publ. Comp., Amsterdam, pp. 386–390, 1963.
41. J. Romberg. *Synthesis of distributed systems from synchronous dataflow programs*. Ph.D. Thesis, Technische Universität München, Fakultät für Informatik, 2006.
42. D.T. Ross. Structured analysis (SA): A Language for Communicating Ideas. *IEEE Transactions on Software Engineering* 3(1): 16–34, 1977.
43. D.T. Ross. Applications and Extensions of SADT. In: E. P. Glinert (ed.), *Visual Programming Environments: Paradigms and Systems*, pp. 147–156. IEEE Computer Society Press, Los Alamitos, 1990.
44. J. Rumbaugh. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
45. B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Ph.D. Thesis, Technische Universität München, Fakultät für Informatik 1996. Published by Herbert Utz Verlag.
46. B. Schätz. Mastering the Complexity of Embedded Systems—The AutoFocus Approach. In: F. Fabrice Kordon, M. Lemoine (eds): *Formal Techniques for Embedded Distributed Systems: From Requirements to Detailed Design*. Kluwer, 2004.
47. Specification and Description Language (SDL), Recommendation Z.100. Technical Report, CCITT, 1988.
48. ITU-T (previously CCITT). Criteria for the Use and Applicability of Formal Description Techniques. Recommendation Z. 120, Message Sequence Chart (MSC), March 1993, 35p.
49. ITU-T. Recommendation Z.120, Annex B. Algebraic Semantics of Message Sequence Charts. ITU-Telecommunication Standardization Sector, Geneva, Switzerland, 1995.
50. B. Selic, G. Gullekson, P.T. Ward. *Real-time Objectoriented Modeling*. Wiley, New York, 1994.
51. M. Spivey. *Understanding Z—A Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3, Cambridge University Press, Cambridge, 1988.
52. J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Languages*. MIT Press, Cambridge, MA, USA, 1977.
53. P. Zave, M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, January 1997.

33

Metamodeling Languages and Metaprogrammable Tools

Matthew Emerson
Vanderbilt University

Sandeep Neema
Vanderbilt University

Janos Sztipanovits
Vanderbilt University

33.1	Introduction	33-1
33.2	Modeling Tool Architectures and Metaprogrammability	33-3
	Metaprogrammable Modeling Tool—GME • Metaprogrammable Model Manipulation Management API—UDM • Metaprogrammable Design Space Exploration—DESERT • Metaprogrammable Model Transformations—GReAT	
33.3	A Comparison of Metamodeling Languages	33-9
	MetaGME • Ecore • Microsoft Domain Model Designer • Language Differentiators	
33.4	Relating Metamodeling Languages and Metaprogrammable Tools	33-14
	Example: UDM • Example: GME-MOF	
33.5	Conclusion	33-16

33.1 Introduction

The convergence of control systems and software engineering is one of the most profound trends in technology today. Control engineers build on computing and communication technology to design robust, adaptive, and distributed control systems for operating plants with partially known nonlinear dynamics. Systems engineers face the problem of designing and integrating large-scale systems where networked embedded computing is increasingly taking over the role of “universal system integrator.” Software engineers need to design and build software that needs to satisfy requirements that are simultaneously physical and computational. This trend drives the widespread adoption of model-based design techniques for computer-based systems. The use of models on different levels of abstraction have been a fundamental approach in control and systems engineering. Lately, model-based software engineering methods, such as objective management group (OMG)’s model-driven architecture (MDA) [1,2] have gained significant momentum in the software industry, as well. The confluence of these factors has led to the emergence of model-driven engineering (MDE) that opens up the opportunity for the fusion of traditionally isolated disciplines.

Model-integrated computing (MIC), one practical manifestation of the general MDE vision, is a powerful model-based design approach and tool suite centered on the specification and use of semantically

rich, domain-specific modeling languages (DSMLs) during the full system life cycle [3]. Domain architects enjoy increased productivity when designing and analyzing systems using DSMLs tailored to the concepts, relationships, needs, and constraints of their particular domains. However, unlike “universal” modeling languages such as universal modeling language (UML) [4], which is standardized and evolved by the OMG [5], industrial DSMLs are best designed and maintained by the domain architects who employ them.

A necessary condition for the industrial adoption of DSMLs is the use of metaprogrammable tool suites that enable the reuse of complex tools for modeling, model transformation, model management, model analysis, and model integration over a wide range of domains. Today, industry managers are faced with a confounding array of tools and metamodeling languages which support the creation and use of DSMLs. The consequences of choosing one tool or metamodeling language over another are not immediately obvious. In this chapter, we examine technical solutions for metaprogrammability and analyze the relationship between metamodeling languages and metaprogrammable tool architectures. Specifically, we will provide a technical comparison of three metamodeling languages: MetaGME, Ecore, and Microsoft’s Domain Model Definition (DMD) language. Note that this list is by no means exhaustive; other metamodeling languages not discussed here include the metamodeling language (MML) [6] and those defined for the executable metamodeling facility (XMF) [7].

MetaGME is the native metamodeling language supported by the Generic Modeling Environment (GME) [8], a graphical metaprogrammable modeling tool [9]. GME is developed by the Institute for Software Integrated Systems (ISIS) of Vanderbilt University. MetaGME’s syntax is based on UML class diagrams [4] with object constraint language (OCL) constraints [10].

Ecore is similar to the EMOF flavor of the pending MOF 2 specification currently being revised by OMG. It serves as the metamodeling language of the eclipse modeling framework (EMF) [11]. Every Ecore metamodel may be equivalently represented through an Ecore dialect of XML model interchange (XMI) [12] or as Java source code. EMF was initially released in 2003 by IBM.

The DMD* language will be the metamodeling language supported by Microsoft’s forthcoming Visual Studio Domain-Specific Language Toolkit. This toolkit realizes the Software Factories vision for model-based design [13]. DMD is currently still prerelease—our work in this paper is based on the September 2005 beta of the DSL toolkit.

Metamodeling languages play a highly significant role in the metaprogrammable modeling tools, which support them because they are used to create domain-specific modeling configurations of those tools. However, model-based design technology is rapidly evolving, and tool designers may need to support multiple metamodeling languages side by side, change the metamodeling interface exposed by a tool suite, or otherwise enable the portability of models between metamodeling languages. Recent work [14,15] has shown that model-to-model transformations can enable the portability of models across tool suites; however, understanding the impact of the selection of a metamodeling language on the reusability of such a suite of tools is also an important issue. We argue that the consequences of choosing one metamodeling language over another are fairly minor, since all of the metamodeling languages support the same fundamental concepts for DSML design. Consequently, model-to-model transformation approaches can aid in retrofitting a metaprogrammable tool suite such that it supports a new metamodeling language.

This chapter is organized as follows: Section 33.2 defines metaprogramming, metaprogrammable tools, and metaprogrammable tool suites. Section 33.3 compares the abstract syntax of the three metamodeling languages briefly described above to establish their fundamental similarity. We also provide some analysis of the trade-offs made when selecting one language over another in terms of the convenience afforded by each language’s abstract syntax and the capabilities of the current supporting tools of each. Section 33.4 outlines a set of procedures and tools which can enable the support of a new metamodeling language by an existing metaprogrammable tool suite. The methods described depend on knowledge of the deep similarities between metamodeling languages established in Section 33.3. Section 33.5 presents our conclusion.

*The DMD label is not standardized by Microsoft, but we adopt here for lack of a better short-hand name for Microsoft’s forthcoming metamodeling language.

33.2 Modeling Tool Architectures and Metaprogrammability

This section defines metaprogramming, metaprogrammable tools, and metaprogrammable tool suites. Traditionally, metaprogramming implies the writing of programs that write or manipulate other programs (or themselves) as their data. Metaprogramming allows developers to produce a larger amount of code and increase productivity. Although the end purpose of metaprogramming in model-based design is still developer productivity, the means is slightly different. The task of metaprogramming in model-based design is to automatically configure and customize a class of generic modeling, model management, transformation, and analysis tools for use in a domain-specific context.

The primary hurdle with the domain-specific approach is that building a new language and modeling tool to support a narrowly used niche domain (e.g., a codesign environment specialized for a single type of automobile electronic control unit) might be unjustifiably expensive. In contrast, a general modeling tool with “universal” modeling concepts and components would lack the primary advantage offered by the domain-specific approach: dedicated, customized support for a wide variety of application domains. The solution is to use a highly configurable modeling tool which may be easily customized to support a unique environment for any given application domain. These are known as metaprogrammable modeling tools, because users must “program” the tool itself before they can use it to build models. The chief virtue of metaprogrammable tools is that they enable relatively quick and inexpensive development of domain-specific modeling environments (DSMEs)—especially when the tool customizations themselves can be easily specified through a model-based design process.

Experience with metaprogrammability indicates that metaprogrammable tools generally implement the following core elements:

- A *metaprogramming language* that defines the core concepts and operations for customizing the generic tool into a DSME. The role of the metaprogramming language is to dictate the configuration of a generic tool to support modeling in a specific domain. Since all metaprogrammable modeling tools need to perform similar tasks (i.e., store models, allow users to access and manipulate models, and transform models into some useful product), we might expect that the metaprogramming languages associated with different tools will have a common expressive capability in terms of entity-relationship modeling. The primary differences between metaprogramming languages from different tools generally result from the model presentation capabilities of the tools themselves.
- A *metamodeling language* (or metalanguage) used to model (at the minimum) DSML abstract syntax, which expresses the language concepts, and relationships and well-formedness rules [10]. The primary benefit metamodeling languages provide over metaprogramming languages is that metamodeling languages take a model-based approach to language specification—they operate at a higher level of abstraction from the tool. Traditionally, tools relied solely on metaprogramming languages for DSME specification. However, the community has come to realize that the interoperability of models between metaprogrammable tools suites can most easily be accomplished by agreeing on a standard metamodeling language which can abstract away the differences between various modeling tools. Consequently, the idea of decoupling the metamodeling language from the underlying metaprogramming concepts of the tool and adopting a standards-based metalanguage such as MOF [16] has come into practice. Interestingly, while all modeling tools have settled on the entity-relationship paradigm for expressing systems, there are a variety of different approaches to model presentation, broadly including
 - Full graphical—Users freely draw graphically complex entity-relationship models on a modeling palette. Model elements can be moved about the palette and can be visualized using domain-specific sprites, abstract forms, or even domain-appropriate animations.
 - Restricted graphical—Users see graphical representations of the model compositional structure (generally expressed as a tree-like containment hierarchy), but the visualization itself is not domain specific.
 - Textual.

The plurality of approaches to model presentation means that model presentation information is not generally portable between modeling tools. Consequently, the metalanguages closest to being true industry standards, MOF and Ecore, standardize no first-class constructs specifically modeling concrete syntax. Furthermore, XMI, the most widely adopted model-XML serialization language, defines no elements specifically for capturing presentation metadata. Tools which support domain-specific concrete syntax as well as abstract syntax capture concrete syntax in the metaprogramming language or even in a metamodeling language (as is the case with MetaGME) [9].

- A *Mapping* that overlays the DSML abstract syntax expressed using the metalanguage onto the syntax of the metaprogramming language supported by the generic (metaprogrammable) tool. The mapping process is generally referred to as metatransformation or metainterpretation.

Formally stated, let *MML* be a metamodeling language, *MPL* the metaprogramming language supported by the generic tool, and *DSML* a domain-specific modeling language. Also, let the notation $L_1 A L_2$ denote the abstract syntax of *L2* expressed using *L1*. The metatransformation can then be stated as follows:

$$MML T_{MPL} : MML A_{DSML} \rightarrow MPL A_{DSML} \quad (33.1)$$

Metaprogramming thus involves the following two steps:

1. Specify the DSML abstract syntax $MML A_{DSML}$ using a metamodel,
2. Apply the metatransformation $MML T_{MPL}$ to that metamodel

The resulting abstract syntax specification of *DSML* using *MPL* represents customization of the generic tool to support modeling in *DSML*.

Of course, simple metaprogrammable modeling tools as described above do not provide the full range of required capabilities for semantically rich model-based design. These tools must be integrated with helper tools, plug-ins, and languages to provide constraint management, model transformation, model interpretation, and other capabilities [17]. A constraint manager is responsible for enforcing the well-formedness rules of DSMLs (and consequently must be metaprogrammable). Model transformers translate models of one sort into models of another sort, or generate code from models. For example, model interpreters translate domain models into “executable” models to perform simulation, or into analysis input models for analysis tools. In this sense, model interpreters may be said to assign the behavioral semantics of the supported modeling language. The XMF provided by Xactium composes a set of tool-supported

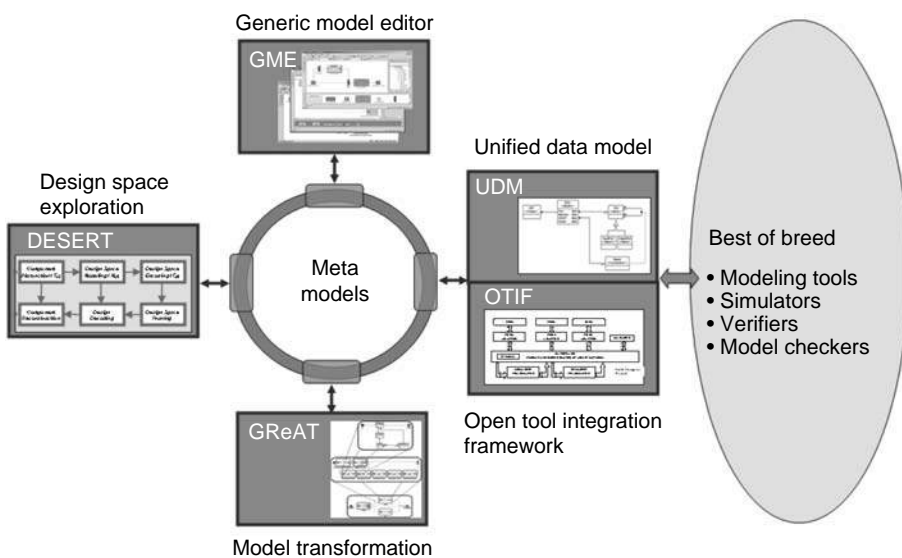


FIGURE 33.1 MIC Metaprogrammable tool suite.

languages to provide rich metamodeling capabilities capable of capturing the syntax and semantics of modeling languages. This set includes an abstract syntax modeling facility which supports an MOF-like language, a concrete syntax language, a constraint language, an action language for modeling behavioral semantics, and a model transformation language [7].

In the remainder of this section, we describe the MIC suite of metaprogrammable tools: GME, universal data model (UDM), graph rewriting and transformation (GReAT), and design space exploration tool (DESERT) (depicted in Figure 33.1). We elucidate the role of metaprogramming and metamodeling languages in each of these tools.

33.2.1 Metaprogrammable Modeling Tool—GME

GME is the MIC metaprogrammable graphical modeling tool [8]. It includes a graphical model builder, supports a variety of model persistence formats, and is supported by a suite of helper tools; its architecture is further depicted in Figure 33.2. The model builder component serves as the tool frontend which allows modelers to perform standard CRUD (create, request, update, destroy) operations on model objects which are stored GME's model repository. The model builder provides access to the concepts, relationships, composition principles, and representation formalisms of the supported modeling language. The model repository is physically realized via a relational database. GME uses OCL as its constraint specification language and includes a metaprogrammable plug-in for enforcing domain constraints.

The soul of GME, however, is its metaprogramming language, which is called GMeta. The semantics of this language are implemented in the GMeta component of GME's architecture as depicted in Figure 33.2. The GMeta component provides a generic interface to the database used to implement GME's model repository. It implements the following GMeta language concepts: Folders, Models, Atoms, Sets, References, Connections, ConnectionPoints, Attributes, Roles, and Aspect.* All of the model-manipulation operations in the GME model builder map to *domain-generic* macromanipulation operations on the objects in the GMeta component. For example, an object-creation operation in the model-builder maps to creation of an instance of one (or more) of the above concepts in the GMeta component of the model repository. Configuration and customization of GME via metaprogramming accomplishes the following:

- It specifies the mapping of domain modeling concepts into GMeta concepts. For example, domain relationships are specified as GMeta Connections.

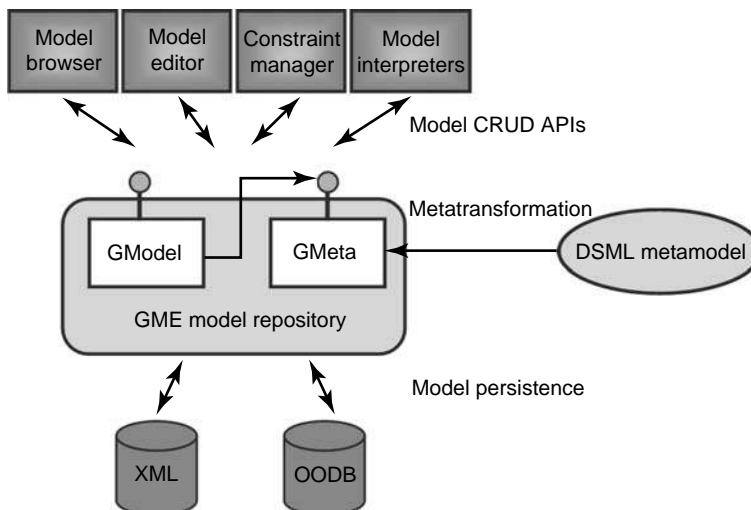


FIGURE 33.2 Modeling tool architecture: GME.

*We expand on these concepts in the next section.

- It specifies a suite of macrooperations for model manipulation. For example, suppose we wrote a metaprogram which configures GME to serve as an environment for modeling FSMs. We specify that the *State* domain concept has a Boolean attribute *IsInitial* which will specify whether a particular state in an FSM is the initial state. Then, creating a state instance in the model builder might result in the instantiation of a GMeta Atom and a GMeta Attribute in the GMeta component of the model repository.
- It constrains the available model operations according to domain well-formedness rules. This means that users of the language will not be able to specify domain models which defy the definitions of the domain constructs as set forth in the metaprogram.

GME currently supports MetaGME as its native metamodeling language. MetaGME is based on stereotyped UML class diagrams with OCL constraints, and serves as a layer of abstraction on top of GMeta. Users may metaprogram GME by specifying the abstract syntax of a DSML as a MetaGME stereotyped class diagram. The stereotypes in MetaGME refer to the concepts of GMeta listed above. Metatransformation maps the metamodel into these lower-level metaprogramming constructs. Formally, metatransformation in GME is a concretization of Equation 33.1 as follows:

$$MetaGME T_{GMeta} : MetaGME A_{DSML} \rightarrow GMeta A_{DSML} \quad (33.2)$$

Note that this transformation is metacircular in the sense that MetaGME can also be realized as a DSML, and GME can be metaprogrammed to provide a DSME for metamodeling.

$$MetaGME T_{GMeta} : MetaGME A_{MetaGME} \rightarrow GMeta A_{MetaGME} \quad (33.3)$$

33.2.2 Metaprogrammable Model Manipulation Management API—UDM

UDM is a tool framework and metaprogrammable API which can be configured to provide *domain-specific* programmatic access to an underlying network of model objects independent of the mechanism used to persist those model objects. UDM serves as a model access and manipulation layer, forming the “backbone” that model editing, database, and generator tools require [18]. It interoperates with and lends additional functionality to GME.* UDM uses UML class diagrams as its metamodeling language. Metaprogramming UDM thus involves specification of a DSML using UML class diagrams. Furthermore, just as GME internally represents domain models in its repository using the domain-generic GMeta constructs, UDM internally represents domain models using domain-generic UML constructs. Although UDM provides a generic UML-based API for accessing models, its chief virtue is to provide a domain-specific layer of abstraction over the object models of various underlying backend persistence technologies. UDM supports multiple such backends as shown in Figure 33.3.

With the GMeta backend, the model object network physically resides in GME’s relational internal object database. Direct access to this database is accomplished through the GMeta domain-generic API. With the XML backend, the model object network resides in an XML file which may be manipulated through the domain-generic DOM API. The generated UDM API for a DSML is a code-level realization of the UML metamodel used to specify the abstract syntax of the DSML. This domain-specific API is a wrapper facade for the domain-generic UML API, which in turn is a wrapper facade for the domain-generic API of each supported underlying model persistence mechanism. Consequently, different metatransformations are used to metaprogram UDM for each of the supported backends.

For example, the metatransformation for the GMeta backend may be formalized as

$$UML T_{GMeta} : UML A_{DSML} \rightarrow GMeta A_{DSML} \quad (33.4)$$

*GME actually supports several mechanisms for providing programmatic access to models, including the Builder Object Network (of which there are both C++ and Java versions). For the sake of brevity, we only discuss one.

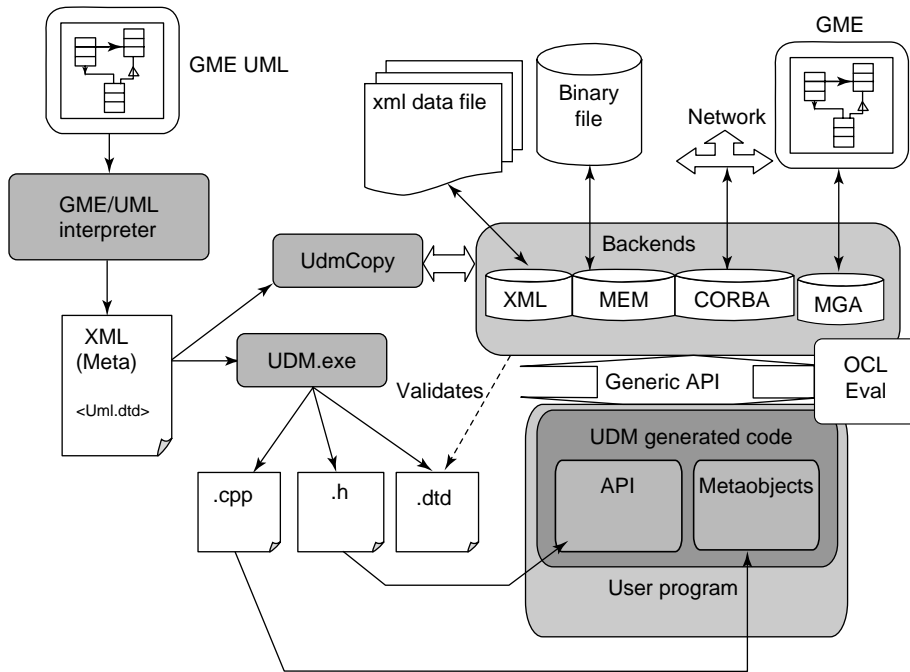


FIGURE 33.3 Universal data model framework.

It is often useful to construct a domain-specific UDM API, which supports a DSML originally specified using MetaGME rather than UDM's implementation of UML. Consequently, the UDM set of tools includes the MetaGME2UML model-to-model transformation:

$$\text{MetaGME}T_{\text{UML}} : \text{MetaGME}A_{\text{DSML}} \rightarrow \text{UML}A_{\text{DSML}} \quad (33.5)$$

Clearly, UDM was designed with an eye toward the manipulation of models irrespective of the platform on which they were developed or the mechanism used to persist them. Because of its relationship with UML, UDM even supports XMI [12] import and export of models and metamodels.

33.2.3 Metaprogrammable Design Space Exploration—DESERT

When large-scale systems are constructed, in the early design phases it is often unclear what implementation choices could be used to achieve the required performance. In embedded systems, frequently multiple implementations are available for components (e.g., software on a general purpose processor, software on a DSP, FPGA, or an ASIC), and it is not obvious how to make a choice, if the number of components is large. Another metaprogrammable MIC tool can assist in this process. This tool is called DESERT. DESERT expects that the DSML used allows the expression of alternatives for components (the design space) in a complex model.

Once a design space is modeled, one can attach applicability conditions to the design alternatives. These conditions are symbolic logical expressions that describe when a particular alternative is to be chosen. Conditions could also link alternatives in different components via implication. One example for this feature is “if alternative A is chosen in component C1, then alternative X must be chosen in component C2.” During the design process, engineers want to evaluate alternative designs, which are constrained by high-level design parameters such as latency, jitter, and power consumption. Note that these parameters can also be expressed as symbolic logic expressions. DESERT provides an environment in which the design space can be pruned and alternatives rapidly generated and evaluated.

DESERT consumes two types of models: component models (which are abstract models of simple components) and design space models (which contain alternatives). Note that for design space exploration the internals of simple components are not interesting, only a “skeleton” of these components is needed. The design space models reference these skeletons. DESERT uses a symbolic encoding technique to represent the design space that is based on ordered binary decision diagrams (OBDDs) [19]. OBDDs are also used to represent applicability constraints as well as design parameters. The conversion to OBDDs happens in an encoding module.

Once the symbolic representation is constructed, the designer can select which design parameters to apply to the design space and thus “prune away” unsuitable choices and alternatives. This pruning is controlled by the designer, but it is done on the symbolic representation: the OBDD structures. Once the pruning is finished, the designer is left with 0, 1, or more than one designs. 0 means that no combination of choices could satisfy the design parameters, 1 means a single solution is available, and more than one means multiple alternatives are available that cannot be further pruned based on the available information. This latter case often means that other methods must be used to evaluate the alternatives, for example, simulation. The result of the pruning is in symbolic form, but it can be easily decoded and one (or more) appropriate (hierarchical) model structure reconstructed from the result.

DESERT is metaprogrammable as it can be configured to work with various DSMLs (however, all of them should be capable of representing alternatives and constraints). The metaprogramming here happens in two stages: one for the model skeleton generation, and the other one for the model reconstruction. DESERT has been used to implement domain-specific DESERT for embedded control systems and embedded signal processing systems.

33.2.4 Metaprogrammable Model Transformations—GReAT

GReAT is the MIC model-to-model transformation language [20,21]. GReAT supports the development of graphical language semantic translators using graph transformations. These translators can convert models of one domain into models of another domain. GReAT transformations are actually graphically expressed transformation algorithms consisting of partially ordered sets of primitive transformation rules. To express these algorithms, GReAT has three sublanguages: one for model instance pattern specification, one for graph transformation, and one for flow control. The GReAT execution engine takes as input a source domain metamodel, a destination domain metamodel, a set of mapping rules, and an input domain model, and then executes the mapping rules on the input domain model to generate an output domain model.

Each mapping rule is specified using model instance pattern graphs. These graphs are defined using associated instances of the modeling constructs defined in the source and destination metamodels. Each instance in a pattern graph can play one of the following three roles:

- *Bind*: Match objects in the graph.
- *Delete*: Match objects in the graph and then delete them from the graph.
- *New*: Create new objects provided all of the objects marked Bind or Delete in the pattern graph match successfully.

The execution of a primitive rule involves matching each of its constituent pattern objects having the roles Bind or Delete with objects in the input and output domain model. If the pattern matching is successful, then for each match the pattern objects marked Delete are deleted and then the objects marked New are created. The execution of a rule can also be constrained or augmented by Guards and AttributeMappings, which are specified using a textual scripting language.

GReAT’s third sublanguage governs control flow. During execution, the flow of control can change from one potentially executable rule to another based on the patterns matched (or not matched) in a rule. Flow control allows for conditional processing of input graphs. Furthermore, a graph transformation’s efficiency may be increased by passing bindings from one rule to another along input and output ports to lessen the search space on a graph.

Ultimately, GReAT transformation models are used to generate C++ code which uses automatically generated domain-specific UDM APIs for the source and destination metamodels to programmatically execute model-to-model transformations. Consequently, GReAT inherits its metaprogrammability from UDM. The source and destination metamodels accepted by GReAT are expressed using UDM-style UML class diagrams.

Formally stated, let SRC and DST be respectfully the source and destination metamodels of GReAT Transformation $_{SRC}T_{DST}$, and let IN and OUT be arbitrary input and output domain models expressed using the languages defined by SRC and DST . Then, assuming that the GMeta backend is always used for persistent model storage, any execution of $_{SRC}T_{DST}$ may be formalized as the following series of transformations:

$$\begin{aligned} IN T_{OUT} : & (((GMeta A_{IN} \rightarrow UML A_{IN}) \rightarrow SRC A_{IN}) \\ & \rightarrow DST A_{OUT}) \rightarrow UML A_{OUT}) \rightarrow GMeta A_{OUT} \end{aligned} \quad (33.6)$$

To summarize, the MIC metaprogrammable tool suite allows users to

- Specify a domain-specific modeling language, include a concrete and abstract syntax and domain constraints using GME and MetaGME.
- Build system models using the metamodeled DSML and verify that the models do not violate the domain constraints.
- Construct model interpreters to parse and analyze the system models using UDM.
- Transform the system models into models expressed using a different DSML using GReAT; a specialized case of this is generating code directly from models to begin implementing the system.
- Formally express the semantics of the DSML by mapping it onto another modeling language that expresses a formal model of computation using GReAT.
- Perform design-space exploration on the modeled system using DESERT.
- Serialize the models to or import models from XMI using UDM.

The MIC tool suite supports full-featured, semantically rich model-based design. Furthermore, we note that it is a framework which already utilizes multiple metamodeling languages mediated by a model-to-model transformation: MetaGME and UDM UML mediated by the MetaGME2UML transformation.

33.3 A Comparison of Metamodeling Languages

This section provides a brief overview of three metamodeling languages: MetaGME, Ecore, and Microsoft's Domain Model Designer (DMD) language. We compare these languages to draw out the close kinships and similar expressive power they share via their common application of UML-like object-oriented principles to abstract syntax modeling. We phrase their differences in terms of trade-offs for selecting one language over another.

We discuss how each of these languages captures in some way a common core set of metamodeling constructs derived from UML [4]:

- *Classes* are types whose instances have identity, state, and an interface. They are used to specify the concepts of a modeling language. The state of a class is expressed by its attributes and its interface is defined by the associations in which it participates. Class definitions may be extended through the use of inheritance. Some languages support multiple inheritance; some support only single inheritance.
- *Associations* describe relationships between classes, including composition. Associations may be binary or n -ary, and may include attributes or not, depending on the language.
- *Data Types* are noninstantiable types with no object identity, such as primitive types and enumerations, which are used to type attributes. All three metamodeling languages we consider support a common core set of primitive data types, and some also support additional data types. The common set includes strings, integers, doubles, Booleans, and enumerations.

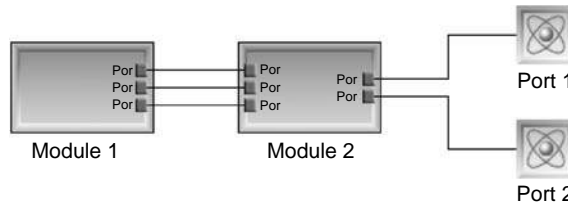


FIGURE 33.5 Modular interconnect in GME.

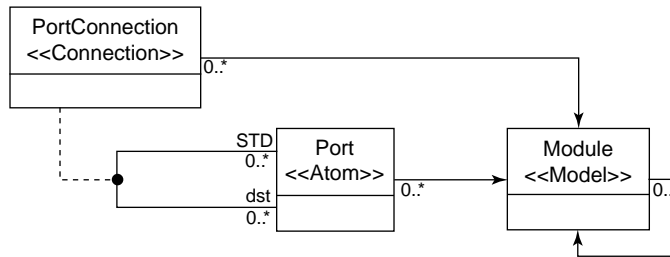


FIGURE 33.6 Metamodel fragment specifying modular interconnection.

- *Aspects* are classes which group other classes together (through a special metalevel relationship) into logical visibility partitions to present different views of a model.

GME provides special default visualizations for languages that use various patterns of these stereotypes. For example, the default visualization for the modular interconnect pattern appears in Figure 33.5. The metamodel fragment used to specify this visualization appears in Figure 33.6, where the Port class is composed into the Module class as a port. This visualization could be further customized by specifying icons for Module and Port and tailoring the appearance of PortConnection in the metamodel.

Finally, MetaGME provides robust support for metamodel composition. The GME tool itself allows the importation of one model into another, and MetaGME includes a special set of “class equivalence” operators for specifying the join-points of the composed metamodels. It also defines two additional inheritance operators to aid in class reuse: implementation inheritance and interface inheritance [23]. Implementation inheritance allows a derived class to inherit only the attributes and component classes of its base type, while interface inheritance allows a derived class to inherit only the noncomposition associations of its base type.

33.3.2 Ecore

Ecore is the metamodeling language for the EMF [11]. The DSMEs built using EMF run as plug-ins for the Eclipse platform. The EMF tool suite includes the following tools:

- The Ecore language for specifying the abstract syntax of DSMLs. The framework supports import and export with an Ecore-flavored dialect of XMI (it can also import standard MOF XMI).
- EMF.Edit, which includes a domain-generic set of classes used to construct DSMEs.
- EMF.Codegen, the metatranslator which renders a DSME and a domain-specific model CRUD API from an Ecore metamodel.
- A domain-generic API for accessing and manipulating EMF models.

The DSMEs produced by EMF “out of the box” all share a domain-generic hierarchy-based tree visualization, and EMF does not provide tool support for modeling concrete syntax. Complicated domain-specific visualizations can be created by extending the generated DSME or by hand-coding an editor using

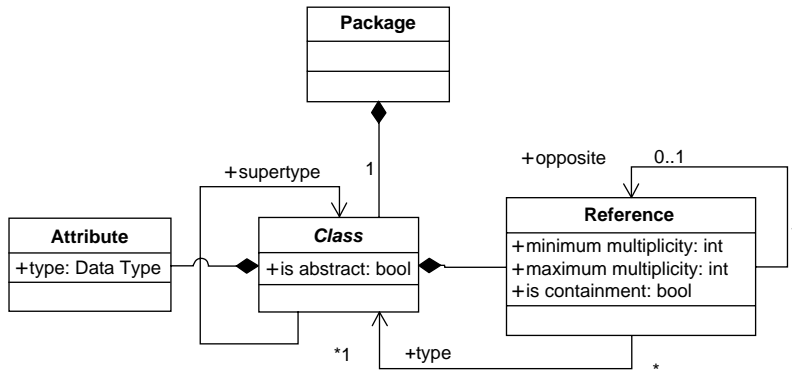


FIGURE 33.7 Simplified version of the Ecore Metamodel.

the Eclipse graphical editing framework (GEF). EMF lacks native tool support for modeling domain constraints, model migration, and multispect (or multiview) modeling. It is possible to perform metamodel composition, however, by importing one metamodel into another. Although it is not possible to capture domain constraints in an Ecore metamodel, it is possible to hand-weave constraints into the Java code generated from a metamodel using a Java-based OCL library developed at Kent University [24].

The Ecore metamodeling language itself strongly resembles the EMOF flavor of the MOF 2.0 specification [25]. However, EMF takes a more bottom-up, code-centric, and Java-specific view of modeling than is reflected in the MOF standard, which results in a few simplifications in Ecore. Ecore does not support class nesting or a first-class association construct, but it does support multiple inheritance.* A simplified UML class diagram for the Ecore metamodel (as it relates to domain-specific modeling) is provided in Figure 33.7.†

Because Ecore lacks a first-class association construct, association is expressed using references. Each class forming one end of a logical association may have a reference to the class which forms the other end of the logical association. References may be used to represent both symmetric and asymmetric association; in the symmetric case, each end class possesses a reference to the other end class. In asymmetric association, only one of the end classes possesses a reference to the other end class. Ecore's representation of associations is sufficiently expressive to capture the information relevant to DSML design: class relation, association roles, composition, and navigability. Clearly, however, Ecore cannot represent associations with state. This certainly makes many modeling languages more tedious to define, but does not decrease the relative expressive power of the Ecore. Every association with attributes can be modeled as two associations which link to a Class with the same attributes.

Through its close relationship with Java, Ecore enables a vast array of data types. In addition to all of the usual primitive types, Ecore allows bytes, shorts, and in general any Java class to be wrapped as a data type. Many of these additional types, such as collections, structures, and arbitrary classes, are of limited use in the context of graphical DSMLs because of the difficulty of providing a simple graphical interface for setting the values of attributes with complex types. For the most part, these only come in to play when using Ecore itself as a DSML for defining logical data models or database schemas.

33.3.3 Microsoft Domain Model Designer

Microsoft Domain Model Designer is part of the Microsoft Domain-Specific Languages tool suite [26]. This new tool suite will realize the Software Factories vision of a workbench for creating, editing,

*When an Ecore class with multiple base types is serialized to Java, the corresponding class extends one of those base classes and implements the others as mix-in interfaces.

†The Ecore metamodeling constructs are all more properly named with “E” preceding the construct title (e.g., “EClass” instead of “Class”). We omit these preceding “Es.”

visualizing, and using domain-specific data for automating the Windows-based enterprise software development process. The DSMEs produced by MS DSL plug-in to the Visual Studio. For the sake of clarity, we should point out that with its new tool, Microsoft has introduced new terminology for talking about familiar domain-specific modeling concepts:

- *Domain Model*: a metamodel.
- *Designer*: a DSME.
- *Designer Definition*: an XML-based metaprogramming specification for the DSL tool suite which includes concrete syntax and syntactic mapping information.

Currently, the metaprogrammable DSL tool suite consists of the following major components:

- The Domain Model Designer metalanguage/tool, which is used to formulate DSL abstract syntax.
- The Designer Definition language, which is used to specify the DSL concrete syntax, and some primitive language constraints.
- A metatranslator, which generates from a domain model and a designer definition a designer and a domain-specific API for model access and manipulation.
- The metadata facility (MDF) API, which provides generic CRUD access to in-memory model storage.
- A tool for creating template-based model parsers with embedded C# scripts which may be used for code generation.

Because of the current immaturity of this framework, many supporting tools remain unimplemented. Microsoft eventually plans to provide tool support DSL constraint specification, model migration, language composition, and multispect (or multiview) modeling.

Although Microsoft provides no explicit meta-metamodel, it is easy to derive one from experimentation with their tools. We provide a simplified visualization of this meta-metamodel using a UML class diagram in Figure 33.8.

Notice that the language does not allow multiple inheritance, but does include the notion of stateful associations (Relationships). These Relationships come in two varieties. Reference Relationships merely denote association, while Embedding Relationships denote composition. DMD also captures a variety

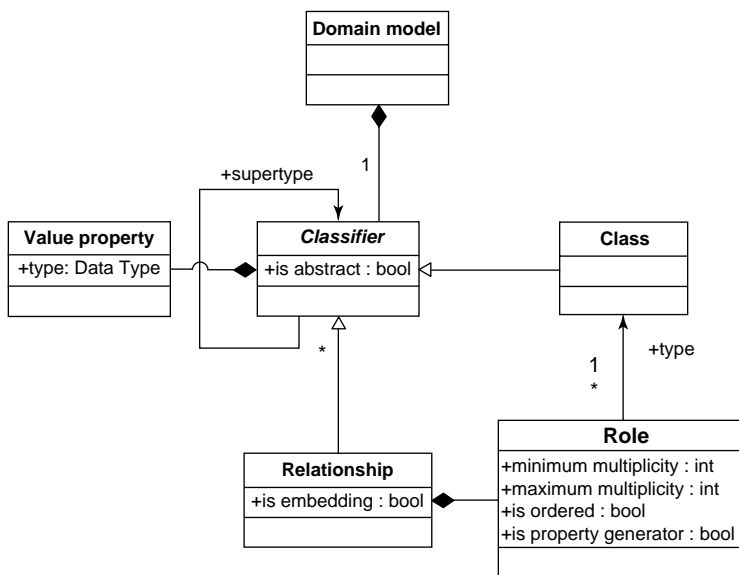


FIGURE 33.8 Simplified version of the Microsoft Domain Model Designer Metamodel.

of data types, including the standard set of primitive datatypes, longs, DateTimes, and GUIDs. As stated above, the Microsoft DSL tools currently do not provide a modeling interface for capture concrete syntax—that is done in XML through a designer definition.

33.3.4 Language Differentiators

Recall that in Section 33.2 we stated that the role of the metamodeling language is twofold: it must express the concepts and relationships of the modeled domain and it must allow users to dictate modeling-tool-specific functionality. We have demonstrated that all three of the languages have similar expressive power for defining the abstract syntax of DSMLs because they all reuse a common core set of metamodeling constructs derived from UML class diagrams. The primary differentiators of the languages are the mechanisms which they expose at the metamodeling level for specifying concrete syntax, the robustness and focus of their respective supporting tool suites (GME, EMF, and MS DSL), and support for special language features such as constraint specification and metamodel composition.

33.4 Relating Metamodeling Languages and Metaprogrammable Tools

In this section, we outline the tools and procedures which may be used to support another metamodeling language alongside the native metamodeling language of a metaprogrammable tool suite. Our technique depends on model-to-model transformation. We use the the MIC tool suite, the UDM framework [18], and our previous work of integrating a MOF metamodeling environment into GME as illustrative examples and proofs-of-concept [27].

In Section 33.3, we stated that the primary differentiator between two metamodeling languages is not their expressive power for defining DSML abstract syntax. Indeed, for the sole purpose of modeling DSMLs, metamodeling languages are differentiated mainly by the support they provide for specifying tool-specific information such as concrete syntax. Consequently, the primary issue when adapting a metaprogrammable tool suite to support a new metamodeling language is merely expressing this tool-specific information in the new language. There is no modeling language whose abstract syntax can be expressed in one of MetaGME, Ecore, or MS DMD, and not in both of the other two.

Generalizing from Figure 33.2, we see that metaprogrammable modeling frameworks expose three interfaces where users are exposed to the concepts which compose the metamodeling language supported by the tool: the metaprogramming interface, the model persistence interface, and the domain-generic model CRUD API.

33.4.1 Example: UDM

In the context of the MIC tool suite, the UDM framework described in Section 33.2 demonstrates the use of model-to-model transformation to enable GME to provide a model access and manipulation of API based on a different metamodeling language (UDM UML) than the one it natively supports (MetaGME). Furthermore, recall that UDM incorporates the capability to export models into a variety of different formats including the GME binary format and flat files of XMI (similar to those exported by EMF). UDM accomplishes this through the use of a series of model-to-model translations, including MetaGME2UML and a series of translators between the in-memory UDM UML model representation and the various supported backends (Figure 33.3). Data abstraction layers such as UDM are key enabling technologies for model portability across metamodeling languages. However, no tool currently in the UDM framework is capable of metaprogramming GME to support a DSME.

33.4.2 Example: GME-MOF

GME-MOF is an alternative MOF-based metamodeling environment for GME. It allows users to build metamodels to metaprogram GME using MOF [27]. Noting that MOF itself is merely a DSML for defining

metamodels, we used a bootstrapping process to implement MOF for GME. Since MOF is specified meta-circularly using the same set of object-oriented metamodeling constructs inherited from UML which form the core of MetaGME, MetaGME possesses sufficient expressive power to fully model MOF. Accordingly, we bootstrapped our MOF implementation by modeling MOF using MetaGME's constructs and creating a MOF DSML for GME.

MOF standardizes no method for specifying language concrete syntax, but model visualization is a key element of the GME modeling experience. So, we needed to find a natural way to allow MOF to express the visualization configurations realizable by GME. We accomplished this by augmenting MOF classes and associations with a set of attributes mapping each MOF class onto a MetaGME stereotype (e.g., FCO or Atom) and each association onto either a MetaGME Connection, set membership relationship, or reference/referent relationship. Additional attributes allow users to specify icons for DSML language constructs and tailor connection visualizations.*

Next, we needed to implement the metatransformation $MMLT_{MPL}$ defined in Equation 33.1 to transform MOF metamodels into valid metaprogramming configurations of GME. Because MetaGME by design already reflects the full range of configurations which can be realized by GME (including model visualization configurations), the easiest way to build a translation tool is by defining a model-to-model transformation algorithm from MOF-specified metamodels into MetaGME-specified metamodels, then reusing the metatransformer built for MetaGME. Such a metamodel transformation tool may be specified quite naturally due to the fundamental similarity of MetaGME and MOF. GREAT [20,21] was the natural choice for implementing the metamodel translation algorithm.

The shaded components in Figure 33.9 represent the new facilities required to implement MOF for GME: $MetaGMEMM_{MOF}$ is the MetaGME-specified MOF model and T_2 is the MOF2MetaGME transformation algorithm from MOF-specified metamodels to MetaGME-specified metamodels. T_1 is MetaGME's

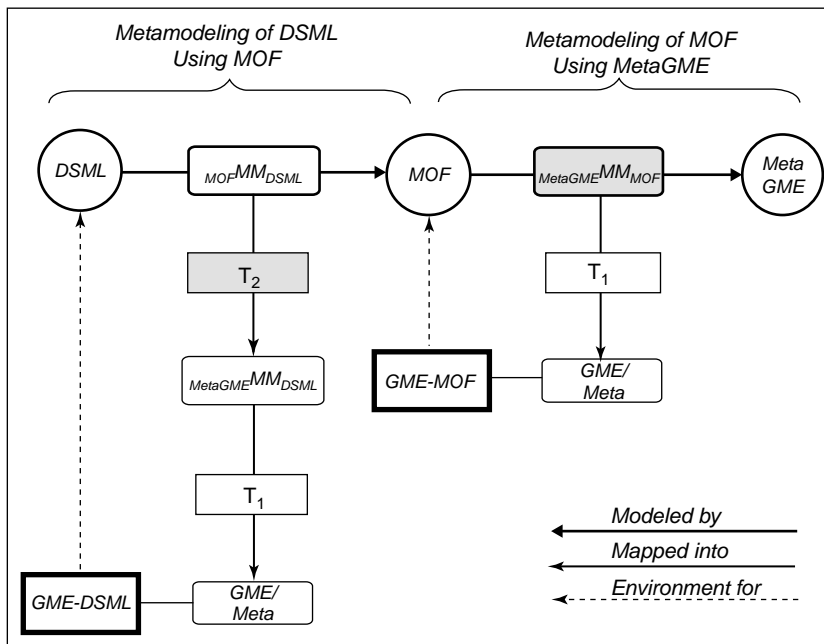


FIGURE 33.9 Building the MOF-based metamodeling environment.

*Because all of this information is captured through attributes, it could also have been captured using the MOF Tag construct. However, this would have been an extremely tedious solution for users, who would need to create tags to specify the visualization of every element in a metamodel.

metainterpreter, the translator which generates GME configuration files from metamodels. These configuration files customize GME's graphical model builder, constraint manager, and model database to support the modeled DSME. Notice how our approach allows us to reuse T_1 in the process of converting $MOFMM_{DSML}$ (a MOF-specified metamodel of some DSML) into a valid metaprogramming configuration for GME.

So, we can concretize Equation 33.1 for GME-MOF:

$$MOFT_{GMeta} : MOFA_{DSML} \rightarrow MetaGMEA_{DSML} \rightarrow GMetaA_{DSML} \quad (33.7)$$

Generalizing from GME-MOF, we can see how a layered approach combined with model-to-model transformation can enable the reuse of the existing metamodeling infrastructure of a metaprogrammable modeling tool in supporting a new metamodeling language. However, because the modeling community has not settled on any method for porting model visualization information across tools, tool developers must find ways of exposing an interface for DSME visualization configuration in each new metamodeling environment to be supported.

33.5 Conclusion

Metaprogrammable modeling tools require metamodeling languages which are expressive enough both to model DSML abstract syntax and to address tool-specific concerns such as concrete syntax. We have defined metaprogrammability, metamodeling, and metaprogrammable tool suites. Furthermore, we have examined three metamodeling languages and shown that these languages have very similar expressive power for DSML abstract syntax specification. Essentially, DSML abstract syntax is tool-agnostic, which means that it is possible to build equivalent DSMLs incorporating the same domain concepts and addressing the same concerns using different metamodeling languages in different metaprogrammable modeling tools. This implies that the basis for selecting a model-based design tool suite should not be the metamodeling language supported by the tools—it should be the capabilities of the tools themselves. Furthermore, a metaprogrammable tool suite can be bootstrapped to support alternative metamodeling languages through a layered approach which combines the use of metamodeling and model-to-model transformation. This bootstrapping process merely implements new metamodeling languages as layers of abstraction above the native metamodeling language supported by the tool. One of the chief virtues of this approach is that it does not break compatibility with legacy DSMLs and tools which depend on the old metamodeling environment. Another virtue is that it enables the possibility of designing and supporting all of the metamodeling languages side by side on a single tool and easily exporting metamodels from one language to another using a common data abstraction layer.

References

1. Object Management Group. Model Driven Architecture. Available from: www.omg.org/mda/.
2. Frankel D. *Model Driven Architecture*. OMG Press, New York, 2003.
3. The ESCHER Institute Available from: www.escherinstitute.org.
4. Object Management Group. *UML*. Available from: www.omg.org/UML/.
5. Object Management Group. Available from: www.omg.org.
6. Alvarez J., Evans A., and Sammut P. MML and the Metamodel Architecture, January 2001. Available from: www.cs.york.ac.uk/puml/mmf/SammutWTUML.pdf.
7. Clark T., Evans A., Sammut P., and Willans J. *Applied Metamodelling: A Foundation for Language Driven Development v 0.1*. Xactium, 2004. Available from: www.xactium.com.
8. Ledecz A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., and Sprinkle J. Composing Domain-Specific Design Environments. *IEEE Computer Magazine*, pp. 44–51, November 1997.

9. Ledeczki A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., and Volgyesi P. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.
10. Object Management Group. *UML 2.0 OCL Specification*, 2003. Available from: www.omg.org/docs/ptc/03-10-14.pdf.
11. EMF. The Eclipse Modeling Framework (EMF) Overview. Available from: download.eclipse.org/tools/emf/scripts/docs.php?doc=references/overview/EMF.html.
12. Object Management Group. *XML Metadata Interchange (XMI) Specification v2.0*, 2003. Available from: www.omg.org/docs/formal/03-05-02.pdf.
13. Greenfield J., Short K., Cook S., Kent S., and Crupi J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. Wiley, New York, 2004.
14. Bzivin J., Brunette C., Chevrel R., Jouault F., and Kurtev I. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *OOPSLA International Workshop on Software Factories*, San Diego, California, October 2005. Available from: www.softwarefactories.com/workshops/OOPSLA-2005/Papers/Bezivin.pdf.
15. Bzivin J., Brunette C., Chevrel R., Jouault F., and Kurtev I. Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). In *OOPSLA Workshop on Best Practices for Model Driven Software Development*, San Diego, California, October 2005. Available from: www.softmetaware.com/oopsla2005/bezivin2.pdf.
16. Object Management Group. *Meta Object Facility Specification v1.4*, 2002. Available from: www.omg.org/docs/formal/02-04-03.pdf.
17. Ledeczki A., Maroti M., Karsai G., and Nordstrom G. Metaprogrammable Toolkit for Model-Integrated Computing. In *Engineering of Computer Based Systems (ECBS)*, Nashville, TN, March 1999, pp. 311–317.
18. Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., and Karsai G. UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling*, 2003.
19. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3): 293–318, 1992.
20. Agrawal A., Karsai G., and Ledeczki A. An End-to-End Domain-Driven Development Framework. In *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.
21. Karsai G., Agrawal A., and Shi F. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11): 1296–1321, 2003.
22. Institute for Software Integrated Systems. *GME 4 Users Manual Version 4.0*, 2004. Available from: www.omg.org/docs/ad/00-09-02.pdf.
23. Ledeczki A., Nordstrom G., Karsai G., Volgyesi P., and Maroti M. On Metamodel Composition. In *IEEE CCA*, Mexico City, May 2001.
24. Wahler M. Using OCL to Interrogate Your EMF Model, August 2004. Available from: www.zurich.ibm.com/~wah/doc/emf-ocl/index.html.
25. Object Management Group. *Meta Object Facility Specification v2.0*, 2002. Available from: www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf.
26. Microsoft. *Microsoft Domain-Specific Language (DSL) Tools, September 2005 CTP Release for Visual Studio 2005 Beta 2*, 2005. Available from: <http://msdn2.microsoft.com/en-us/vstudio>.
27. Emerson M., Sztipanovits J., and Bapty T. A MOF-based metamodeling environment. *Journal of Universal Computer Science*, 10(10): 1357–1382, 2004.

34

Hardware/Software Codesign

Wayne Wolf
Princeton University

34.1	Introduction	34-1
34.2	Hardware/Software Partitioning Algorithms	34-2
34.3	Cosynthesis Algorithms	34-4
34.4	CPU Customization	34-5
34.5	Codesign and System Design	34-6
34.6	Summary	34-7

34.1 Introduction

Hardware/software codesign [14] is the simultaneous design of the hardware and software components of a digital system. There have traditionally been two reasons why the system designers have separated the design of hardware and software. First, they may build their systems from standard hardware components; the hardware designers have no knowledge of the end use to which their components will be put. Second, hardware and software teams may be organizationally separated. However, a number of pressures have led many organizations to adopt hardware/software codesign techniques.

Traditional software engineering emphasizes functional requirements on systems—that is, their input/output behavior. Many embedded computing systems have complex functional requirements—consider, for example, the functional specification of a multimedia compression system. But embedded systems must often meet several types of nonfunctional requirements as well: they must provide their results in real time; they must operate at low power and energy levels; they must be inexpensive to produce; and they must be designed quickly. If only one of these criteria applied, the design task might be straightforward. But as more constraints are put on the end product, the design becomes harder. To meet those constraints, more degrees of freedom must be opened up in the design process to allow new ways to meet the system requirements. Hardware/software codesign is the product of this process toward more flexible design processes to meet complex constraints.

To meet multiple, often conflicting requirements, embedded computing systems often use architectures that are heterogeneous in both their hardware and software architectures. General-purpose systems typically use regular architectures to provide a simpler programming model. Heterogeneous architectures sacrifice some ease of programming to improve the nonfunctional requirements of the system. One way to look at heterogeneous architectures is to start with a regular machine and take out hardware and software components that you do not need or can get along without. This process reduces the manufacturing cost of the system. It also reduces the power consumption of the system.

What may be less obvious is that heterogeneous systems are often the best way to assure consistent real-time behavior. A regular architecture allows many different parts of the system to access resources. It often becomes difficult or impossible to analyze the behavior of the system to determine how long a real-time operation will take. If we restrict the number of components that can access a resource, we can more accurately bound the system's temporal behavior. Heterogeneous architectures are not simply

an attempt to reduce cost but a fundamental approach to real-time system design. Because the hardware architecture ultimately determines the performance of the software that runs on top of the hardware, hardware/software codesign gives us a better handle on how to design real-time systems.

Hardware/software codesign is a family of design methodologies. Methodology is particularly important in embedded system design. General-purpose computers are designed by small teams for very large and varied markets. The designers of general-purpose computers have traditionally used loose methodologies to design their computers, instead ascribing a large role to invention. If we need to design a small number of computer systems, simple and loose methodologies are quite feasible. But embedded computers are used in a huge number of products and we need to design a steady stream of embedded systems. The results of that design process must be predictable. Not only must the system meet its functional and nonfunctional goals, but the amount of time and resources required to design the system must be reasonable. Well-understood design methodologies, whether they rely on computer-aided design tools or on manual processes, help ensure that we can meet all the goals of our project.

The earliest research in embedded computing was on real-time computing, starting with the landmark paper of Liu and Layland in the early 1970s. Hardware/software codesign emerged as a term and a topic in the early 1990s. At that time, a number of systems were designed that used a CPU to control several ASICs that were attached to the CPU bus. Hardware/software cosynthesis algorithms helped to determine what tasks should go on the ASICs versus the CPU. Hardware/software cosimulation systems were designed to simulate such systems and coordinate several different types of simulators. Hardware/software codesign has grown over the past 15 years to handle both more sophisticated components (e.g., pipelined processors and memory hierarchies) and more complex architectures (e.g., multiprocessors and CORBA).

Embedded system designers typically speak of performing an operation in software or in hardware. This is useful shorthand, but it is important to keep in mind that software does not exist in isolation—it can only execute on a hardware platform. The choice of hardware platform, including CPU, caching, multiprocessing features, etc., determines many of the execution characteristics of the software.

In the next section, we will discuss some basic principles of hardware/software cosynthesis. We will then look at several cosynthesis systems in more detail. Next, we will briefly consider CPU customization as a technique for developing custom platforms. We will end with a discussion of design verification and codesign.

34.2 Hardware/Software Partitioning Algorithms

Computer system design problems are often divided into four subproblems:

- Partitioning divides the functional specification into smaller units.
- Scheduling determines the times at which operations are performed.
- Allocation chooses the functional units that perform each operation.
- Mapping refines allocation to determine the detailed characteristics of the functional units.

These subproblems interact and the way these interactions must be handled vary from domain to domain.

These four subproblems are useful in codesign, but partitioning is generally used with a somewhat different meaning. Hardware/software partitioning is a particular form of hardware/software cosynthesis. As shown in Figure 34.1, the target architecture for hardware/software partitioning is a bus-based CPU system with one or more accelerators attached to the bus to perform specialized operations. The accelerators are separate devices, unlike coprocessors that are assigned opcodes and controlled by the CPU's execution unit. Hardware/software partitioning determines what functions should be performed by the accelerators (allocation and mapping) and schedules those operations. This operation is called partitioning because it bears some resemblance to graph partitioning—the CPU bus acts as the boundary between the two partitions and the algorithm determines which side of the boundary should hold each operation. Hardware/software partitioning can also benefit from dividing the functional specification into smaller units that can be more efficiently scheduled and allocated.

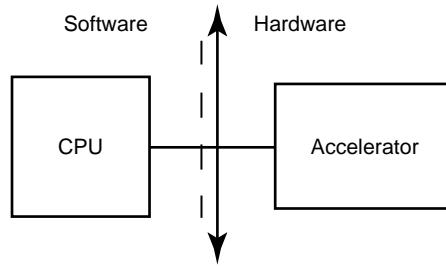


FIGURE 34.1 A bus-based CPU/accelerator system.

Platform FPGAs have emerged as an important implementation medium for hardware/software codesign. Platform FPGAs generally include both FPGA fabrics and high-performance CPUs; small CPUs can also be implemented directly in the FPGA fabric. Since they are preexisting chips, platform FPGAs allow codesign to be used in both low-volume and higher-volume applications.

COSYMA [6] was an early and popular hardware/software partitioning system. The algorithm that COSYMA uses helps us illustrate some important points about codesign. COSYMA was designed to implement systems with computationally intensive nested loops; many of the early examples were video systems. The basic problem is to determine how much of the nest of loops should be pulled from the CPU into a hardware accelerator.

First, we need to estimate the properties of candidate designs, both the components and the complete system. We are interested in the performance of both hardware and software components. COSYMA measured the hardware size but not the software size. COSYMA used measurement techniques for both hardware and software cost estimation. It used high-level synthesis algorithms to create a register-transfer design from the specification of a candidate hardware module. The number of clock cycles required to execute the function can be measured exactly from the register transfer, while the clock cycle period and the physical size of the final ASIC implementation could then be estimated from the register-transfers design. The performance of a software module was measured by compiling the code onto the target CPU, executing the code, and measuring its performance using instrumentation.

Analyzing the system behavior requires a model of the interaction between the components. COSYMA uses a C^X model, which is a directed graph that extends C with task and timing constructs.

COSYMA searches the design space of possible hardware/software partitions to find a good design. It starts with all operations performed in software and moves some operations to the hardware partition. Its search algorithm generally moves from the inner loops to the outer loops as it puts more operations in hardware.

At each step, COSYMA needs to estimate the speedup derived by moving the chosen operation or operations from software to hardware. Speedup depends on the relative execution time of the operations in hardware and software, but it must also take into account the time required to get data into and out of the accelerator. The basic speedup formula used by COSYMA can be written as

$$S = n[T_{SW}(T_{HW} + T_{in} + T_{out})] \quad (34.1)$$

where S is the total speedup, n the number of loop iterations performed, T_{SW} and T_{HW} the execution times in software and hardware implementations, T_{in} the time required to move the required data into the accelerator, and T_{out} the time required to move the results out of the accelerator. The search of the design space must be able to find a set of operations that can be sped up and that do not require excessive input and output. Loop kernels are a good target for this style of optimization because the values in a loop typically encompass a small working set with enough reuse to enable acceleration.

COSYMA uses high-level synthesis to estimate the cost of a hardware implementation and execution measurements to determine the execution time of software. COSYMA uses simulated annealing to search the design space. Because simulated annealing generates a large number of design points, COSYMA

uses a two-phase method to reduce the CPU time required for speedup and cost estimation. It uses fast estimators to roughly estimate costs for some number of design points. It periodically uses more sophisticated estimation mechanisms to generate better estimates and recalibrate the fast estimators.

Vulcan [8] is another early and well-known hardware/software partitioning system. Vulcan divides the behavioral description into tasks that communicate at the start and end of their execution. Its search algorithm takes the opposite approach—all tasks start in hardware and are moved to software to reduce system cost.

Different hardware/software partitioning algorithms use different objective functions, depending on the type of search algorithm they use. Vulcan and COSYMA both minimize hardware cost subject to a performance bound. However, COSYMA starts with a performance-infeasible solution while Vulcan starts with a cost-infeasible initial solution. Vahid et al. [15] designed an objective function that allows them to use binary search:

$$X = k_{\text{perf}} \sum_{1 \leq j \leq m} V_{\text{perf}}(C_j) k_{\text{area}} V_{\text{area}}(W) \quad (34.2)$$

where $V_{\text{perf}}(C_j)$ is the amount by which the j th performance constraint is violated and $V_{\text{area}}(W)$ is the amount by which the hardware cost exceeds the given maximum allowable hardware cost W .

D'Ambrosio and Hu [4] designed objective functions for Pareto optimal search. This form of search uses multiple objectives and groups solutions into equivalence classes—two designs with different combinations of objective functions may be Pareto equivalent. They computed lower and upper bounds on the feasibility of a design, then computed a feasibility factor:

$$\lambda_p = \frac{T_P - T_L}{T_U - T_L} \quad (34.3)$$

where T_L and T_U are the lower and upper bounds on throughput and T_P is the actual throughput required to finish all tasks on time.

Multithreading is an important source of performance improvements in hardware/software codesign. A single-threaded architecture uses blocking communication to coordinate the CPU and accelerator—the CPU waits while the accelerator performs its operations. This style of communication was used by both COSYMA and Vulcan. Single-threaded systems rely on the intrinsic speedup of the accelerator for performance improvements. The accelerator's computation time, including data loading and storage, must be sufficiently faster than CPU execution to make the accelerator worthwhile. A multithreaded system allows the CPU and accelerator to execute simultaneously, sharing results with nonblocking communication. The available parallelism may be limited by the application, but in the best case multithreading provides significant speedups. However, multithreading complicates system performance analysis. Speeding up one thread of execution may not affect the total system execution time if that thread is not on the critical performance path.

34.3 Cosynthesis Algorithms

Kalavade and Lee [11] developed the global criticality/local phase (GCLP) algorithm, which alternates between performance and cost optimization. A global criticality sweep tries to improve the performance on the critical path. A local phase sweep reduces hardware cost by consolidating operations, cost-reducing components, and eliminating unnecessary components. Their algorithm adaptively selects global criticality or local phase operations, depending on the current state of the objective function.

Wolfs algorithm [16] synthesizes a platform architecture and software architecture without an initial template. Cosynthesis in this general case is more difficult because the algorithm cannot rely on invariant properties of the architecture platform. To estimate the cost of the system, we need to allocate tasks to processing elements, but to allocate tasks properly we must have some idea of the relative costs of different allocations. Wolfs algorithm breaks this vicious cycle by starting with a performance-feasible design and

reducing the cost of the platform while maintaining performance feasibility. The initial hardware design uses a separate processing element for each task, with each task implemented on the fastest possible processing element; if this platform is not feasible then the specification is not feasible. The algorithm then reduces system cost by reallocating tasks and eliminating components. Hill-climbing search helps to move all the tasks off a processing element so that it can be eliminated from the platform. Hill-climbing in this case means moving a task to a slower or more expensive element. A task may move several times during optimization so that its original processing element can be removed and a low-cost processing element can be provided for the task.

SpecSyn [7] uses a three-step specify–explore–refine methodology. The design is represented internally as a program state machine model. Design space search explores the possible refinements. Data and control refinements add detail to the implementation of data and control operations. Architectural refinements take care of scheduling, allocation, and communication conflicts. A refinement is chosen, which serves as the starting point for the next specify–explore–refine sequence.

LYCOS [13] breaks the behavioral specification into basic scheduling blocks. It computes the speedup for moving a basic scheduling block from software to hardware. It looks for combinations of basic scheduling blocks that can be arranged to provide the greatest possible speedup within a maximum allowed chip area. They formulate the allocation problem as a dynamic program.

COSYN [2] is designed to synthesize systems with large numbers of tasks. Communication systems are examples of machines that must support a large number of concurrent tasks: each call is represented as a separate task and a machine may handle thousands of simultaneous calls. COSYN takes advantage of the similarity between tasks—a given task type may be instantiated many times to service different calls. It clusters tasks together to take advantage of similarities in behavior; it then allocates and schedules the tasks.

Genetic algorithms have also been used to explore the cosynthesis design space. MOGAC [3] uses genetic algorithms to generate candidate designs and evaluate them based on multiple objective functions. A design is represented by a string that describes its processing elements, the allocation of tasks to processing elements and communication links, the tasks, and other design characteristics. Genetic algorithms perform three types of modifications on these strings to create a new candidate design from an old one: they copy a string, they mutate a string, and they crossover two strings by exchanging parts of the strings. The synthesis algorithm starts with an initial set of strings and generates mutations at each step. The new strings are evaluated using the several objective functions. Some strings are kept for the next step while others are discarded. Random factors are used to select the strings to be kept, so that some poorly ranked strings will be kept and some highly ranked strings will be discarded. MOGAC clusters designs to reduce the search space: each element of a cluster has the same processing element allocation string but different link allocations.

Eles et al. [5] compared simulated annealing and tabu search. They found that both optimization algorithms gave similar results but the tabu search ran about 20 times faster. Their cost metrics were derived from a combination of static and dynamic analysis of a VHDL specification. They used the relative computational load—the number of computations in a block of statements divided by the total number of computations in the specification—as one of the several metrics for optimization. Another metric was the ratio of the number of operations to the length of the critical path, which measures the available parallelism. They also classified operations as being better suited to implementation in hardware or software and used the percentage of hardware- and software-specific operations to help guide allocation decisions.

34.4 CPU Customization

An alternative to hardware/software partitioning is CPU customization. A CPU can be customized in a number of ways:

- Specialized instructions
- New registers
- Custom cache configurations
- Customized memory management

On the one hand, a custom CPU does not provide the multithreading that can be performed on a CPU + accelerator system. On the other hand, custom instructions have easy access to both the standard and the custom registers, thus reducing the overhead of moving data into and out of the accelerator. For some types of operations, customized CPUs may provide a good balance of efficiency and programmability.

A number of systems have been developed to customize CPUs. On the academic side, the most widely used systems are Lisa citehof01 and PEAS/ASIP Meister [12]. On the commercial side, Tensilica, ARC, Toshiba, and other companies provide commercial systems for CPU customization. These systems create a complete register-transfer implementation of the CPU given a variety of inputs from the designer, including architectural features, instruction specifications, and descriptions of how to implement custom instructions. These systems also generate simulators for the processor that can be used for both CPU design verification and system-level simulation.

CPU customization systems allow us to consider the synthesis of custom instructions. Instruction synthesis requires analyzing the application program to determine opportunities for new instructions that will improve performance. Instructions can improve application-level performance in several ways: reducing instruction fetch/execute times, reducing the number of clock cycles required to perform an operation, and reducing the number of registers required.

Holmer and Despain [9] proposed a 1% rule for selecting instructions—any instruction that improved performance by less than 1% should be rejected. They used algorithms similar to those used for microcode compaction to identify combinations of operations that could be combined into a single instruction. Huang and Despain [10] used simulated annealing to search the design space. More recently, Atasu et al. used graph covering algorithms to identify new clusters of operations that create new instructions.

34.5 Codesign and System Design

Methodology is at the heart of embedded system design. Hardware/software codesign affects the methodology used to build embedded computing systems. Whether codesign is performed manually or with tools, it affects the techniques used to design the hardware and software components as well as the design verification methodology.

Figure 34.2 shows the basic design flow for hardware/software cosynthesis-based design. The cosynthesis tool generates two types of results—software and hardware—each with its own design flow for implementation. The results of software and hardware implementation are put together during manufacturing to create the final system.

This figure divides hardware implementation into two phases: architectural synthesis and register-transfer synthesis. Architectural synthesis deals with nonsynthesizable components and assembles the system block diagram. Many of the top-level components will be CPUs, I/O devices, and memories that come as predefined units. However, the synthesis process must still generate a proper netlist to connect these components. Given the large number of variations of a given product family, the architectural synthesis process can be complex. Register-transfer synthesis uses well-known techniques to design any custom logic required. This step may be followed by the physical design of printed circuit boards or integrated circuits.

Hardware/software codesign requires changes to traditional verification and debugging techniques because many of the components of the system have been synthesized. Synthesis must generate components that provide necessary observability and controllability points to allow both tools and humans to be able to understand the operation of the system.

Hardware/software codesign also requires new forms of simulation. Because different parts of the system are implemented in different technologies, we want to be able to run several different types of simulation against each other to simulate the complete system. While we could simulate the software operation by running a hardware simulation of the CPU as it executes the software, this would be extremely slow. Instead, we want to run the software directly as instructions but have it communicate with the simulated hardware. Hardware/software cosimulators [1] are a form of mixed-mode simulation. A simulation

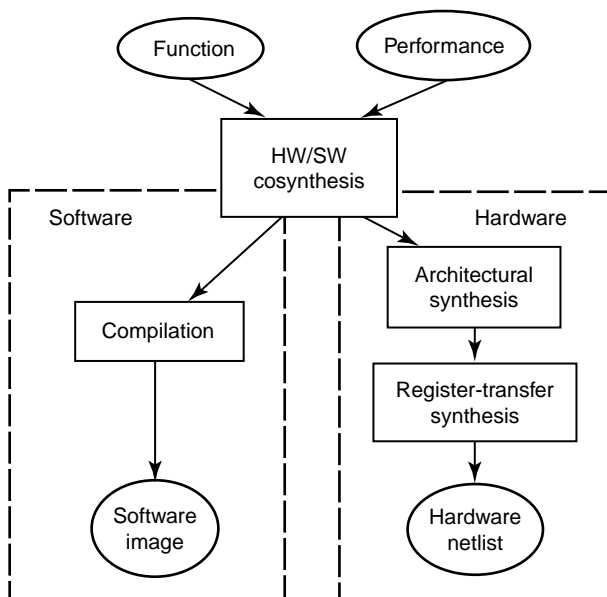


FIGURE 34.2 The hardware/software cosynthesis design flow.

backplane moderates the communication between the hardware and software simulation components. When the hardware and software must communicate, the simulation backplane acts as an interprocess communication mechanism and makes sure that all communication requests are satisfied.

34.6 Summary

Hardware/software codesign has become a major, if not predominant technique for the design of embedded computing systems. A great deal is known about how to design hardware and software together to efficiently meet performance and power goals. Understanding the communication requirements between different parts of the application is key to finding a good architecture for the system. A variety of tools have been developed to search the embedded systems design space. These tools make somewhat different assumptions about both the specification and the underlying components used for implementation; understanding the techniques used by a tool helps designers get the most out of their tools.

References

1. D. Becker, R. K. Singh, and S. G. Tell. An engineering environment for hardware/software co-simulation. In *Proceedings of the 29th Design Automation Conference*, pp. 129–134. IEEE Computer Society Press, 1992.
2. B. P. Dave, G. Lakshminarayana, and N. K. Jha. Cosyn: Hardware-software co-synthesis of heterogeneous distributed embedded systems. *IEEE Transactions on VLSI Systems*, 7(1): 92–104, 1999.
3. R. P. Dick and N. K. Jha. MOGAC: A multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design*, 17(10): 920–935, 1998.
4. J. G. D'Ambrosio and X. S. Hu. Configuration-level hardware/software partitioning for real-time embedded systems. In *Proceedings of the 3rd International Workshop on Hardware-Software Co-Design*, pp. 34–41. IEEE, 1994.

5. P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2: 5–32, 1996.
6. R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design and Test of Computers*, 10(4): 64–75, 1993.
7. D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on VLSI Systems*, 6(1): 84–100, 1998.
8. R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3): 29–41, 1993.
9. B. K. Holmer and A. M. Despain. Viewing instruction set design as an optimization problem. In *Proceedings of the 24th International Symposium on Microarchitecture*, pp. 153–162. ACM Press, 1991.
10. I.-J. Huang and A. M. Despain. Synthesis of application-specific instruction sets. *IEEE Transactions on Computer-Aided Design*, 14(6): 663–675, 1995.
11. A. Kalavade and E. A. Lee. A hardware/software methodology for dsp applications. *IEEE Design & Test of Computers*, 10(3): 16–28, 1993.
12. S. Kobayashi, K. Mita, Y. Takeuchi, and M. Imai. Rapid prototyping of JPEG encoder using the ASIP development system: PEAS-III. In *Proceedings, ASSP 2003*, pp. 485–488. IEEE, 2003.
13. J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. Lycos: The lyngby co-synthesis system. *Design Automation for Embedded Systems*, 2: 195–235, 1997.
14. G. D. Micheli, R. Ernst, and W. Wolf (Eds.). *Readings in Hardware/Software Co-Design*. Morgan Kaufman, 2001.
15. F. Vahid, J. Gong, and D. D. Gajski. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *Proceedings of the Conference on European Design Automation*, pp. 214–219. IEEE Computer Society Press, 1994.
16. W. Wolf. An architectural co-synthesis algorithm for distributed embedded computing systems. *IEEE Transactions on VLSI Systems*, 5(2): 218–229, 1997.

35

Execution Time Analysis for Embedded Real-Time Systems

35.1	Introduction	35-1
	The Need for Timing Analysis • WCET Analysis	
35.2	Software Behavior	35-4
35.3	Hardware Timing	35-5
	Memory Access Times • Long Timing Effects • Caches • Branch Prediction • Timing Anomalies • Multicore and Multiprocessor Systems • Custom Accelerator Hardware	
35.4	Timing by Measurements	35-8
	Measurement Techniques	
35.5	Timing by Static Analysis	35-10
	Flow Analysis—Bounding the Software Behavior • Low-Level Analysis—Bounding the Hardware Timing • Calculation—Deriving the WCET Estimate • Relation between Source and Binary • Analysis Correctness • Supporting a New Target Architecture	
35.6	Hybrid Analysis Techniques	35-15
35.7	Tools for WCET Analysis	35-15
35.8	Industrial Experience with WCET Analysis Tools	35-15
35.9	Summary	35-17

Andreas Ermedahl

Mälardalen University

Jakob Engblom

Virtutech AB

35.1 Introduction

This chapter deals with the problem of how to estimate and analyze the execution time of embedded real-time software, in particular the worst-case execution time (WCET).

A real-time system must react within precise time constraints, related to events in its environment and the system it controls. This means that the correct behavior of a real-time system depends not only on the *result* of the computation but also on the *time* at which the result is produced. Thus, knowing the execution time characteristics of a program is fundamental to the successful design and execution of a real-time system [22,57].

The size and the complexity of the software in the real-time system are constantly increasing. This makes it hard, or even impossible, to perform exhaustive testing of the execution time. Furthermore, the hardware used in real-time systems is steadily becoming more complex, including advanced computer architecture features such as caches, pipelines, branch prediction, and out-of-order execution. These features increase

the speed of execution on average, but also make the timing behavior much harder to predict, since the variation in execution time between fortuitous and worst cases increase.

Execution time analysis is any structured method or tool applied to the problem of obtaining information about the execution time of a program or parts of a program. The fundamental problem that a timing analysis has to deal with is the following: the execution time of a typical program (or other relevant piece of code) is not a fixed constant, but rather varies with different probability of occurrence across a range of times. Variations in the execution time occur due to variations in input data, as well as the characteristics of the software, the processor, and the computer system in which the program is executed.

The WCET of a program is defined as the longest execution time that will ever be observed when the program is run on its target hardware. It is the most critical measure for most real-time work. The WCET is needed for many different types of system analysis for real-time systems. For example, it is a critical component of schedulability analysis techniques, it is used to ensure that interrupts are handled with sufficiently short reaction times, that periodic processing is performed quickly enough, and that operating system (OS) calls return to the user application within a specified time-bound. The simplest case is the question whether a particular small piece of code executes within an allocated time budget.

The *best-case execution time* (BCET) is defined as the shortest time ever observed. The BCET can, for example, be of interest in case some code should not finish too quickly, or to analyze the potential jitter in the execution time. The *average-case execution time* (ACET) lies somewhere in-between the WCET and the BCET, and depends on the execution time distribution of the program.

35.1.1 The Need for Timing Analysis

Reliable timing estimates are important when designing and verifying many types of embedded and real-time systems. This is especially true when the system is used to control safe critical products, such as vehicles, aircraft, military equipment, and industrial plants. Basically, only if each hard real-time component of such a system fulfills its timing requirements the whole system could be shown to meet its timing requirements.

However, whether timing analysis is needed for a program is not a black-and-white question. In reality, there is a continuum of criticality for real-time systems, as shown with some typical examples in Figure 35.1. Depending on the criticality of the system, an approximate or less accurate analysis might be acceptable. It is really a business issue, where the cost of a timing-related failure has to be weighed against the cost of various means of preventing or handling such a failure.

In many embedded systems, only some part of the system is actually time critical. In a mobile telephone, for example, the parts that deal with communication and signal coding have hard real-time requirements, while the user interface is less time-critical. The fact that only a part of the code is timing-critical helps make timing analysis feasible, since trying to analyze the total system would be virtually impossible owing to its size and complexity.

For other type of systems, the goal is to maintain a high throughput on average, and not necessarily that each task is completed within specified time limits. In a telecom system, for example, it is acceptable that calls are dropped occasionally, and that parts of the system crash and reboot. Glitches are annoying but not dangerous as long as the system as a whole keeps running.

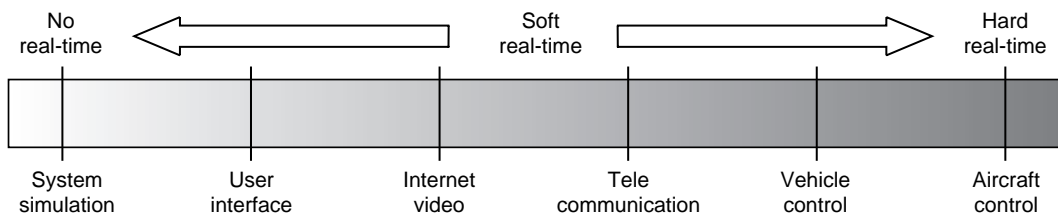


FIGURE 35.1 Continuum of timing criticality.

35.1.2 WCET Analysis

Since the WCET is a key component in the analysis of real-time systems, a lot of research and tool development have been devoted to the WCET determination problem. A timing analysis with a main focus on WCET determination is called a *WCET analysis*, even though most tools also produce other information such as the BCET or maximum stack depth.

Figure 35.2 shows how different timing estimates relate to the WCET and the BCET. The example program has a variable execution time, and the darker curve shows the probability distribution of its execution time. Its minimum and maximum are the *BCET* and the *WCET*, respectively. The lower gray curve shows the set of actually observed and measured execution times, which is a subset of all executions. Its minimum and maximum are the *minimal measured time* and the *maximal measured time*, respectively. In most cases, the program state space and the hardware complexity are too large to exhaustively explore all possible executions of the program. This means that the measured times will in many cases overestimate the BCET and underestimate the WCET.

A WCET analysis derives an *estimate* of the WCET for a program or a part of the program. To guarantee that no deadlines are missed, a WCET estimate must be *safe* (or conservative), i.e., a value greater than or equal to the WCET. To be useful, avoiding over allocation of system resources, the estimate must also be *tight*, i.e., provide little or no overestimation compared to the WCET. Similarly, a BCET estimation should not overestimate the BCET and provide acceptable underestimations. Some real-time literature does not maintain the crucial distinction between the WCET and its estimates derived by timing analysis tools. We will strive to avoid such confusion in this chapter.

It should be noted that all timing analysis techniques presented here are focused on the issue of timing a single program or code fragment. This means that timing analysis does not consider that several tasks or programs normally are run together on the same computer, that there might be an OS which schedules and interrupts the programs, etc. All such interactions are handled on a higher analysis level, e.g., using schedulability analysis, where it should be shown that the whole computer system works even in the most stressful situations. For such analyses, reliable WCET estimates are an important input.

The following two sections will explain in more detail the issues that any timing or WCET analysis must consider. In particular, both the *software behavior* (Section 35.2) and the *hardware timing* (Section 35.3) must be considered to derive reliable timing estimates.

Timing and WCET analysis can be performed in a number of ways using different tools. The two main methodologies employed are *measurements* (Section 35.4) and *static analysis* (Section 35.5). In general, measurements are suitable for less time-critical software, where the average case behavior is of interest. For time-critical software, where the WCET must be known, static analysis or some type of *hybrid method* (Section 35.6) is preferable. The research invested in static WCET analysis has resulted

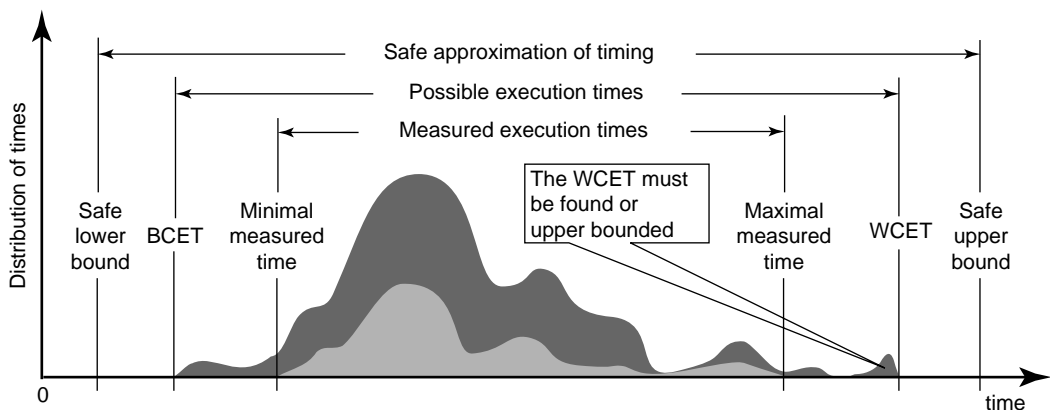


FIGURE 35.2 Example distribution of execution time.

in sophisticated commercial tools (Section 35.7) able to produce estimates within a few percent of the WCET. Experiences from industrial usage of these tools (Section 35.8) and a chapter summary (Section 35.9) complete this chapter.

35.2 Software Behavior

One of the main reasons for the success of computers in embedded applications is that they are *programmable*, making it possible to use a standard part (the processor) for a variety of applications, and enabling new functions that could never have been implemented using fixed-function logic. However, this also means that to correctly analyze and understand the behavior and timing of an embedded system we need to understand the software behavior.

Embedded software comes in many different flavors, using many different languages. In our experience, the most timing-critical real-time software is written in C or Ada, with some assembly language. It is also common to use various code-generating tools working on a high-level model of the software (e.g. MatLab/Simulink, LabView, UML, StateCharts, SDL, and Esterel) to make programming more efficient. This means that a wide variety of codes and coding styles have to be considered for WCET analysis tools.

There have been some studies on the properties of embedded software, and they indicate a huge range of programming practices, from very disciplined simple and predictable code, to code with deep loop nesting, complex if-statements, and significant use of pointers [6,9,49]. Dynamic memory allocation is rare due to resource constraints, but the logic can be very convoluted.

The software behavior contributes a large part of the execution time variability of a program, often dominating the effect of local hardware timing variability [32]. Note that this does not mean that the hardware timing analysis is unnecessary; we need the hardware timing to get a concrete execution time. But we need to have a good understanding of the software behavior to get a precise analysis result.

As illustrated in Figure 35.3, even small codes might exhibit variable and interesting behavior. The example code consists of two functions `task_N` and `convert`. The first function reads two values from two different sensors* and calls `convert` twice, with the values read in each case. The results of the calls to `convert` are used to set an actuator. The `convert` function contains a typical case of software variability, having a loop that iterates a variable number of times depending on input data, and conditional statements where one branch takes longer than the other branch to execute.

A good real-life example of input-dependent flow is the message-polling loop described in Ref. 15, where the number of CAN messages received over a network can vary in number, immediately impacting the execution time of an interrupt handler. On the contrary, some compute-oriented codes exhibit a single possible execution path, by virtue of having no input-dependent loops or decision statements [65]. Instructions in such codes will only exhibit execution time variation owing to hardware effects. This fact has triggered research for rewriting programs to only have single-path behavior [44].

<pre> 1. // The main function 2. void task N(void) { 3. // Read values from sensors 4. int val1 = SENSOR1; 5. int val2 = SENSOR2; 6. // To hold calculated values 7. int res1 = 0; 8. int res2 = 0; 9. // Call twice with different values 10. res1 = convert(val1); 11. res2 = convert(val2); 12. // Set actuator to calculated sum 13. ACTUATOR = res1 + res2; 14. }</pre>	<pre> 15. // Convert read value 16. int convert(int val) { 17. int i = 0; 18. int j = 0; 19. total = 0; 20. while(i <= val) { 21. if(j < 5) 22. j++; 23. if(j > val) break; 24. total = total + j - 2; 25. i++; 26. } 27. return total; 28. }</pre>
---	--

FIGURE 35.3 Code illustrating software variability.

*This is a simplification compared to a real system where the sensor values needs to be calibrated after reading.

Note that complex software behavior is equally problematic for static tools and measurement techniques. A program with complex structure and variable flow will be just as hard to measure correctly as to analyze statically. Real-time code has to be written for predictability and analyzability regardless of the techniques used to analyze their timing.

35.3 Hardware Timing

All timing analysis ultimately has to deal with the timing of the hardware on which a program is executing, and the precise sequence of machine instructions that actually make up a program. As hardware gets more complicated, analyzing the hardware timing behavior becomes progressively harder.

The main complexity in hardware timing analysis is the behavior of the processor itself, along with its memory system. Other components of a computer system, such as IO, networks, sensors, and actuators, have less impact on the program timing. They usually dictate when a program is executed in response to external stimuli, but do not affect the instruction-level execution time to any appreciable extent.

Processor instruction timing has been getting increasingly variable over time, as features improving average-case performance and overall throughput are invented and put in use. Typically, performance improvements are achieved using various speculation and caching techniques, with the effect that the span between best-case and worst-case times increases. The goal is for the common case or average case to be close to the best case, and that this best case is better than the best-case times for previous processors. In the process, the worst case typically gets worse, and the precise timing becomes more dependent on the execution history.

Traditional 8- and 16-bit processors typically feature simple architectures where instructions have fixed execution times, and each instruction has minimal effect on the timing of other instructions. There is little variability in execution time. Somewhat more complex 32-bit processors are designed for cost-sensitive applications. Typical examples are the ARM7, ARM Cortex-M3, and NEC V850E. With simple pipelines and cache structures, variability is present but fairly easy to analyze and limited in scope.

The high end of the embedded market requires processors with higher performance, and these are beginning to deploy most of the variability-inducing features, such as caches, branch prediction, and aggressive pipelining. These range from the ARM11 and MIPS24k designs where the pipeline is still in-order, to full out-of-order superscalar processors, such as the PowerPC 755 and 7448, or even Intel Xeons. Analyzing such processors require quite sophisticated tools and methods in the hardware timing analysis. Nevertheless, there have been impressive success stories for tools analyzing quite complicated hardware systems with good results [20,52].

The mainstream trend in computer architecture is still to add ever-more speculation to improve overall performance. However, it is clear that embedded computer architecture is becoming its own field, and that some designs actually have the issues of real-time systems as their primary design requirements. Such designs emphasize predictability, short interrupt latencies, and bounded worst cases over maximal average-case performance.

35.3.1 Memory Access Times

Even disregarding caches, the time to access main memory on a modern system is highly variable. Modern RAM technology uses a variety of techniques to take advantage of access locality to improve performance, but this also increases variability in execution time. For static analysis, this means that knowing the precise sequence of addresses of memory accesses becomes important. It is also common to mix different memory technologies and thus memory access times in a system, and at the very least the area of memory accessed by an instruction needs to be known for a tight analysis to be performed [15,52].

35.3.2 Long Timing Effects

Conventional wisdom in WCET analysis used to hold that the timing of instructions flowing through a simple pipeline (in-order issue, single instruction per cycle, in-order completion) could be accounted for

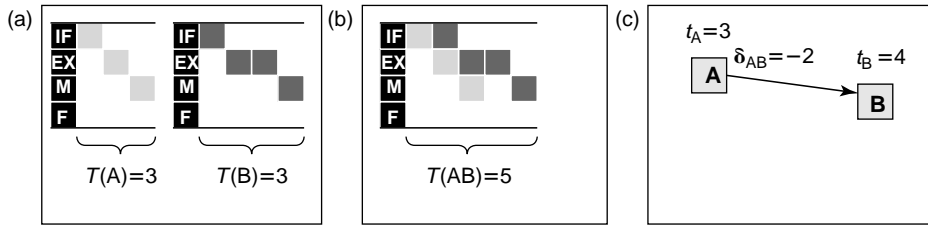


FIGURE 35.4 Pipelining of instruction execution. (a) Separate execution, (b) sequence execution, and (c) modeling the timing using times and timing effects.

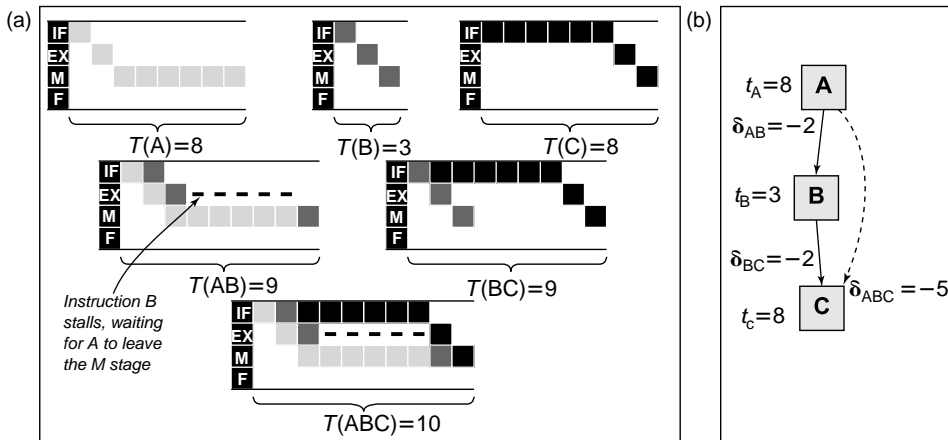


FIGURE 35.5 Example long timing effect. (a) Execution times and (b) timing model.

by pair wise combination of instructions, as illustrated in Figure 35.4. However, this is not true. Even very simple three-stage pipelines exhibit a phenomenon known as *Long Timing Effects* (LTEs). The essence of an LTE is that interactions occur between instructions that are not adjacent to each other, as illustrated in Figure 35.5.

The example in Figure 35.5 shows an interaction between instructions A and C that is not visible when just analyzing the neighboring pairs of instructions AB and BC. It has been shown that such timing effects can occur across arbitrary distances [12,14]. In experiments, some processors have been shown to exhibit many very long LTEs [47]. The practical consequence of this is that a static WCET analysis tool has to be very careful about how interactions between instructions are analyzed, and has to ensure that all possible LTEs are found or a safe margin added to account for them. Note that since WCET tools are allowed to selectively *overestimate* execution times, this problem is tractable. There have also been attempts to make a processor not have any LTEs, thus facilitating analysis [47].

35.3.3 Caches

Caches add to the variability of execution time of a program, and make the difference between the worst case and average case quite large [62]. This effect is well known, and cache optimization is a concern for any performance-oriented programmer. On average, caches work well, but a poorly designed cache or poor use of a cache has the potential to cause disastrous worst cases. From a timing analysis and predictability perspective, caches with least-recently-used replacement policies are preferable, since their state converges over time, making the behavior more predictable [20].

The traditional assumption in analyzing cache behavior is that misses are always worse than cache hits, but as discussed below in Section 35.3.5 this is not necessarily true on all processors.

To overcome some of the unpredictability of caches for critical code, many embedded processors offer the ability to *lock* parts of the cache. This makes it possible to obtain predictable timing for selected code, even in the presence of interrupts and task switching [43,61]. An extreme variant of this is to use a software-controlled cache, where the programmer has complete control and responsibility for what is in the cache and what is not.

35.3.4 Branch Prediction

Dynamic branch prediction mechanisms try to predict which way a particular branch instruction will go (taken or not-taken), long before the branch has actually been resolved in the processor pipeline. The processor will then speculatively fetch instructions along the predicted path, and if the prediction was wrong, the speculatively fetched instructions will have to be *squashed* and instruction fetching redirected. Thus, branch prediction can affect the state of both instruction and data caches, as well as the processor pipeline, and it has a large impact on the variability in instruction execution time.

Effective branch prediction is very important in modern processors with deep pipelines and high clock frequencies. Very sophisticated predictor mechanisms have been developed that on average achieve very high prediction rates. Indeed, to get above 75% accuracy, dynamic prediction is necessary [13].

However, these branch predictors typically have complicated worst-case behavior. There are examples of cases where executing more iterations of an inner loop takes less time than iterating fewer iterations [13], due to branch prediction effects in a loop which is visited several times. Finally, just like the FIFO caches discussed in Section 35.3.5, dynamic branch predictors do not necessarily converge to a known state over time, thus complicating WCET analysis.

35.3.5 Timing Anomalies

When WCET researchers started working on complex, out-of-order processors with caches and branch prediction, a phenomenon known as *timing anomalies* was observed [39]. Timing anomalies are cases where a *local worst case does not entail the globally worst case*. Examples are that a cache hit for a particular instruction causes a longer execution time for a program than a cache miss.

Timing anomalies are usually associated with out-of-order dynamic scheduling of instructions. Most anomalies are similar to the known *scheduling anomalies* studied in scheduling theory, where making a certain task (instruction) faster can cause an entire schedule to take longer. As noted in Ref. 46, there are some cases where the behaviors of processors go beyond scheduling anomalies, since a dynamic decision in a branch predictor can cause the actual set of instructions executed to change.

A strict in-order pipeline does not suffer timing effects in and of itself [12,39]; however, the cache attached to such a pipeline might. For example, a miss in a cache using the FIFO replacement policy might create a better cache state for later code, causing the overall execution to speed up. Since the FIFO cache has a potentially infinite memory, this can cause problems at any later point in time [46].

A variant of a timing anomaly called an *acceleration effect* is that the global slow-down ensuing from a slow-down of some instruction is greater than the local penalty. For example, an instruction being delayed by 8 cycles causing the overall program to be delayed by 11 cycles, as seen in the examples in Ref. 39. Such acceleration effects can be unbounded [39].

35.3.6 Multicore and Multiprocessor Systems

The use of multiple processors and its cores is a clear trend in computer architecture. Depending on how systems are designed and programmed, using multiple processor cores can both benefit and hinder timing analysis.

A system using many specialized processors, each with its own defined task, is easier to analyze than a system combining all the tasks onto a single processor. Less interference between tasks and less competition for shared resources such as caches makes the analysis easier. Private memory for each processor is definitely the recommended design here, as that helps predictability and reduces variability, at the expense of some more work for the programmers. This design template is common in mobile phones, where you typically

find an ARM main processor combined with one or more digital signal processor (DSP)s on a single chip. Outside the mobile phone space, the IBM–Sony–Toshiba Cell processor contains a PowerPC core along with eight DSP-style processors designed with timing predictability in mind [27].

On the contrary, using a classic shared-memory multiprocessor model like those found in PCs and servers makes it significantly harder to analyze the timing of programs. Programs might be interrupted and scheduled on a different processor in mid-execution, and shared data will cause cache misses due to the cache coherency activity on other processors. Even with scheduling tricks, such as binding a particular task to a particular processor, there are just too many complicating interacting factors to make any precise timing analysis impossible.

35.3.7 Custom Accelerator Hardware

Various forms of accelerator hardware is becoming more common in embedded systems, implemented as part of an ASIC, System-on-Chip, or FPGA. This means that real-time software will contain calls to activate the accelerator hardware, and that the WCET analysis will need to account for these calls. This does not have to make the WCET analysis more difficult, as the execution time of accelerator functions is typically fixed or easy to compute. Compared to a software implementation of the same function, hardware accelerators typically exhibit much simpler behavior and less variation. The support necessary in the WCET analysis tool is to be able to identify calls to hardware accelerators, and to provide information about the time the calls take.

35.4 Timing by Measurements

The classic method for obtaining information about the execution time of a program is to execute the program many times with different input data, and then measure the execution time for each test run. Finding the input that causes the WCET to occur is very difficult in the general case, and guaranteeing that it has been found is basically impossible without some form of static analysis. Nevertheless, measurements are often the only means immediately at the disposal of a programmer and are useful when the average case timing behavior of the program is of interest.

On the hardware side, the measurement method has the potential advantage of being performed on the actual hardware. This avoids the need to construct a model of the hardware as required by static analysis techniques (an advantage shared with hybrid analysis techniques as discussed below in Section 35.6). However, measurement requires that the target hardware is available, which might not be the case for systems where hardware is developed in parallel with the software [59].

Note that as hardware gets more complex and execution time variability increases (see Section 35.3), it becomes harder and harder to explore all possible timing with measurements. A static analysis tool has the advantage that it in principle can consider all possible executions and thus the entire possible execution time span.

Measurements can be performed in the lab prior to software deployment, or in the field after deployment. Measuring in the field has the advantage that only real executions are observed, but the clear disadvantage that the data is obtained only after the system has been fielded. If some mistake was made when dimensioning the system, timing-related failures could occur in the field. For systems that can tolerate occasional timing problems and which are continuously upgraded, online measurements of performance can be immensely useful.

35.4.1 Measurement Techniques

Over the years, many software- and hardware-based timing measurement techniques have been developed [56]. We ignore manual methods, such as using a stopwatch, as that is too low in accuracy for real-time code. There are some issues that all measurement methods need to address:

- *Probe effect*: Measuring a system might cause the timing to change. This is especially common when the program is instrumented or extended with measurement support.

- *Resolution*: Executing code can take a very short time, and the resolution of the timing system has to be fine enough to accurately capture the variations that occur. Typically, you want microseconds or better resolution.
- *Interruptions*: The program under measurement might be interrupted by hardware or OS scheduling intervals. This will lead to intermittent large increases in the end-to-end execution time of a program, and this effect needs to be identified and compensated for [67].
- *Visibility*: It is typically hard to deduce the detailed execution path that a program under measure took in a particular measurement run. This leads to problems interpreting the results and attributing execution time appropriately [15,67].

Most of these problems can be resolved by designing a hardware platform that supports debugging and timing measurements from the outset. Today, built-in debug support in embedded processors is improving thanks to Moore's law, allowing more functionality to be put on a single chip. Still, far from all, embedded computer platforms have useful debug and analysis features.

Other issues that need to be considered when selecting a suitable measurement technique are *Cost*, e.g., special purpose hardware solutions, such as an emulator, are more often costly than general purpose ones, such as an oscilloscope. *Availability*, not all type of target systems are using an OS with suitable timing facilities support. *Retargetability*, a solution suitable for a particular processor and a hardware platform might not be directly applicable on another one [56,67].

Some of the more important measurement techniques used in practice today are:

- *Oscilloscopes and logic analyzers*: Both these methods look at the externally visible behavior of a system while it is running, without affecting its execution. Using an oscilloscope (Figure 35.6c) typically involves adding a bit-flip on an externally accessible pin of the processor to the program segments of interest, and then observing the resulting waveform to divine the periodicity and thus the execution time [56,67]. A logic analyzer (Figure 35.6b) looks at the data or address bus of the system and can see when certain instructions are being fetched, greatly improving the visibility. However, it requires that relevant memory transactions do reach the bus, which is not necessarily the case on a system with a cache.
- *Hardware traces*: Hardware traces and debug ports are extra features added on-board processor chips to help debugging and inspection. The most well-known example is the ARM embedded trace macrocell (ETM), and JTAG and Nexus debug interfaces. Note that such interfaces while powerful often have particular limitations that can make their use more complicated than one might think [3].
- *High-resolution timers*: Adding code to a program to get timing information from timers in the system and then collect start and stop times. Sometimes, special clock hardware is added to a system for this purpose. This will slightly modify the program code, under programmer control.
- *Performance counters*: Most high-end processors from families such as x86, PowerPC, and MIPS offer built-in performance counters that can be used to determine details about the performance of programs. However, such mechanisms are typically oriented toward spotting hot-spots and performance problems in a program and less toward reliable and precise execution time measurements.
- *Profilers*: Profilers are typically provided with compilers. They are dependent on good timing sources in hardware to obtain time measurements, but can provide very good insight into where a program



FIGURE 35.6 Tools for dynamic timing analysis. (a) Emulator, (b) logic analyzer, and (c) oscilloscope.

is spending its time. Profilers can use periodic interrupts to sample program execution, which does not provide for very precise measurements, or instrument the code, which creates a probe effect.

- *Operating system facilities:* OS support for timing measurement can take many forms. *High-water marking* is a common feature of real-time OSs, where the longest execution time observed for a particular task is continuously recorded. There can also be command-line tools for timing programs available. Note that all OS-based solutions depend on the availability of suitably precise timing facilities in the hardware.
- *Emulator:* An *in-circuit emulator* (Figure 35.6a) is a special-purpose hardware that behaves like a particular processor but with better debug and inspection capabilities. Provided that they do match the target processor, they can provide very detailed data. Today, emulators are being replaced with hardware trace facilities, since they are too hard to construct for current processors. There is also the risk that they do not actually perfectly match the behavior of the real processor [67].
- *Simulators:* Processor simulators are sometimes used as a substitute for the real hardware for the purpose of timing analysis. Developing and validating correct simulators is very hard [11,12], and in practice there are very few simulators that are guaranteed to be totally accurate.

35.5 Timing by Static Analysis

As mentioned in Section 35.1.2, measurements are suitable for soft real-time applications where the average timing is of interest. However, for hard real-time applications, where the WCET must be known, static analysis techniques are preferable since they provide stronger evidence about the worst possible execution time of a program.

A *static timing analysis tool* works by statically analyzing the properties of the program that affect its timing behavior. It can be compared to determining the stability of a bridge by investigating its design, instead of building the bridge and testing it by running heavy trucks across it. Most static timing analysis has been focused on the WCET determination problem. Given that the inputs and analyses used are all correct, such a *static WCET analysis* will derive a WCET estimate larger than or equal to the actual WCET.

In general, both the software behavior and the hardware timing must somehow be bounded to derive a safe and tight WCET estimate. Consequently, static WCET analysis is usually divided into three (usually independent) phases, as depicted in Figure 35.7: (1) a *flow analysis* phase, where information on the possible execution paths through the program is derived; (2) a *low-level analysis* phase, where information

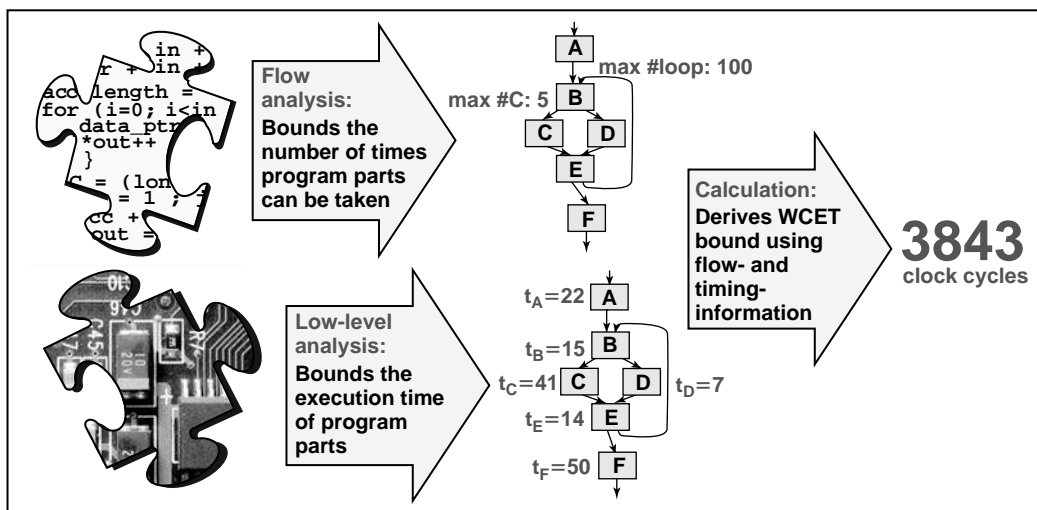


FIGURE 35.7 Phases in static WCET analysis.

about the execution time of program instructions is obtained; and (3) a *calculation* phase, where the flow and the timing information derived in the first two phases are combined to derive a WCET estimate.

Some tools integrate two or more of these phases. For example, the approach in Ref. 39 performs all three functions at the same time, approximately executing the program. Nevertheless, all three phases are needed. As discussed in Section 35.7, most tools also include a *decoding phase*, for reading the program to be analyzed, and a *visualization phase*, for presenting the result of the WCET analysis.

35.5.1 Flow Analysis—Bounding the Software Behavior

The purpose of the *flow analysis* phase is to derive bounds on the possible execution paths of the analyzed program, i.e., to find constraints on the dynamic behavior of the software. Such *flow information* can be provided manually by the system programmer, or by an automatic flow analysis. The result is information on which functions that get called, bounds on loop iterations, dependencies between conditionals, etc.

To find exact flow information is in general undecidable*: thus, any flow analysis must be approximate. To ensure a safe WCET estimate, the flow information must be a safe (over)approximation including (at least) *all* possible program executions.

Upper bounds on the number of loop iterations are needed to derive a WCET estimate at all. Without such *loop bounds*, any instruction occurring in a loop might be taken an infinite number of times, leading to an unbounded WCET estimate. Similarly, recursion depth must also be bounded. For the loop in Figure 35.7, the flow analysis has derived a loop bound of 100. This bound has been added to the control-flow graph as a `max #loop: 100` annotation.

Flow analysis can also identify *infeasible paths*, i.e., paths that are executable according to the control-flow graph structure, but not feasible when considering the semantics of the program and possible input data values. In contrast to loop bounds, infeasible path information is not required to find a WCET estimate, but may tighten the result. An extreme case of an infeasible path is dead code. In Figure 35.7, the infeasible path annotation `max #C: 5` is specifying that node C can be executed at most five times.

There are a number of approaches to automatic loop bound and infeasible path analyses, using techniques such as abstract interpretation, symbolic execution, Presburger analysis, specialized data flow analyses, and syntactical analysis on parse trees [1,5,23,25,26,30,33,38,39,58]. Some of the methods are general, while others are specialized for certain types of code constructs. The methods also differ in the type of codes they analyze, i.e., source, intermediate (inside the compiler), or machine codes. Most WCET analysis tools allow the user to provide additional flow information as manual annotations [17,21,30,34].

Once again, consider Figure 35.3 as an illustration of the work a flow analysis must perform. After a careful examination of the system it has been found that the two sensors, `SENSOR1` and `SENSOR2`, can give readings within certain boundaries, $0 \leq \text{val1} \leq 100$ and $0 \leq \text{val2} \leq 200$.

After studying the code in `convert` we see that its loop will, for each call, iterate as many times as the value of the input argument `val`. A safe upper loop bound for the two calls to `convert` in this program would therefore be to assume that the loop iterates 200 times each time `convert` is called, i.e., a bound corresponding to the largest possible value of `val2`.

However, the loop bound could be improved by noting that the first time that `convert` is called, its input argument cannot be larger than 100. This gives that in total, for the whole execution of the program, the loop in `convert` could iterate at most $100 + 200 = 300$ times. Furthermore, we note that each time `convert` is called, the `j++;` statements on row 22 could be executed at most five times. The last two observations are not required to derive a WCET estimate, but could result in a tighter estimate.

35.5.2 Low-Level Analysis—Bounding the Hardware Timing

The purpose of low-level analysis is to determine and bound the possible timing of instructions (or larger code parts) given the architectural features of the target hardware. For example, in Figure 35.7 each basic block in the control-flow graph has been annotated with an upper execution time derived by a low-level

*A perfect flow analysis would solve the halting problem.

analysis. The analysis requires access to the actual binary code of a program, since it has to analyze the processor timing for each instruction.

For modern processors, it is especially important to analyze the effects of complex features such as pipelines, caches, branch predictors, speculative, and out-of-order execution. As explained in Section 35.3, all these features increase the variability in instruction timing and make the prediction more complex.

Most static low-level analyses work by creating a *timing model* of the processor (and other hardware that affect the instruction timing). The model does not need to include all hardware details, as long as it can provide bounds on the timing of instructions. It is common to use safe approximations where precise timing is impossible or very difficult to provide. Typical examples include assuming that an instruction always misses the cache, or that an instruction whose precise execution time depends on data always executes for the longest possible time. Note that certain intuitive assumptions might not always be correct, as discussed in Section 35.3.5.

One should note that standard cycle-accurate processor simulators differ in functionality from timing models. When determining a timing bound for an instruction not just one single concrete execution of the instruction, but rather all possible executions must be accounted for by the timing model. This can be a very large set of possible states for a complex processor.

The complexity of the low-level analysis and the timing model depends on the complexity of the processor used. For simple 8- and 16-bit processors, the timing model construction is fairly straightforward but still time consuming [42]. For somewhat more advanced 16- and 32-bit processors, using simple (scalar) pipelines and maybe caches, the timing effects of different hardware features can be analyzed separately, making the models fairly efficient and simple [12]. For more advanced processors, the performance enhancing features such as branch prediction, out-of-order execution, and caches typically influence each other, and models that integrate all aspects of the processor timing behavior are needed [20,35,52]. Obviously, such timing models can get very complex, with large state spaces and correspondingly long analysis times.

The low-level analyses in today's WCET analysis tools are usually fully automatic. Only information on the hardware characteristics, such as CPU model, processor frequency, cache and memory layouts, need to be provided as parameters to the analysis. The tools usually allow the user to assist and improve the low-level analysis results, e.g., by specifying what memory area certain memory accesses go to and known bounds on register values [21].

35.5.3 Calculation—Deriving the WCET Estimate

The purpose of the calculation phase is to combine the flow and timing information derived in the preceding phases to derive a WCET estimate and the corresponding worst-case execution path(s). For example, in Figure 35.7 the calculation derives a WCET estimate of 3843 clock cycles. This corresponds to an execution where each of the nodes A, B, E, and F are taken 100 times, while C and D are taken 5 and 95 times, respectively.

There are three main categories of calculation methods proposed in the WCET literature: *tree-based*, *path-based*, and *IPET* (*implicit path enumeration technique*). The most suitable calculation method depends on the underlying program representation as well as the characteristics of the derived flow and timing information.

In a *tree-based calculation*, the WCET estimate is generated by a bottom-up traversal of a tree corresponding to a syntactical parse tree of the program [7,8,37,41]. The syntax-tree is a representation of the program whose nodes describe the structure of the program (e.g., sequences, loops, or conditionals) and whose leaves represent basic blocks. Rules are given for traversing the tree, translating each node in the tree into an equation that expresses the node timing based on the timing of its child nodes.

Figure 35.8a shows an example control-flow graph with timing on the nodes and loop bound flow information. Figure 35.8d illustrates how a tree-based calculation method would proceed over the graph according to the program syntax-tree and given transformation rules. Collections of nodes are collapsed into single nodes, simultaneously deriving a timing for the new node.

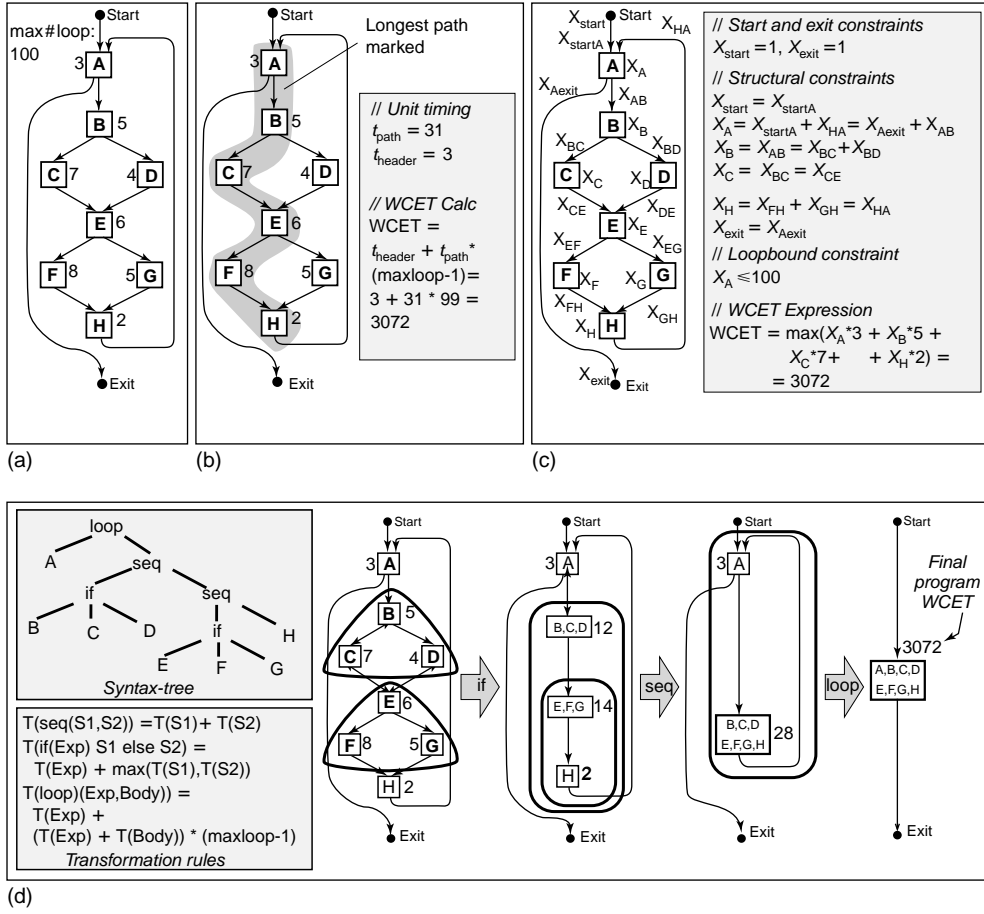


FIGURE 35.8 Different calculation methods. (a) CFG with timing and flow information, (b) path-based calculation, (c) IPET calculation, and (d) tree-based calculation.

In a *path-based calculation*, the WCET estimate is generated by calculating times for different paths in parts of a program, to form an overall path with the worst execution time [24,53,54]. The defining feature is that possible execution paths are *explicitly* represented.

Figure 35.8b illustrates how a path-based calculation method would proceed over the graph in Figure 35.8a. The loop in the graph is first identified and the longest path within the loop is found. The time for the longest path is combined with the loop bound flow information to extract a WCET estimate for the whole program.

A frequently used calculation method is *IPET* [17,28,36,45,58]. Here, the longest path no longer is explicit, but instead *implicitly* defined. IPET represents the program flow and execution times using algebraic and logical constraints. Each basic block or edge in the basic block graph is given a time (t_{entity}) and a count variable (x_{entity}), the latter denoting the number of times that block or edge is executed. The WCET is found by maximizing the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. The WCET estimate is then derived using integer linear programming (ILP) or by constraint solving. The result is a worst-case count for each node and edge, and not an explicit path like in path-based calculation. IPET is usually applied on a whole program basis, but can also be applied on smaller program parts in a bottom-up fashion [16,28].

Figure 35.8c shows the constraints and WCET formula generated by an IPET-based calculation method for the program illustrated in Figure 35.8a. The *start* and *exit constraints* states that the program must be

started and exited once. The *structural constraints* reflect the fundamentals of program flow, where a basic block has to be entered the same number of times as it is exited. The *loop bound* is specified as a constraint on the number of times node A can be executed. Additional constraints on the execution counts of nodes can be used to tighten the WCET bound.

35.5.4 Relation between Source and Binary

Static WCET analysis has to be performed on the compiled object code of a program, since that is the only level that actually reveals the actual sequence of machine instructions being executed. However, performing flow analysis on the object-code level is harder than on source code, since the identity of variables and the program structure is obscured by the process of compilation.

Ideally, flow analysis should be performed on the source code level, where program flow is easier to divine than on the compiled code. Analyzing the source code also makes it simpler for the programmer to help the analysis with annotations. Combining source-code level flow analysis with object-code level timing analysis requires flow information to be *mapped* from source code to object code.

This mapping process is complicated by the fact that the compiler performs many transformations on the code during code generation. For example, transformations such as unrolling loops, inlining function calls, or removing unused code all make the relationship between source code and compiled code non obvious [10,34].

The pragmatic solution today is to use only low levels of optimization when compiling programs to be analyzed, relying on debug information in compiler provided symbol tables, and using manual intervention to resolve tricky cases [30,58]. Some research tools are attempting to integrate the WCET analysis with the compiler [19,23,24], which gets around the problem and should allow for a more precise analysis. However, it binds the analysis tightly to a particular compiler.

Note that tools operating on object code are to some extent dependent on the compilers used, since they need to understand the peculiarities of their code generation to build a model of the code structure in the decoding phase, as discussed in Section 35.7.

35.5.5 Analysis Correctness

To guarantee that a static WCET analysis tool derives safe WCET estimates, all parts of a tool must be proven safe. For the low-level analysis, the correctness of the timing model is crucial. Validation of a timing model is a complex and burdensome work, especially for complex CPUs [12,40]. Usually, a large number of test cases are used to compare the model with the actual target hardware, forcing the model to follow the same path as the execution on the hardware.

The correctness of flow analysis methods are much easier to prove, since the semantics of programming languages are well defined and the analysis methods used are often well studied in the compiler world. Even for machine code, the semantics of instructions are relatively simple and well defined, but unfortunately this does not hold for the timing behavior. The calculation part of a WCET tool is usually small and simple, and thus easy to validate, while functions such as visualization of results (see Section 35.7) are non critical and do not influence the correctness of the tool as such.

35.5.6 Supporting a New Target Architecture

Compared to other types of programming tools, such as compilers and functional simulators, a WCET analysis tool requires much more work to support a new processor, something which is reflected in the number of processors supported by different tools and the price of the tools.

The main bottleneck is the creation and validation of the CPU timing model [42]. This has led to the development of hybrid WCET analysis methods (see Section 35.6), where measurements are used instead of a static hardware model. Other approaches intended to simplify the porting of a WCET tools is to reuse existing cycle-accurate simulation models [2,12,66] or derive models from VHDL code [60].

35.6 Hybrid Analysis Techniques

Hybrid analysis techniques use measurements and static analyses in *complement* to each other. The basic idea is the following: first, a model of the program is constructed by static analysis of the code. The model is then annotated with measurement points at suitable places, partitioning the program into a number of smaller parts for measurement. The program, or smaller parts of the program, is then executed a number of times and time measurements are performed. For each measured program part, a worst measured execution time is noted. The measured times are then brought back into the static model and used to deduce a WCET estimate [2,63,66].

Similar to static WCET analysis, hybrid method requires that all loops are bounded. These loop bounds can either be derived using measurements (by keeping track on the largest number of iterations for a loop observed), through static analysis or by manual annotation.

The main advantage of the hybrid method is that no timing model of the processor needs to be created. In a sense, the low-level analysis is replaced with structured measurements. Furthermore, traditional test coverage criteria can be used to show that the program has been thoroughly tested.

The main drawback to the method is that there are no guarantees that the obtained WCET estimates are safe. Timing for program parts are derived using measurements on a subset of all program executions. Thus, it is hard to guarantee that each program part has encountered its worst case timing. Moreover, it is possible to add worst case timings for program parts that can never occur together, potentially ruining the precision.

The user or the tool also has to somehow provide suitable input data to ensure that all program parts are executed. The method also requires that there exists suitable measuring mechanisms to obtain the timing for program parts, see Section 35.4.1.

35.7 Tools for WCET Analysis

During the last couple of years a number of WCET analysis tools have been developed, both research prototypes and commercial tools. The latter are all research projects that have evolved into commercial products. Commercial tools include *aiT* (www.absint.com) and *Bound-T* (www.tidorum.fi), both static WCET analysis tools, and *Rapitime* (www.rapitasystems.com), which is a hybrid analysis tool. For a more detailed presentation of commercial and research tools, we refer to Ref. 64.

The tools differ in the processors supported and the types of codes they can analyze. In general, all tools contain the three analysis phase described in Section 35.5 (see Figure 35.9). Normally, there is also a *decoding* phase, where the executable program is read and converted into an internal representation for the analysis. The decoding phase can be quite complicated depending on the instruction set and the compiler used to generate the code (see Section 35.5.4). Most tools also contain some form of *visualization* of the results, e.g., a program flow graph where different program parts have been annotated with the number of times they are taken in the worst-case execution.

Most tools allow the user to improve the results and guide the analyses by various forms of *manual annotations*. Certain annotations are mandatory, e.g., the specific hardware configuration used, the start address of the code that should be analyzed, possible values of input data, and bounds for loops that cannot be determined by the flow analysis. Some annotations are optional, e.g. information on infeasible paths or the memory area certain memory accesses go to, but may give a tighter WCET estimate.

Some tools also provide other information that can be obtained with a small extra effort, based on the analysis already made while doing the WCET. A typical example is the maximum stack depth a program can use, which is useful for configuring certain OSs.

35.8 Industrial Experience with WCET Analysis Tools

As mentioned above, there are several commercially available WCET analysis tools today. The tools have been used to analyze software timing in avionics, space, and car industries, e.g., *aiT* has been used during

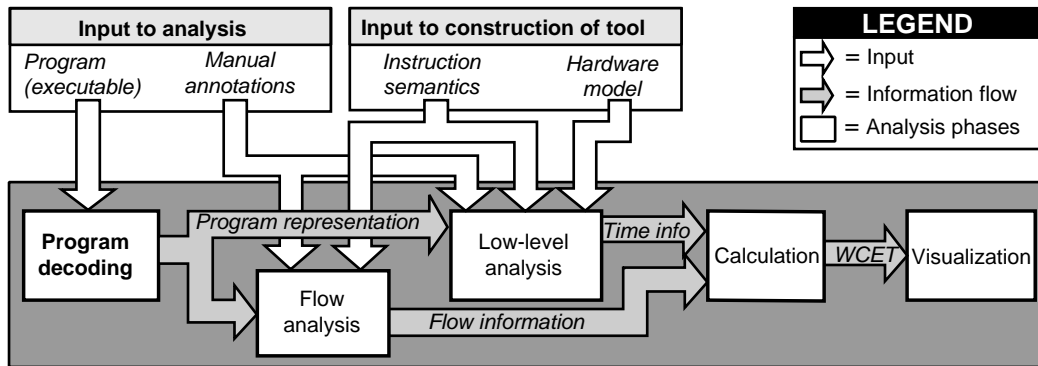


FIGURE 35.9 WCET analysis tool layout.

the development of the software for the fly-by-wire system in Airbus A380 [64]. There are published reports about industrial [4,15,18,28,29,50–52,59] and educational use [42] of these tools. The following list summarizes some of the most important lessons reported:

- Fundamentally, it is possible to apply today's WCET analysis tools to a variety of industrial systems and codes. The tools used derived safe WCET estimates in comparison to measured times and received positive feedback regarding analysis time, precision, and usefulness.
- WCET analysis is not yet a fully automated "one-click" analysis. Manual user interaction, typically annotations to describe the program flow and bounds on input data, is required to obtain useful results. This, in turn, requires a detailed understanding of the analyzed code and its operating environment [4,15,18,51].
- A higher degree of automation and support from the tools, e.g., automatic loop bound calculation, would in most cases have been desirable. The tools used today are only able to handle a subset of the loops encountered [4,29,48,50,52].
- The structure of the code of a system has significant impact on the analyzability of a system. Many small well-defined tasks scheduled by a strict priority Real-time OS or a time-triggered schedule is easier to analyze than monolithic interrupt-driven programs based on an infinite main loop [15,51,52].
- Manual annotations can sometimes be burdensome to generate, even when the code is well structured and easy to understand, owing to the volume of annotations needed. For example, if a C macro construct requires an annotation to be analyzed, that macro requires an annotation for each use [52].
- Once a static analysis tool is setup, reanalyzing a system after changes is typically faster than measuring it. This could potentially save large amounts of time in development [29].
- Code should be written to facilitate timing analysis, by adhering to strict coding rules. Typically, you want all loops to be statically bounded, input-data dependence minimized, pointer use minimized, and code structure as simple and straightforward as possible. Such coding style is commonly recommended for safety critical and real-time systems programming [22,31,55].
- Presenting the results in a graphical user interface is very valuable for obtaining an overview of the analyzed code and to interpret and understand the result of the analysis [4,15,18].
- A single WCET bound, covering all possible scenarios, is not always what you want. Sometimes it is more interesting to have different WCET bounds for different execution modes or system configurations rather than a single WCET bound. The latter would usually be an overapproximation. In some cases, it was possible to manually derive a parametrical formula [4,15,50] showing how the WCET estimate depends on some specific system parameters.
- Automatically generated code can be very amenable to the WCET analysis, provided the code generator is designed with predictability and analyzability in mind [52]. Generated code can also be

hard to analyze, if predictability and analyzability were not designed into the code generator since code generators can generate very convoluted and complex code that is hard to understand for both human programmers and automatic program analyses.

- Not only application-level code is interesting to analyze, but also other code such as OS services and interrupt handlers [6,15,50].
- The large diversity in processors used in embedded systems hinders the deployment of static WCET analysis. A tool often has to be ported to the particular processor(s) used in a project, which costs both time and money [4,42] see also Section 35.5.6.
- Measurements and static timing analysis should not be seen as isolated tasks. Instead, they could complement each other, together giving a better understanding of the system timing and increase the trust in the resulting timing estimates [15,67].

35.9 Summary

This chapter has dealt with the problem of how to estimate and analyze the execution time of programs. Knowing the execution time characteristics of a program is fundamental to the successful design and execution of a real-time system. For hard real-time systems, reliable WCET estimates are especially important.

A variety of methods are available to performing timing and WCET analysis, ranging from manual measurements to automated static analyses. The method(s) suitable for a particular application depends in high degree on its real-time criticality and the availability of tools suited to the particular system.

For the static WCET estimation problem, a lot of research investment has been made, and today several commercial WCET analysis tools are available. Industrial usage of these and other WCET analysis tools show very promising results in published case studies.

References

1. H. Aljifri, A. Pons, and M. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *Proc. 19th IEEE International Performance, Computing, and Communications Conference (IPCCC2000)*. IEEE, February 2000.
2. G. Bernat, A. Colin, and S. Petters. pWCET: A tool for probabilistic worst-case execution time analysis. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'03)*, 2003.
3. A. Betts and G. Bernat. Issues using the nexus interface for measurement-based wcet analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, July 2005.
4. S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17th Euromicro Conference of Real-Time Systems (ECRTS'05)*, pp. 249–258, July 2005.
5. T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, pp. 40–43, July 2005.
6. A. Colin and I. Puaud. Worst-case execution time analysis for the RTEMS real-time operating system. In *Proc. 13th Euromicro Conference of Real-Time Systems (ECRTS'01)*, June 2001.
7. A. Colin and G. Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems (ECRTS'02)*, pp. 50–59, 2002.
8. A. Colin and I. Puaud. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, 18(2/3): 249–274, 2000.
9. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

10. J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10th Euromicro Workshop of Real-Time Systems*, pp. 146–153, June 1998.
11. J. Engblom. On hardware and hardware models for embedded real-time systems. In *Proc. IEEE Real-Time Embedded Systems Workshop, Held in Conjunction with RTSS2001*, December 2001.
12. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.
13. J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Proc. 8th IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.
14. J. Engblom and B. Jonsson. Processor pipelines and their properties for static WCET analysis. In *Proc. 2nd International Workshop on Embedded Systems (EMSOFT2001)*, 2002.
15. O. Eriksson. *Evaluation of Static Time Analysis for CC Systems*. Master's thesis, Mälardalen University, Västerås, Sweden, August 2005.
16. A. Ermedahl, F. Stappert, and J. Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9): 1104–1122, 2005.
17. A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.
18. A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, pp. 19–22, July 2005.
19. H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-aware C compiler. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)*, July 2006.
20. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. 1st International Workshop on Embedded Systems (EMSOFT2000)*, LNCS 2211, October 2001.
21. C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, 2003.
22. J. Ganssle. Really real-time systems. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, April 2006.
23. J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, February 2005.
24. C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1): 53–70, 1999.
25. C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2–3): 129–156, 2000.
26. C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999.
27. P. Hofstee. Power efficient processor architecture and the cell processor. In *11th Symposium on High-Performance Computing Architecture (HPCA-11)*, February 2005.
28. N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.
29. N. Holsti, T. Långbacka, and S. Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, September 2000.
30. N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002)*, 2002.

31. G. Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6): 95–99, June 2006.
32. C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proc. 28th International Symposium on Computer Architecture (ISCA 2001)*.
33. D. Kebbal and P. Sainrat. Combining symbolic execution and path enumeration in worst-case execution time analysis. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)*, July 2006.
34. R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.
35. X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Proc. 25th IEEE Real-Time Systems Symposium (RTSS'04)*, pp. 92–103, December 2004.
36. Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd Design Automation Conference*, pp. 456–461, 1995.
37. S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7): 593–604, 1995.
38. Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pp. 31–40, June 1998.
39. T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.
40. S. Montán. *Validation of Cycle-Accurate CPU Simulator against Actual Hardware*. Master's thesis, Dept. of Information Technology, Uppsala University, 2000. Technical Report 2001–2007.
41. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS'90)*, pp. 72–81, December 1990.
42. S. Petersson, A. Ermedahl, A. Pettersson, D. Sundmark, and N. Holsti. Using a WCET analysis tool in real-time systems education. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, Palma de Mallorca, Spain, July 2005.
43. I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.
44. P. Puschner. The single-path approach towards WCET-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pp. 699–704, December 2003.
45. P. Puschner and A. Schedl. Computing maximum task execution times—A graph-based approach. *Journal of Real-Time Systems*, 13(1): 67–91, 1997.
46. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)*, July 2006.
47. C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proc. 2nd conference on Computing frontiers*, May 2005.
48. M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in calculating the WCET of a complex on-board satellite application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, 2003.
49. C. Sandberg. Inspection of industrial code for syntactical loop analysis. In *Proc. 4th International Workshop on Worst-Case Execution Time Analysis (WCET'2004)*, June 2004.
50. D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, October 2004.
51. D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.

52. J. Souyris, E. L. Pavec, G. Himbert, V. Jegu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, July 2005.
53. F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4): 339–355, 2000.
54. F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, November 2001.
55. D. B. Stewart. Twenty-five most common mistakes with real-time software development. In *Proc. of the Embedded Systems Conference, San Francisco 2001 (ESCSF 2001)*, April 2001.
56. D. B. Stewart. Measuring execution time and real-time performance. In *Proc. of the Embedded Systems Conference, San Francisco 2004 (ESCSF 2004)*, March 2004.
57. D. B. Stewart. Twenty-five most common mistakes with real-time software development. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, April 2006.
58. S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
59. S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003)*, June 2003.
60. S. Thesing. Modeling a system controller for timing analysis. In *Proc. 6th International Conference on Embedded Software (EMSOFT2006)*, October 2006.
61. X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proc. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pp. 272–282, San Diego, CA, June 2003.
62. L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'05)*, March 2005.
63. I. Wenzel, B. Rieder, R. Kirner, and P. P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'05)*, Volume 1, pp. 606–611, 2005.
64. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—Overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems (TECS)*, 2007 (Accepted for publication).
65. F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3–4): 339–356, 2001.
66. F. Wolf, J. Kruse, and R. Ernst. Timing and power measurement in static software analysis. *Microelectronics Journal, Special Issue on Design, Modeling and Simulation in Microelectronics and MEMS*, 6: 91–100, 2002.
67. Y. Zhang. *Evaluation of Methods for Dynamic Time Analysis for CC Systems AB*. Master's thesis, Mälardalen University, Västerås, Sweden, August 2005.

36

Dynamic QoS Management in Distributed Real-Time Embedded Systems

36.1	Introduction	36-1
36.2	Issues in Providing QoS Management in DRE Systems.....	36-2
	The Need for End-to-End QoS Management • The Need for Multilayer QoS Management • The Need for Aggregate QoS Management • The Need for Dynamic QoS Management	
36.3	Solutions for Providing QoS Management in DRE Systems	36-5
	Middleware for Dynamic QoS Management • Software Engineering of QoS Management	
36.4	Case Studies of Providing QoS Management	36-12
	HiPer-D, Reactive QoS Management in a US Navy Testbed • WSOA, Hybrid Control and Reactive QoS Management in a Dynamic Mission Replanning US Air Force Flight Demonstration • PCES, End-to-End and Multilayered Controlled QoS Management in a Multiplatform Live-Flight Demonstration	
36.5	Conclusions	36-30

Joseph P. Loyall
BBN Technologies

Richard E. Schantz
BBN Technologies

36.1 Introduction

Increasingly, embedded systems are part of larger distributed real-time embedded (DRE) systems in a wide variety of domains, including military command and control (C2), avionics and air traffic control, and medicine and emergency response. DRE systems combine the stringent quality of service (QoS) requirements of traditional closed embedded systems with the challenges of the dynamic conditions associated with being widely distributed across an often volatile network environment. Traditionally, embedded systems have been able to rely on their closed environments and self-contained bus architectures to limit the dynamic inputs possible and could rely on static resource management techniques to provide the QoS and reliable performance they need. The environment of distributed, networked systems is more open with heterogeneous platforms, where inputs can come from external devices and platforms, and dynamic, in which conditions, resource availability, and interactions can change. Because of this, achieving the necessary predictable real-time behavior in these DRE systems relies more on the ability to

manage resources end-to-end, map system-level requirements to platform-level controls, aggregate and manage conflicting requirements, and adapt and reconfigure to changing conditions.

Middleware, such as the common object request broker architecture (CORBA), is being applied to these types of applications because it provides a high-level platform of important services for interconnected systems and because of its ability to abstract issues of distribution, heterogeneity, and programming language from the design of systems. CORBA, specifically, has spearheaded this trend because of its development of standards supporting the needs of DRE systems, such as real-time CORBA (RT-CORBA) [27], fault-tolerance CORBA (FT-CORBA) [25], and Minimum CORBA [26].

This chapter describes research efforts to develop middleware for providing dynamic, adaptive QoS management in DRE systems. We describe issues in providing a middleware platform for QoS adaptive systems, middleware-based solutions we have developed as part of our research, and case studies illustrating and evaluating their application to the DRE system context.

36.2 Issues in Providing QoS Management in DRE Systems

In DRE applications, quality of the service provided is as important as functionality, that is, how well an application performs its function is as important as what it does. Many DRE applications control physical, chemical, biological, or defense processes and devices in real time, so that degraded performance or reliability could have catastrophic consequences.

As these traditionally closed embedded systems have become networked, they have formed DRE systems that consist of

- Multiple competing end-to-end streams of information and processing
- Changing numbers and types of participants, with changing roles and relative importance in the system's overall mission or goal
- Heterogeneous, shared, and constrained resources

QoS management is a key element of the design and runtime behavior of DRE systems, but it is often defined in terms of management of individual resources, for example, the admission control provided by network management or CPU scheduling mechanisms or services. While individual resource management is necessary, it is not sufficient in DRE systems because

- Effective QoS management spans and integrates individual resources. The consumer of information determines the QoS requirements, which might change over time, while the information source (frequently remote from the consumer and therefore using different resources) and transport determine significant aspects of the quality and form of information as consumed.
- Management of individual resources is typically applied as a local view, while effective resource management must take into account the overall mission requirements, the relative importance of various resource users to a mission, and what constitutes effective resource usage to achieve mission goals.
- Effective QoS management must mediate the contention for resources from many simultaneously operating applications, streams of information, and systems.
- There might be multiple, simultaneous bottlenecks (i.e., the most constrained resources) and the bottlenecks might change over time.

QoS management for DRE systems must therefore derive and relate the QoS requirements to the mission requirements, simultaneously manage all the resources that could be bottlenecks, mediate varying and conflicting demands for resources, efficiently utilize allocated resources, and dynamically reconfigure and reallocate as conditions change.

Based on the above bulleted items, we break the discussion of issues in providing QoS management in DRE systems into the following four major subsections, each of which we will discuss in turn

- End-to-end QoS management—The management of QoS for an individual end-to-end information stream, from information sources to information consumers. That is, managing the resources

associated with information collection, processing, and delivery to satisfy a particular use of information.

- **Multilayer QoS management**—The management of QoS for a mission or set of high-level operational goals, which includes the mapping of high-level, system-wide concepts into policies driving QoS at the lower levels, followed by enforcement at the lowest level.
- **Aggregate QoS management**—The mediation of demands and negotiation for resources between multiple end-to-end streams that are competing for resources.
- **Dynamic QoS management**—Adapting to changes in resource availability, mission and application needs, and environmental conditions (e.g., scale, number of elements under QoS management, failure and damage management, and cyber attacks) to maintain, improve, or gracefully degrade delivered QoS.

36.2.1 The Need for End-to-End QoS Management

QoS is often defined in terms of management of individual resources, for example, the admission control provided by network management or CPU scheduling mechanisms or services. While managing the resources at a bottleneck might be sufficient at any given time, it is important to understand the limitations of managing the resources only at a specific point. Eliminating a bottleneck by providing additional resources might simply expose a different bottleneck elsewhere that must also be managed. For example, allocating more bandwidth to a specific bottleneck (e.g., using bandwidth reservation, differentiated services, alternate paths, or reducing the other load on the network) might simply expose that there is not enough available CPU resource at a node along the end-to-end path to process the now plentiful data.

Furthermore, shaping application or data usage to fit a specific bottleneck can also have consequences that change, but do not eliminate, the bottleneck. For example, an application facing a constrained network can use data compression to consume less bandwidth. However, in doing so the application is consuming more CPU, which might or might not be available. There might be multiple, simultaneous bottlenecks, the bottlenecks might change over time, and so effective QoS management needs to span individual resources.

The end-user of information and the purpose for which it is used determine the QoS requirements for the information, but the information source and transport determine what quality can be provided. Therefore, QoS management for DRE systems must be end-to-end, that is, it involves capturing the QoS requirements of the information user; managing the quality of the information production, processing, and delivery; managing the resources at all points from end-to-end that could be bottlenecks; and shaping application or data usage to effectively and efficiently use the managed resources.

36.2.2 The Need for Multilayer QoS Management

Effective QoS management requires resolving multiple views of QoS and resource management, as illustrated in Figure 36.1, ranging from higher-layer views closer to total system or mission goals and expressed in higher-level terms down to lowest-layer views closer to individual resources and platforms, and their controls. Although there can be many layers, we identify three in particular:

- *Mission/system layer view.* At the system layer, there is the knowledge of the mission goals, the applications in the system, and the available resources. This is also the layer at which there is an understanding of the relative importance of applications to mission goals, resource allocation strategies for each goal, and the policies for mediating conflicting application resource needs.
- *Application/application string layer view.* An application view of resource management involves acquiring whatever resources are needed to meet a single end-to-end application string's (i.e., a distributed application from the information source to information consumer) requirements and to effectively utilize the resources available to them. If there are not enough resources available, then an application needs to be flexible enough to adjust its resource needs (with corresponding graceful degradation of its functionality) or it will fail, for example, throw an exception. Applications greedily acquiring all the resources they need do not scale in dynamic and resource constrained

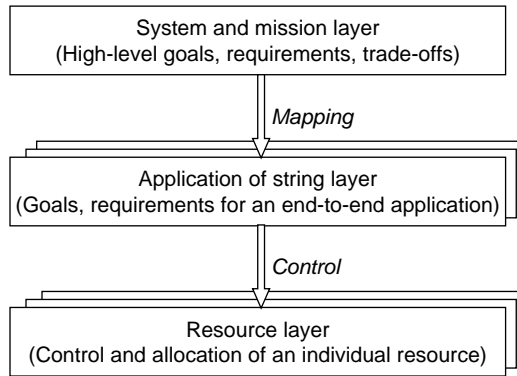


FIGURE 36.1 Multilayered view of QoS management.

environments, but applications cooperating to share resources effectively require sophisticated coordination and control.

- *Resource layer view.* Resource-specific mechanisms control access to resources, deciding whether and how a request for a resource allocation should be granted. Typical CPU and network resource allocation is based on priorities or reservations. Typical resource allocation mechanisms have little or no knowledge of the applications and missions using them or their requirements, although some limited information can be propagated to the resource level in the form of relative priorities and reservation requests.

Trying to manage QoS at any one of these layers is insufficient. Managing QoS only at the system layer results in specification without enforcement; only at the application layer results in limited context and narrow, easily subverted enforcement; and only at the resource layer results in control without context and localized decisions without understanding their global consequences. It is necessary to combine the system, application and resource views to effectively manage and provision QoS. The system layer determines effective allocations; the application layer chooses how to effectively combine and use the allocations; and resource mechanisms manage access to shared and constrained resources.

36.2.3 The Need for Aggregate QoS Management

In a distributed system, resources are frequently shared among end-to-end application strings. The sharing might be as simple as a network backbone through which all traffic from end-to-end travels, and can also include computation resources (at either end or in the middle), and less obvious and sometimes abstract contended resources such as access to specific data, screen real estate or attention from a human operator. Free-for-all unmanaged environments, such as the Internet, purposefully provide no QoS guarantees, in the form of best effort for all. In contrast, to get the predictable performance and control that DRE systems need, aggregate QoS management must be provided, that is, QoS management must mediate the conflicting demands for resources, enforce starvation and fairness policies, and place constraints on individual applications' usage of shared resources.

Application layer QoS control without aggregate QoS management leads to a situation called the tragedy of the commons [10]. In this phenomenon, there is no incentive for applications to share resources and, in fact, there is a disincentive to do so. Each application grabs as many resources as it can and the entire system degrades to best effort. Aggregate QoS management is necessary to control this and ensure controllable QoS across and among applications. Approaches to aggregate QoS management include

- *Negotiation*—Applications negotiate for access to resources and then use only those that they have been assigned through agreement with other applications. An example is a commerce or free market approach, in which applications get a certain amount of currency to spend on resource allocations and QoS guarantees.

- Hierarchical—As described in the multilayer section above, a management entity with a higher-level view determines which applications are relatively more important, or more in need of resources, than others. The higher-layer manager imposes policies and constraints to which the applications must adhere.
- Static—The traditional approach in embedded systems uses offline analysis of the system to determine an allocation of resources across the applications. This approach has limitations in DRE systems, as described in the next section.

36.2.4 The Need for Dynamic QoS Management

The move from self-enclosed embedded systems to distributed embedded systems is also one of moving from static to dynamic environments. DRE systems are frequently parts of larger systems of systems, with participants that come and go, deployment in hostile environments, changing resource availability and contention, failures and cyberattacks, as well as other dynamic conditions. Static resource allocation might be appropriate for a situation at a single point in time, for example, initial deployment, but quickly become insufficient as conditions change.

Designing and implementing QoS management into a system therefore requires building in the control and flexibility to manage changes in resource availability and mission requirements. In other words, effective QoS management relies on recognizing where bottlenecks exist at any given point in time and effectively managing and reallocating resources to remove the bottlenecks; dynamically adapting and reconfiguring application functionality to compensate for the bottlenecks and meet system requirements; or combinations of both.

36.3 Solutions for Providing QoS Management in DRE Systems

36.3.1 Middleware for Dynamic QoS Management

Although it is possible to provide end-to-end QoS management by embedding QoS control and monitoring statements throughout a software system, such an approach leads to additional code complexity, reduced maintainability, and nonreusable software. A better approach is to separate the QoS concerns from the functional concerns of an application and combine the two into a QoS-managed software system through integration at a middleware layer.

An approach that we have taken to do this is providing extensions to standards-based middleware that allow aspects of dynamic QoS management to be programmed separately and then integrated into distributed object or component-based systems.

36.3.1.1 Quality Objects

Quality objects (QuO) is a distributed object framework that supports the separate programming of (1) QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for controlling and providing QoS and for adapting to QoS variations that occur at runtime. By providing these features, QuO separates the role of functional application development from the role of developing the QoS behavior of the system.

As shown in Figure 36.2, a QuO application inserts additional steps in the remote method invocation of distributed object applications [18]. The QuO runtime monitors the state of QoS before each remote invocation through the use of system condition objects that provide a standard interface to observable and controllable parameters in a platform, such as CPU utilization or bandwidth usage. Delegates intercept remote calls and a contract decides the appropriate behavior to apply. The contract defines the set of possible states of QoS in the system using predicates of system condition object values. Based upon the current QoS state, the contract could (1) allow the method call to proceed as is; (2) specify additional processing to perform; or (3) redirect the invocation to a different method; or (4) invoke a callback on the application to alter its execution.

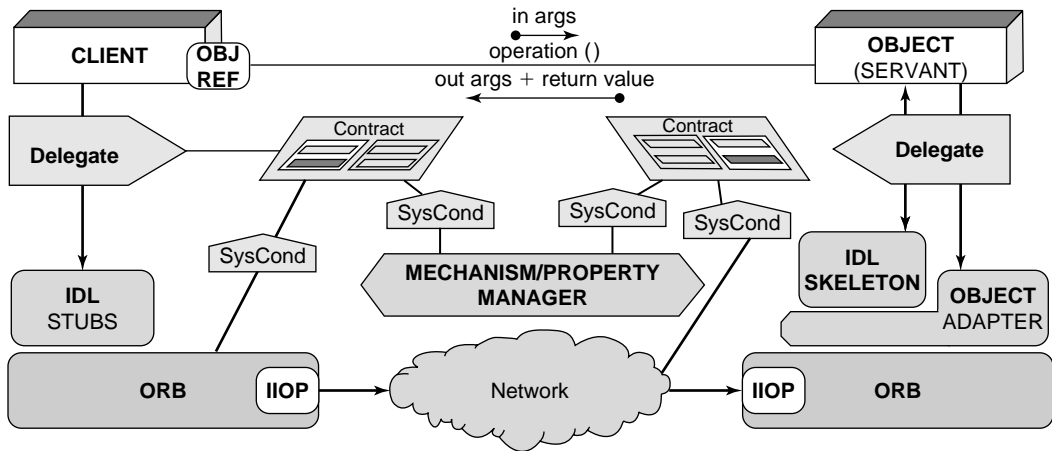


FIGURE 36.2 QuO adds components to control, measure, and adapt to QoS aspects of an application. (From Loyall, J. et al., in *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, AZ, pp. 16–19, April 2001. With Permission.)

QuO supports in-band and out-of-band QoS control and adaptation, as illustrated in Figure 36.3 [18]. QuO delegates trigger in-band adaptation by making choices upon method calls and returns. Contracts trigger out-of-band adaptation when changes in observed system condition objects cause state transitions. The QuO middleware is described in more detail in Refs. 17, 20, 30, 33, and 34.

36.3.1.2 Qoskets and Qosket Components

One goal of QuO is to separate the role of QoS programmer from that of application programmer. A complementary goal of this separation of programming roles is that QoS management code can be encapsulated into reusable units that are not only developed separately from the applications that use them, but that can be reused by selecting, customizing, and binding them to an application program. To support this goal, we have defined *qoskets* as a unit of encapsulation and reuse in QuO applications. Qoskets are used to bundle in one place all of the specifications and objects for controlling systemic behavior, as illustrated in Figure 36.4 [16]. Qoskets encapsulate the following QoS aspects:

- Adaptation and control policies—As expressed in QuO contracts and controllers.
- Measurement and control interfaces—As defined by system condition objects and callback objects.
- Adaptive behaviors—Some of which are partially specified until they are specialized to a functional interface.
- QoS implementation—Defined by qosket methods.

A qosket component is an executable unit of encapsulation of qosket code wrapped inside standards-compliant components, such as the CORBA component model (CCM) [4], which can be assembled and deployed using existing tools. They expose interfaces, so they can be integrated between functional components and services to intercept and adapt the interactions between them. Each qosket component offers interception ports that can be used to provide in-band adaptation along the functional path. Qosket components can also provide ports to support out-of-band adaptation and interact with related QoS management mechanisms, as illustrated in Figure 36.5 [16].

Qosket components provide all the features of qoskets and all the features of components to provide life cycle support for design, assembly, and deployment. Each qosket component can have as many adaptive behaviors as desired. However, encoding each qosket with one and only one adaptive behavior decouples different adaptive behaviors and increases the reusability of each. The trade-off is between the assembly time flexibility allowed by the separation of QoS behaviors versus the performance overhead of having additional components to assemble. This is the same design versus performance trade-off that

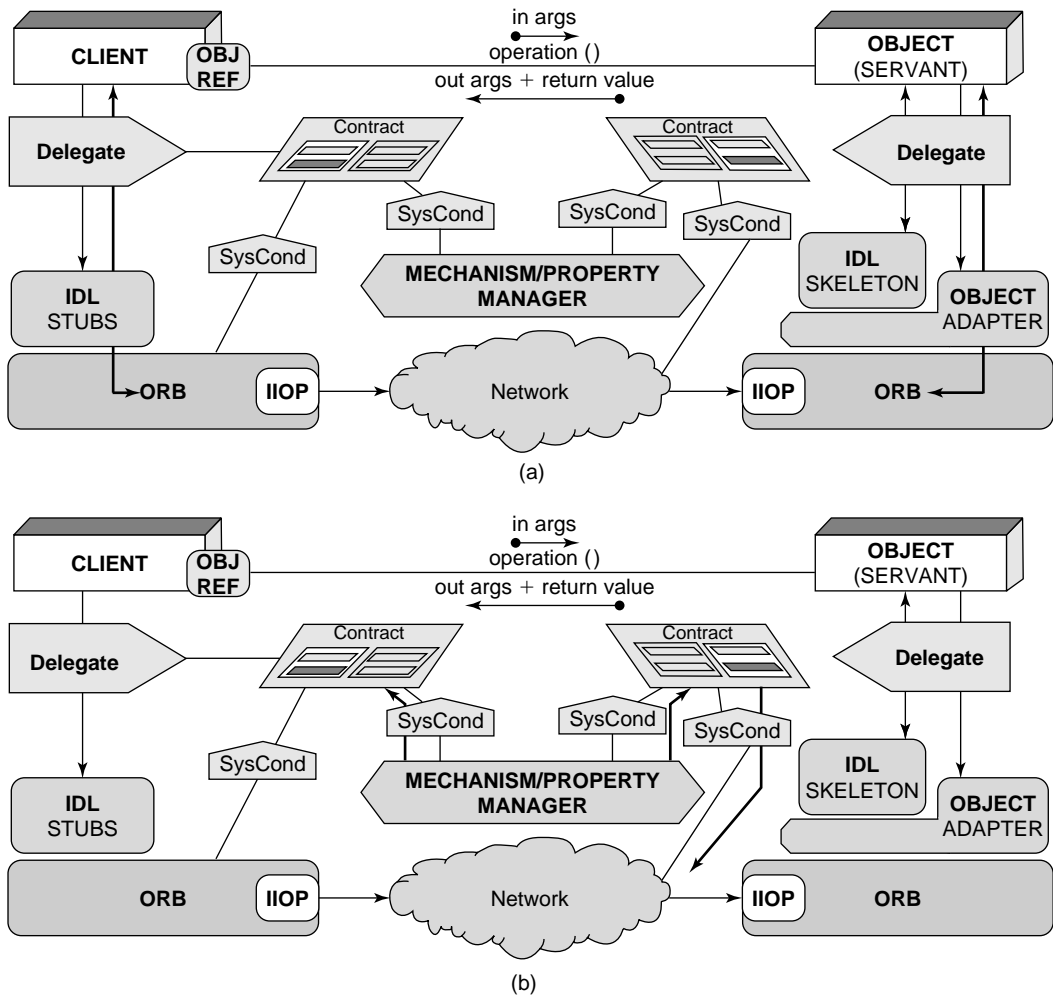


FIGURE 36.3 QuO supports adaptation upon method call/return and in response to changes in the system. (a) QuO's delegates provide in-band adaptation on message call and return. (b) Out-of-band adaptation occurs when system condition objects and contracts recognize system changes [18].

exists in functional component-based applications and which can be alleviated by assembly tools and component implementations that optimize component and container instantiations. Implementations that encapsulate a single QoS behavior in each qosket component can provide an aggregate, end-to-end behavior by combining qosket components. Additionally, there can occasionally be side effects associated with specific combinations of individual qoskets. This issue and its resolution are still under investigation (see Section 36.3.2.2).

The ability to enable dynamic adaptive QoS behavior by assembling qosket components has several important advantages. First, it makes it possible to manage end-to-end QoS requirements without requiring the existing middleware to completely provide such a service. This is important because there is no agreed-upon QoS standard for component standards (although there are proposals being considered). Second, the assembly, packaging, and deployment tools provided by the component middleware can be used for qosket components. Third, we have been able to create qosket components that are independent of the underlying middleware by separating code specific to the particular component model from the generic code required to manage and evaluate the QoS contracts. To date we have built qosket components

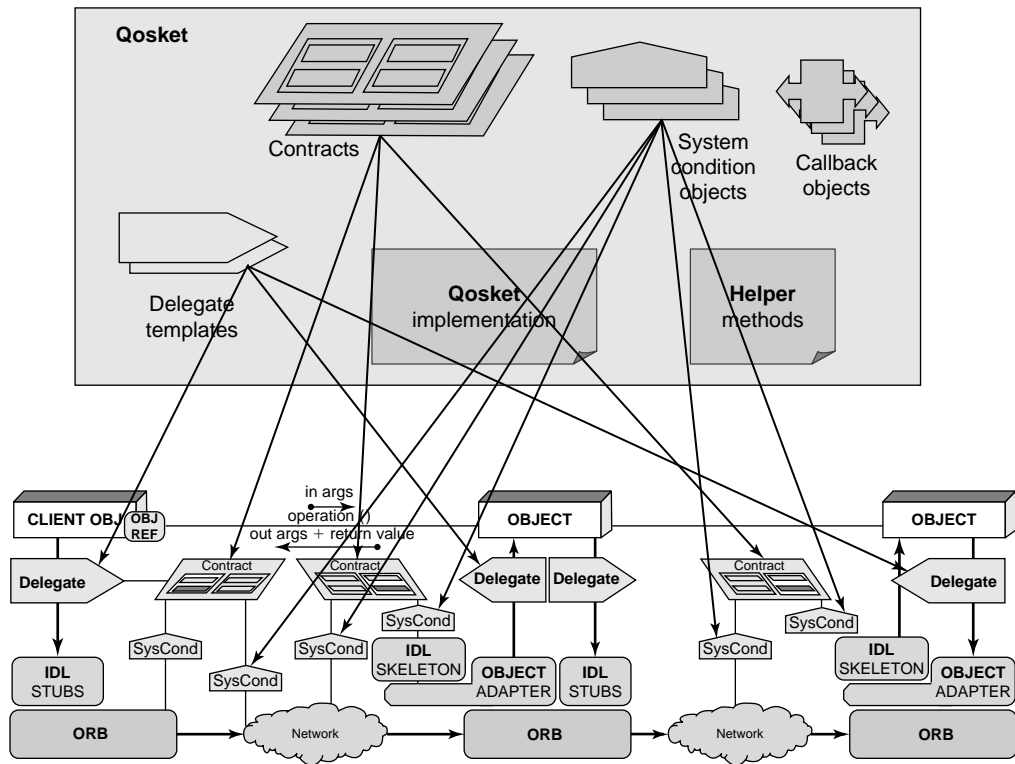


FIGURE 36.4 Qoskets encapsulate QuO objects into reusable behaviors. (Reprinted from Loyall, J. in *Proceedings of the Third International Conference on Embedded Software (EM SOFT 2003)*, Philadelphia, PA, pp. 13–15, October 2005. Copyright Springer Verlag. With permission from Springer Science and Business Media.)

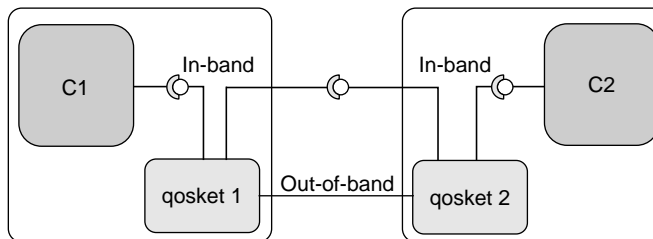


FIGURE 36.5 Qosket components can be assembled in-band or out-of-band in a component application. (Reprinted from Loyall, J. in *Proceedings of the Third International Conference on Embedded Software (EM SOFT 2003)*, Philadelphia, PA, pp. 13–15, October 2005. Copyright Springer Verlag. With permission from Springer Science and Business Media.)

for the CCM-based MICO [24] and component-integrated ACE ORB (CIAO) [3], the JavaBean-based Cougar [5], and an avionics domain-specific component model based on CCM (PriSm) [28].

36.3.1.3 Examples of Qosket Components

This section provides examples of the qosket components that we have developed and applied to the QoS management of DRE systems, such as those described in Section 36.4.

Network Management. We have constructed qoskets that prioritize network traffic by setting DiffServ CodePoints (DSCPs) [12] and that reserve bandwidth using RSVP [37].

CPU Management. We have developed a qosket that reserves CPU, using a CPUBroker [8], in Timesys's Linux RT operating system [13].

Application Adaptation and Data Shaping. The application needs to be able to shape its behavior to effectively use whatever resources are available to it and to gracefully handle degraded situations. This includes the ability to shape data so that whatever resources are available are used to deliver and process the data that is most important to the application's mission at that particular moment. To support this, we have developed the following set of data shaping qoskets, some of them specific to particular types of data:

- **Compression.** This qosket invokes off-the-shelf algorithms to compress data. We have versions that compress images using PNG, JPEG, and wavelet compression algorithms. When assembled as qosket components these need a corresponding decompression qosket at the receiving end.
- **Fragmentation.** For large-size data that needs to be sent over a period of time, such as a high-resolution image, a large database entry, or a large XML document, this qosket breaks the data into smaller chunks (the size is configurable). When assembled as a qosket component this needs a corresponding reassembly qosket at the receiving end.
- **Tiling.** This is an image-specific version of the fragmentation qosket, which breaks an image into tiles, each of which is an individually displayable part of the larger image. This allows each tile to be displayed as it is received, without waiting for the entire image to be received. It is often the case that important information can be viewed (received) earlier than background information.
- **Pacing.** This is a qosket used to control jitter and is particularly useful when combined with the fragmentation and tiling qoskets. It introduces packets into the network at a steady pace over a period of time to avoid bursty traffic.
- **Scaling.** Developed specifically for imagery, this qosket scales the size of an image, trading off resolution for content. The smaller scaled image will represent the same full picture, but will be reduced in resolution and size. A thumbnail image is an example of an image reduced in scale.
- **Cropping.** Again, developed specifically for imagery, this qosket reduces the size of an image by dropping whole parts of the image. It retains the resolution of the original, but loses part of the content and is useful when details about parts of an image are more important than others.

36.3.2 Software Engineering of QoS Management

In addition to middleware support for QoS management in DRE systems, it is useful to have tools and software engineering support for the construction of the QoS management features. This section describes three of these: aspect-oriented languages, composition patterns, and design environments.

36.3.2.1 QoS Aspect Languages

The initial inclination for an application programmer when designing QoS improvements into a program is to modify his code to examine its surroundings and take the appropriate measures. Unfortunately, this extra code tends to tangle the QoS management logic with the application logic, increases the complexity of, and reduces the maintainability of the application code. The recognition that QoS management is a concern that crosscuts the functionality of an application led us to the development of QoS aspect languages, which could be programmed separately from and then woven into the application code, or encapsulated as qosket components (which are then woven into the application).

The QoS aspect languages, which we developed as part of the QuO toolkit and are described in Ref. 7, were designed using the principles of traditional aspect-oriented programming (AOP) [15] in that they define specific points in the runtime behavior of a program at which crosscutting advice can be invoked. However, they extend the join point model provided by many instantiations of AOP, which concentrated on insertion points in the functional language semantics, such as control flow, data flow, and field accesses. The QuO aspect model extends this notion to associate advice with the distribution of method calls. This complements the use of other AOP languages, which can be used to weave crosscutting functionality into application functional components, by providing the ability to weave in aspects between these components as they are distributed throughout a system.

In addition, QuO elements, for example, contracts and system condition objects, provide the ability to weave code between the application and the system in which it is executing. This allows aspect-oriented insertion of crosscutting behaviors affecting the interactions between the program execution elements and the execution of the rest of the system, environment, and infrastructure in which the program is executing. These interactions are outside the standard join point model definition, since the interactions can be asynchronous from the execution path of the program. For example, QuO contracts and system condition objects can recognize and respond to overload in the system, regardless of the current state of the program execution. Furthermore, the QoS managed response could affect the future execution of the program, regardless of the locus of execution for the program. This combination of systemic programming and the aspect weaving described above enables the separate programming of QoS management capabilities that crosscut the functional dominant decomposition of the program by influencing the application behavior only, the system only, or both the application and the system together.

36.3.2.2 Composition and Composition Patterns

A DRE application often consists of many components—both functional and QoS. In the rare case in which these components are independent, the order of assembly or composition can be unimportant. However, in the usual case, the order of execution of these components must be carefully crafted to achieve the desired end-to-end, aggregate behavior (ordering sensitivity is one of the component composition side effects mentioned earlier which are under investigation). In some cases the way in which components are assembled is the difference between correct and incorrect behavior:

- Some qosket components must coordinate to implement a desired behavior. For example, a compression qosket must frequently be paired with a decompression qosket. These must be assembled in the correct order, since decompressing prior to compressing can result in undesired behavior.
- Some qosket components interfere with one another in a contradictory manner. For example, compressing or encrypting data might result in the inability to scale, crop, or tile data because it changes the data into a format that can no longer be manipulated. These qosket components must be composed in a compatible way, for example, scale or crop prior to compression (a variant of order sensitivity), or composed using a decision, for example, a contract determines whether to crop or compress, but not both.
- Some qosket components can affect the dynamics of one another. For example, any qosket component that includes processing, such as compression or encryption, can affect the dynamics of a CPU management qosket.

While the case studies in Section 36.4 describe specific compositions that we used to get the desired end-to-end QoS behavior in specific instances, there are a few general composition techniques that we have extracted that serve as patterns of composition, as illustrated in Figure 36.6 [31]:

- Layered Composition—In this pattern, qosket components that make higher-level decisions (such as a system resource manager, SRM and a QoS management component of larger scope and granularity) are layered upon qosket components that enforce these decisions (such as local resource managers, LRMs and QoS management components of smaller scope and finer granularity), which in turn are layered upon mechanism qosket components that control resources and QoS behaviors.
- Parallel Composition—When qosket components receive data simultaneously and perform their QoS behaviors independently, they can be composed in parallel.
- Sequential Composition—In some cases, a set of components must be tightly integrated such that a set of QoS behaviors are performed sequentially, with the output of each component becoming the input to the next component.

36.3.2.3 Design Environments

The middleware and software engineering technologies that we have described help improve the programming of DRE systems. Another critical area of software engineering support is help with designing these

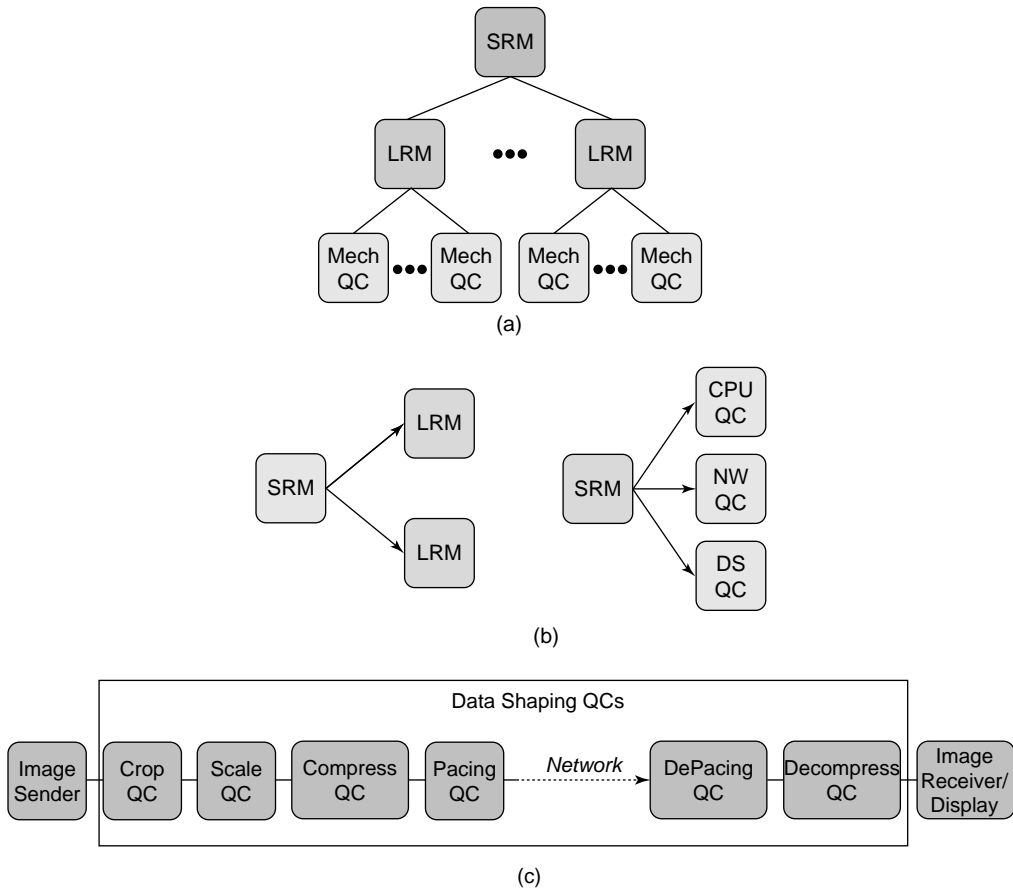


FIGURE 36.6 Composition patterns driving our uses of qosket components. (a) Layered composition—Each layer manages a set of qosket components below it. (b) Parallel composition—A set of qosket components needs to be invoked in parallel. (c) Sequential composition—Qosket components form a chain of composed behavior. (From Sharma, P. et al., *IEEE Internet Computing*, 10(3), 16–23, 2006. with permission.)

systems. Previous work in design environments has made designing and developing the functional logic of a distributed system much easier. However, support is not there for designing the QoS features. As part of the research described in this chapter, we created a prototype QoS design tool and applied it along with functional design tools to the design of the case studies in Section 36.4.

The distributed QoS modeling environment (DQME) [36], illustrated in Figure 36.7, was developed by BBN and Vanderbilt University and focuses on a QoS centric view of a distributed system. It captures the following essential elements of QoS and their interactions:

- **Mission requirements**—The functional and QoS goals that must be met by the application; relative importance of tasks; and minimum requirements on performance or fidelity. These help determine the relative importance and likely effectiveness of various QoS controls, adaptation strategies, and trade-offs.
- **Observable parameters**—Before any runtime QoS control action can be taken, the system has to know its current state with regard to the QoS of interest. These parameters determine the application's current state and are also important to help determine and predict the system's next state.
- **Controllable parameters and adaptation behaviors**—These are the changeable and tunable knobs available to the application for QoS control and adaptation. These can be in the form of interfaces to

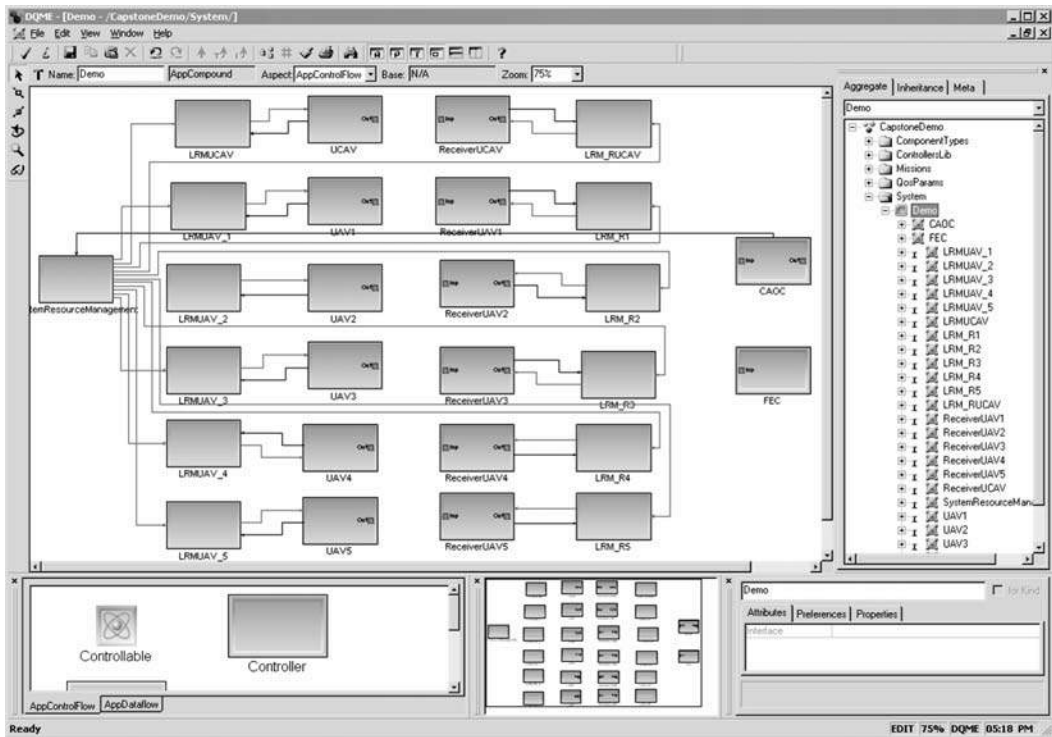


FIGURE 36.7 DQME enables the design of QoS management.

mechanisms, managers or resources, or in the form of packaged adaptations, such as those provided by QuO's qosket encapsulation capability (Section 36.3.1.2).

- System dynamics—Based on the current state of the system (observable parameters) and the set of knobs available (controllable parameters and adaptation behaviors), there are often several options for adaptation to meet QoS requirements. The interactions between these are reflected in the system dynamics and can help define the set of possible trajectories that an application can take.
- Control strategies—Specification of the control actions employed and the trade-offs made in response to dynamic system conditions to maintain an acceptable mission behavior. When making a decision, the control strategies take into account the mission requirements, current system state, and system dynamics. DQME supports the following types of controllers: state machines, supervisory control, feedback control, search spaces, region spaces, and classical compensators.

DQME supports hierarchical models of the above elements, with the higher levels representing the elements of end-to-end QoS adaptations in support of system-wide goals. The model developer can descend into these models to create models of local adaptations, based upon local parameters, but coordinated with the other subordinate models in support of the higher-level goals.

DQME was used to model the QoS management strategies employed in the Program composition of embedded systems (PCES) demonstration described in Section 36.4.3 and to generate qosket code for the resource manager in that demonstration. A version of DQME tailored for the signal analysis domain has been used in experiments to develop highly optimizing signal analyzers [22].

36.4 Case Studies of Providing QoS Management

This section describes three case studies of providing QoS management for DRE systems of increasing complexity, starting with a demonstration of QoS managed imagery dissemination from unmanned vehicles.

Next, we describe a flight demonstration of QoS management during dynamic planning between aircraft while in-flight. Finally, we describe end-to-end QoS management in a medium-scale live demonstration involving multiple vehicles and ground stations in a highly dynamic time-sensitive scenario.

36.4.1 HiPer-D, Reactive QoS Management in a US Navy Testbed

Our first case study applied the principles of dynamic QoS management to a US Navy context and the issues associated with coordinating and integrating operations of Unmanned aerial vehicles (UAVs). A UAV is an autonomous or remote-controlled, unpiloted aircraft that is launched to obtain sensor data (including imagery) for an area of concern. A UAV can receive remote-control commands from a control platform to perform actions such as changing its area of interest or tracking a target. To support UAV operations, a video camera on the UAV produces an imagery stream that, under certain modes of operation, must be displayed with requirements for minimal and predictable real-time delay and fidelity on consoles throughout a ship. As Figure 36.8 illustrates, there are several steps to this process [14]:

1. Video capture and feed from the off-board source (UAV) to the ship.
2. A process on the ship receives the video feed and distributes it to control stations on the ship's network.
3. Users' hosts receive video and display it.
4. Users analyze the data and (at certain control stations) send commands back to the UAV to help control it.

Our prototype simulates the first three of these steps. The command phase of the fourth step is formulated as a requirement to be able to control the QoS of the video distribution properly. The video must be displayed on the end-user's video monitor in a timely manner—if the data is too stale, it will not represent the current situation of the physical UAV and the scene it is observing—and with at least the minimum fidelity needed for visual cues—the user must see enough of the important imagery and motion. Otherwise, the user cannot control the vehicle appropriately or detect important features from the video stream. This means, among other things, that we cannot rely on the “best-effort” characteristics of common network transport protocols. The retransmission of dropped packets by the transmission control protocol (TCP) introduces unacceptable delay (violating the timeliness requirement), while the user datagram protocol (UDP) is not discriminating enough in its packet dropping—it is as likely to drop packets from important images as from less-important ones. We choose to control the QoS in the QuO middleware layer where the application's requirements can directly influence the QoS control and adaptation policies that are available for this use case. For example, it is not acceptable to suspend the display during a period of network congestion and resume the display from the same point in the video flow when bandwidth is restored. It is also not acceptable to simply drop arbitrary frames or to continuously attempt

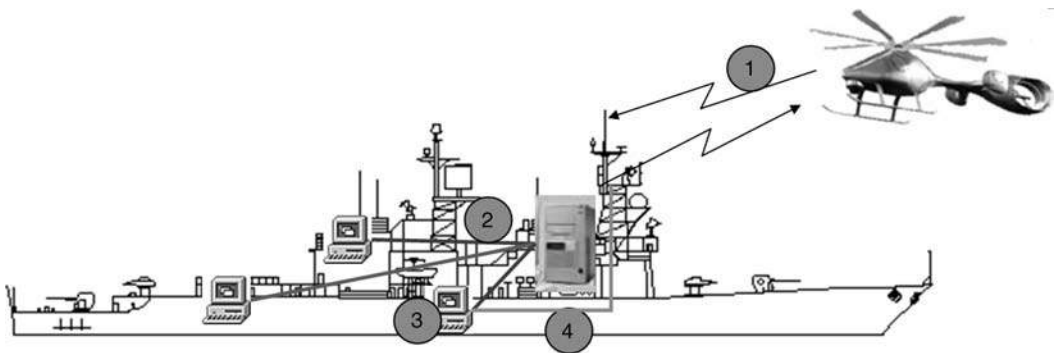


FIGURE 36.8 The UAV use case [14]. (From Karr, D. et al., in *Proceedings of the International Symposium on Distributed Objects and Applications*, Rome, Italy, pp. 18–20, September 2001. with permission.)

to retransmit lost frames. Trade-offs must often be made of one property (e.g., timeliness) against another property (e.g., fidelity) based on the particular requirements of the end-user at that moment.

Figure 36.9 illustrates the architecture of the prototype [29]. It is a three-stage pipeline, with simulated UAVs or live UAV surrogates sending imagery to processes (distributors), which distribute the imagery to the simulated C2 stations, which display, collect, or process the imagery (e.g., with an automated target recognition process). Various versions of the UAV image dissemination prototype have been used for evaluation within the US Navy's HiPER-D platform, as a basis for US Army and US Air Force prototype systems, and as an open experimental platform for DARPA's PCES program. The UAV prototype uses the QuO middleware to integrate several QoS management strategies, services, and mechanisms:

- End-to-end priority-based resource management, using RT-CORBA and scheduling services for CPU resource management and DiffServ [12] for network priority management.
- End-to-end reservation-based resource management, using the CPU reservation capabilities of Timesys Linux and RSVP network reservation [37].
- Application-level management, including frame filtering, compression, scaling, and rate shaping.

The UAV image dissemination prototype employs QoS management all along the pipeline, with CPU management at each node, network management at each link, and in-band and out-of-band application QoS management at several locations. We used AOP techniques to separate the QoS and adaptive control aspects from the functional aspects of the program, as illustrated in Figure 36.10.

One of the QoS management techniques that we utilized is *frame filtering*, selectively dropping intermediate frames to maintain smooth video when bandwidth is constrained, described in detail in Ref. 14. We performed experiments to test and evaluate the effectiveness of the frame filtering adaptation in the UAV application. The three stages were run on three Linux boxes, each with a 200 MHz processor and 128 MB of memory using TCP.

At time $t = 0$, the distributor started. Shortly after this, the video began to flow. At $t = 60$ s, $t = 62$ s, and $t = 64$ s, three load-simulating processes were started on the same host as the distributor, each attempting to use 20% of the maximum processing load (a total of 60% additional processing load). This reduced the

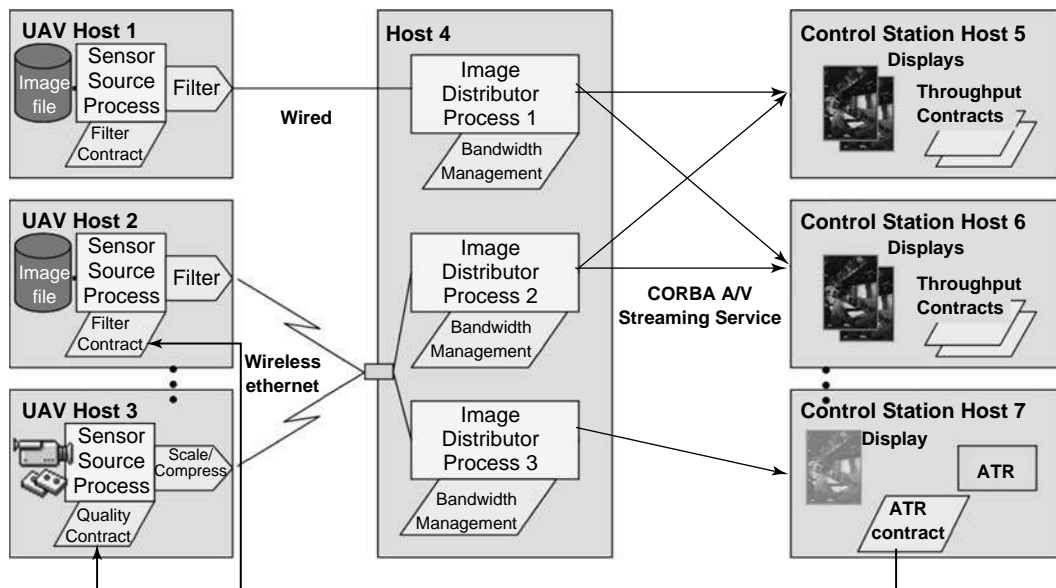


FIGURE 36.9 Architecture of the UAV Prototype. (Reprinted from Schanty, R. et al., in *Proceedings of the ACM/IPIP USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003. Copyright Springer Verlag. With Permission from Springer Science and Business Media.)

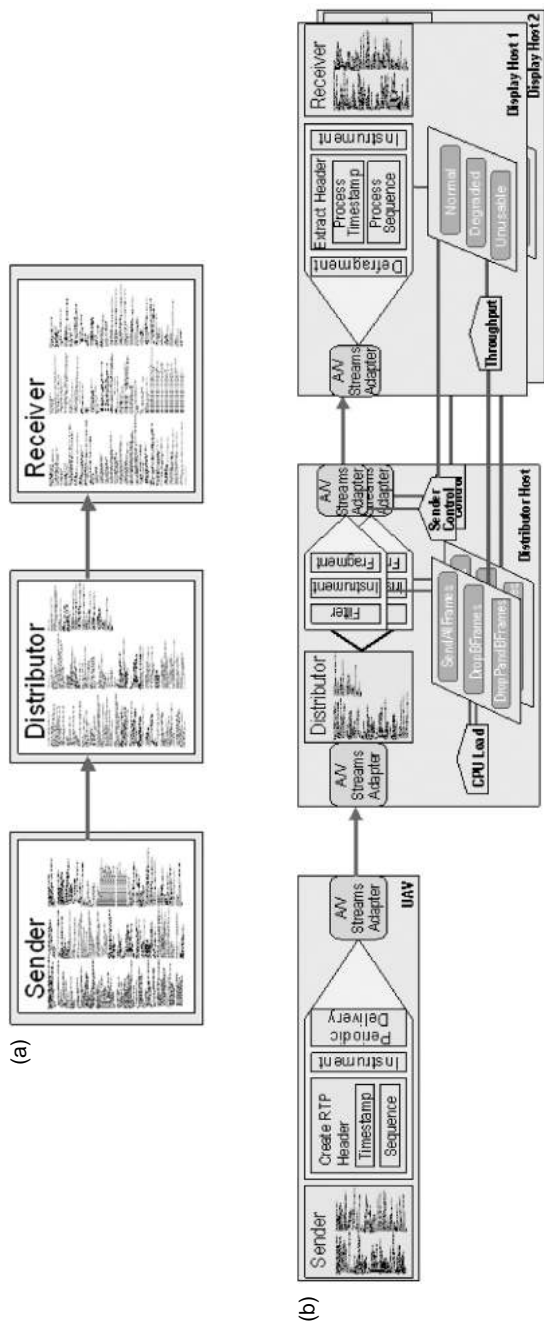


FIGURE 36.10 For the HiPer-D demonstration, we used aspect languages to separate out intertwined QoS aspects. (a) Early UAV application code with tangled concerns. (b) UAV application built using aspect-oriented techniques.

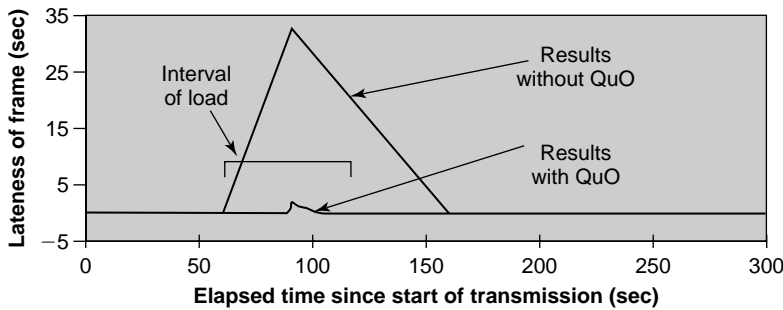


FIGURE 36.11 Lateness of frames received by viewer [14].

distributor's share of processing power below what is needed to transmit video at 30 frames/s. At $t = 124$ s, the load was removed. At time $t = 300$ s (approximately), the experiment terminated. The basic premise is that the full load was applied for 1 min, starting after the pipeline had time to "settle in," and ending a few minutes before the end of measurement so we could observe any trailing effects.

This scenario was run twice, once without QuO attached and without any adaptation (the control case) and once with a QuO contract controlling adaptation (the experimental case). For the purposes of this experiment, the only adaptation enabled was to reduce bandwidth by selectively dropping frames. Figure 36.11 shows the effect of the increased load on the latency of the video stream [14]. In this graph, the x -axis represents the passage of time in the scene being viewed by the video source and the y -axis represents the "lateness" of each image, that is, the additional latency (in delivery to the viewer) caused by the system load. If all images were delivered with the same latency, the graph would be a constant zero. The label "Interval of Load" indicates the period of time during which there was excessive contention for the processor. Without QuO adaptation, the video images fall progressively further behind starting when the contention first occurs, and the video does not fully recover until some time after the contention disappears. With QuO-enabled QoS adaptation using frame filtering, we have reduced frame delivery latency under load to only a slight perturbation of a steady flow.

The outcome of this experiment demonstrates that adaptation can provide improved performance of the application in the form of smoothing critical performance metrics over changing conditions. The added latency caused by adverse system conditions (in this case, excessive CPU load) occurs in a sharply reduced magnitude and duration when adaptation is enabled, and the video image is continuously usable for its intended real-time purpose despite the fluctuation and occasional dropped frames.

36.4.2 WSOA, Hybrid Control and Reactive QoS Management in a Dynamic Mission Replanning US Air Force Flight Demonstration

The Weapon System Open Architecture (WSOA) program was a US Air Force project to develop and prototype capabilities for aircraft to do mission replanning while airborne *en route* to a target (as opposed to on the ground prior to a mission). WSOA prototyped a dynamic collaborative mission planning capability that could be used in-flight between a C2 aircraft and a fighter aircraft. As illustrated in Figure 36.12, the fighter aircraft and the C2 aircraft establish a collaboration to exchange virtual target folders, consisting of images and plain text data to update the fighter's mission [18]. The end goals are to shorten the mission planning time (from hours to minutes) and increase the confidence in mission execution.

WSOA was built upon a layered architecture of domain-specific, QoS adaptive, and distribution middleware, as well as QoS mechanisms, services, and managers. Middleware on the fighter node consisted of

- The Bold Stroke avionics middleware [32]
- QuO QoS adaptive middleware
- TAO distribution middleware

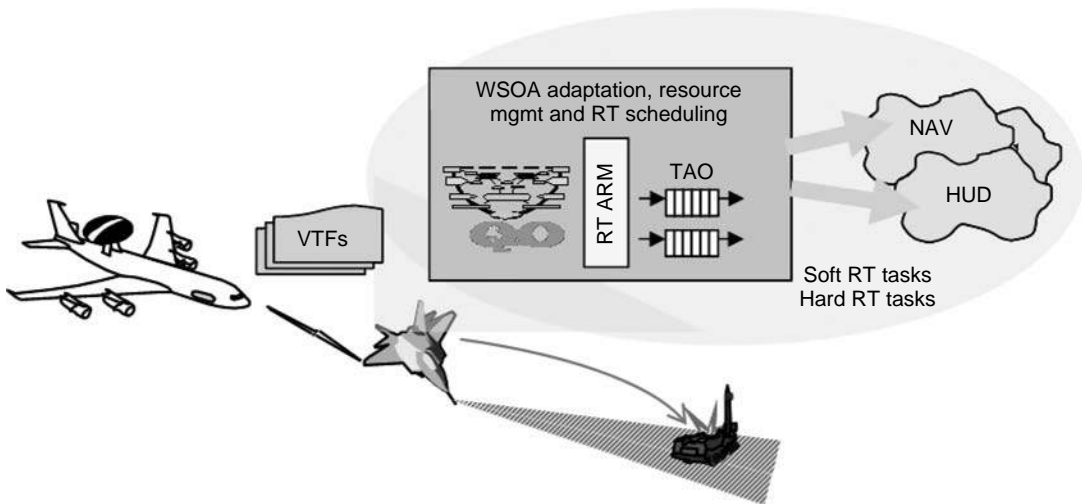


FIGURE 36.12 Collaborative replanning in WSOA [18].

The C2 node utilized ORBexpress, a CORBA object request broker (ORB) supporting the C2 legacy software applications written in Ada, and Visibroker. QoS mechanisms, services, and managers in WSOA consist of

- Real-time adaptive resource manager (RT-ARM) [11], a CPU admission control service, on the fighter node
- Kokyu [9], a real-time scheduling and dispatching service, on the fighter node
- Tactical communication links, using Link 16 [35]

Processing on the fighter node involves hard and soft real-time tasks. Hard real-time tasks are statically scheduled to assigned rates and priorities. WSOA used QuO, RT-ARM, and Kokyu to schedule soft real-time tasks using the extra CPU cycles available based on windfall from the worst-case allocation, but unused in the usual case, and to adapt processing and task rates dynamically to maintain the operation of hard and soft real-time tasks on the fighter's mission computer.

The layered architecture of WSOA is illustrated in Figure 36.13 [18]. The nature of the Link 16 tactical network means that bandwidth is statically allocated, so dynamic management of the network has to be on the edges, that is, adjusting the timing, amount, and nature of the data being transported. WSOA used a QuO contract, whose control behavior is illustrated in Figure 36.14, and a delegate (wrapping a `get_image()` method) to manage the imagery download as a sequence of tiles of varying quality, starting with a point of interest in the image [18]. The QuO contract continuously monitored download progress and the delegate adjusted image tile compression reactively based on the current contract state (i.e., early, on time, or late) to meet the required image transmission deadline. Image tiles near the point of interest—which are the first to be transmitted—were sent at as low compression as possible to improve image quality, but surrounding images can still be useful at lower resolution. In addition, the QuO contract interfaced with RT-ARM to adjust the rates of execution for the decompression operation.

On December 11, 2002, Boeing and the USAF conducted a flight demonstration of the WSOA software in Missouri. An F-15 aircraft equipped with the WSOA software took off from Lambert-St. Louis International Airport with a preplanned mission and target—an exposed aircraft just Northwest of Hannibal, Missouri, over 100 miles away. A simulated airborne warning and control system (AWACS), hosted in a Boeing 737 flying laboratory, received a simulated alert from a joint tactical terminal (JTT) of a time critical target, in this case an SA10 surface to air missile being set up due west of Hannibal. The F-15 and AWACS established an airborne collaboration, exchanged intelligence, imagery, and target information, and the F-15 redirected

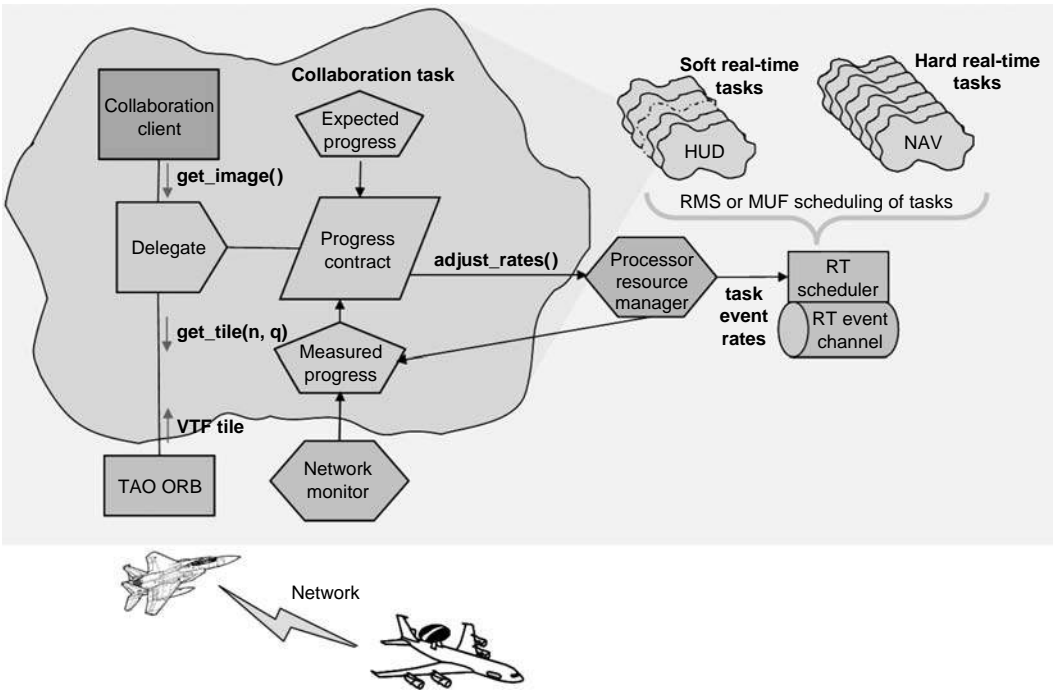


FIGURE 36.13 The WSOA layered architecture included quality objects technology providing dynamic QoS management [18].

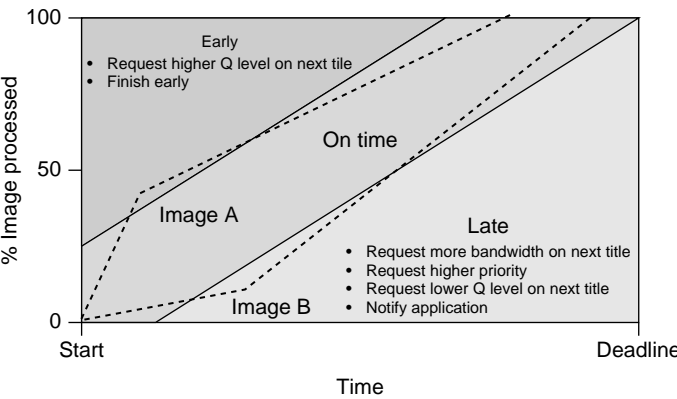


FIGURE 36.14 A QuO contract monitors the progress of image download and adapts to keep it within the “On Time” range [18].

its mission to simulate prosecution of the SA10. Within 20 s of establishing the collaboration, the F-15 Weapon System officer was receiving useful imagery through the Link 16 tactical link between the two aircraft.

We evaluated image download times from the C2 node to F-15 compared to deadlines of 38, 42, 46, 50, 54, and 58 s. The 38 s deadline is lower than the measured minimum latency for any image downloaded without adaptation at the highest compression ratio of 100:1, so that meeting it is infeasible. Similarly,

TABLE 36.1 Image Latency with Adaptive Compression and Scheduling

Deadline	Image 1	Image 2	Image 3	Image 4
42	40.82	41.51	41.06	42.57
46	41.42	42.08	42.43	42.16
50	44.21	43.76	43.98	44.93
54	52.02	53.88	47.27	47.6
58	52.3	54.71	55.4	54.42

a deadline of 58 s exceeds the maximum latency of any image at the lowest-compression ratio of 50:1, and thus does not require any adaptation.

Table 36.1 shows the end-to-end image download latencies with reactive adaptation of compression ratios and adaptation of operation invocation rates. The adaptive QoS management enabled the successful image transfer within its deadline, in all cases but the tightest (42 s) of the feasible deadlines. In the 42 s case, the limited slack in the deadline was insufficient for (only) one of the tests to successfully adapt the image transfer to meet the deadline.

36.4.3 PCES, End-to-End and Multilayered Controlled QoS Management in a Multiplatform Live-Flight Demonstration

As part of the Defense Advanced Research Projects Agency's (DARPA) PCES program, BBN, Boeing, and Lockheed Martin developed a capstone flight demonstration of advanced capabilities for time critical target (TCT) surveillance, tracking, and engagement. The PCES capstone demonstration, depicted in Figure 36.15, was a medium-scale DRE application, consisting of several communicating airborne and ground-based heterogeneous nodes in a dynamic environment with changing mission modes, requirements, and conditions [19]. It consisted of a set of UAVs performing theater-wide surveillance and target tracking and sending imagery to, and under the control of, a C2 Center. Specific UAVs could be commanded to concentrate their surveillance on areas of interest. When a positive identification of a threat was made, the commander directed engagement by ground or air combat units. Specific surveillance units then gathered battle damage indication (BDI) imagery and the process was repeated as needed.

The capstone demonstration was conducted at White Sands Missile Range (WSMR) in Spring of 2005. On the north end of the demonstration, we had two live ScanEagle UAVs and four simulated ScanEagles. 100 miles to the south was the C2 Center, which consisted of multiple machines, including situational assessment (SA) displays, command and control processing, and displays of the reconnaissance and BDI imagery. Network connectivity from north to south was through a resource constrained, shared network.

The PCES capstone demonstration application exhibits the significant challenges and issues concerning managing overall QoS in a complex, multifaceted rapidly changing environment that are indicative of medium-scale DRE applications with similar characteristics. The following sections discuss how we applied the adaptive QoS management concepts and techniques of Section 36.3 to address these challenges and build the demonstration system.

36.4.3.1 Mission Modes and Roles Drive System Requirements

The UAV (ScanEagle) camera transmits imagery through a 2.4 GHz analog television downlink. We captured the imagery from the ScanEagle ground station receiver and piped it through an analog to digital converter, so we could control the resolution, rate, and size of the imagery, controlled by factors such as how the imagery was to be used, the amount of resources currently available to transmit and process the imagery (i.e., network bandwidth and CPU), and the number of UAVs sharing the resources. The way in which information was to be used in the mission determined the changing requirements on the system for the delivery, format, and content of surveillance and C2 information. In the PCES capstone

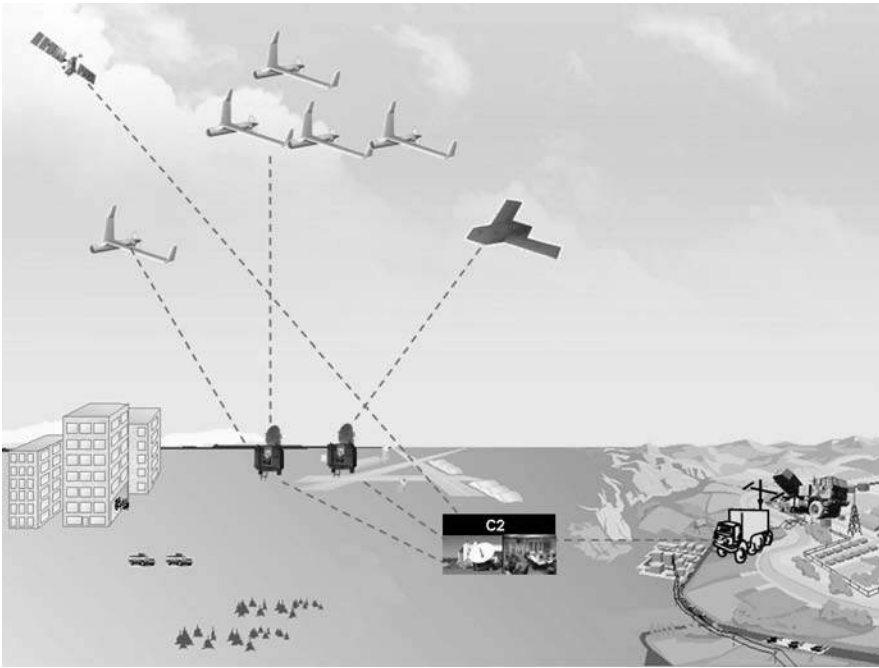


FIGURE 36.15 The PCES capstone demonstration concept of operations includes multiple RUAVs, combat vehicles, and command centers [19].

demonstration, there were three roles that UAVs perform, each with their own distinct requirements on their end products developed over the same set of shared physical resources:

- Surveillance
- Target tracking and engagement
- BDI

When the demonstration began all the UAVs were performing surveillance. As the scenario unfolded, the C2 node would direct a UAV to reroute in response to a potential TCT, prompting a change by the UAV to the target tracking role. After target engagement, a UAV would be directed to perform the BDI role.

The TCT mission provided a relative priority to each of these roles, target tracking and engagement was the most critical, BDI second most critical, and surveillance was the least critical. Each role, likewise, included attributes for its desired QoS and the trade-offs it could afford. A UAV performing target tracking needs to provide high-rate and high-resolution imagery so that positive target or threat identification can be made. If the target was moving, this translated into full motion MPEG video. If the target was still, then the imagery rate could be lower.

For BDI, the UAV needs to provide high-resolution imagery until a human operator determines that he has sufficient detail to discern battle damage. Imagery does not need to start immediately, since dust and smoke will obscure the scene immediately after engagement, but once imagery has started, high-resolution imagery must be delivered regularly, although not necessarily at high rate, until a commander decides it is sufficient.

Finally, for UAVs performing surveillance, the primary mission is to maximize the surveilled area with sufficient resolution for a commander to determine an item of interest. This means that imagery from each surveillance UAV must be sent at a sufficient rate to ensure there are no gaps in surveillance coverage and at sufficient size and resolution for a commander to discern command-level detail.

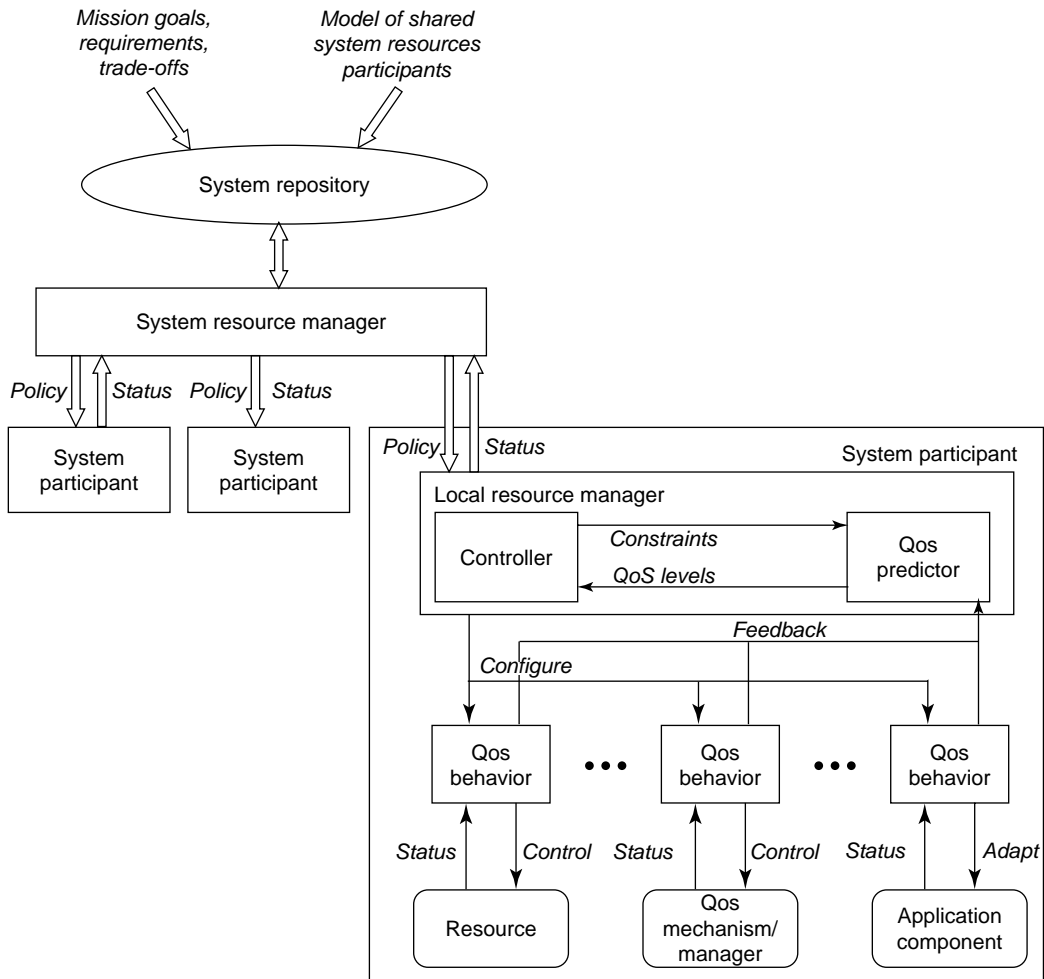


FIGURE 36.16 Elements of multilayer QoS management [23].

The demonstration exhibits the challenges we described in Section 36.2:

- Multilayered mapping of the mission- and role-based requirements to QoS enforcement
- Managing QoS from end-to-end for each UAV sensor to C2 node stream
- Mediating the aggregate QoS across competing streams, and
- Handling the QoS dynamically as the roles of participants change over time

To manage these dimensions of QoS in the PCES capstone demonstration, we developed a multilayered, dynamic QoS management architecture, illustrated in Figure 36.16 [23]. The system resource manager (SRM) is a supervisory controller responsible for allocating resources among the system participants and for disseminating system and mission wide policies to local resource managers (LRMs). These policies include the resource allocation, the relevant mission requirements and parameters, and trade-offs.

An LRM is associated with each demonstration participant, such as a UAV sensor. The LRM receives the policy from the SRM and translates it into local management and control actions. The LRM is a feedback controller, using the mission requirements, trade-off information, and allocated resources part of the policy provided to it to determine which QoS behaviors (e.g., CPU management, network management, data shaping, and application adaptation) should be employed, in what order, and to what degree. The LRM also monitors the actual behaviors and adjusts as needed to maintain the QoS level.

To determine which QoS behaviors to employ, the LRM needs to use a system dynamics model to predict the effect of employing each QoS behavior and combination of QoS behaviors. In Figure 36.16, we separately indicate the control and prediction parts of the LRM, the former illustrated as a controller and the latter as a QoS predictor. The system dynamics (i.e., effect) of some QoS behaviors can be determined analytically, for example, the results of cropping an image (i.e., the amount of data in the resulting image) or reserving an amount of bandwidth (i.e., the amount of bandwidth available to the application). Other behaviors have no analytical model (or less accurate ones), for example, some compression algorithms or setting a network priority (the results of which are difficult to determine analytically without global knowledge of many other external factors). With the former, the QoS predictor contains the model, equation, or formula to predict the behavior. With the latter, the QoS predictor is initialized with experimental data produced in test runs, and updated at runtime with more accurate monitored information.

The QoS mechanism layer consists of encapsulated QoS behaviors that control and monitor the following:

- Resources, such as memory, power, or CPU, which can be monitored and controlled through knobs exposed by the resource.
- Specific QoS mechanisms, such as network reservation or network priority services that expose interfaces to resource monitoring and control; or QoS managers, such as bandwidth brokers [6] or CPU brokers [8], that provide higher-level management abstractions.
- Application or data adaptation, such as changing the rate of tasks, algorithms or parameters of functional routines, or shaping the data used or produced by application components.

36.4.3.2 Construction of DRE Systems

The PCES capstone application illustrates challenges in the manner in which medium- and large-scale DRE systems are, and will be, built. Traditionally, these types of systems are built as one of a kind, stovepiped systems. In contrast, we constructed the multilayered QoS management and the end-to-end UAV to C2 imagery streams by composing it from reusable QoS and functional components as described in Section 36.3.2.2.

We implemented the elements of our end-to-end QoS management architecture as qosket components so that they can be assembled with the components of the functional application, as illustrated in Figure 36.17 [23]. The SRM qosket component includes decision-making code to decide how resources should be allocated among participants and wraps that allocation into a policy, with some monitoring code to determine the number of current participants, the amount and type of shared resources, and other information affecting the policy decision, such as mission states, requirements, and conditions.

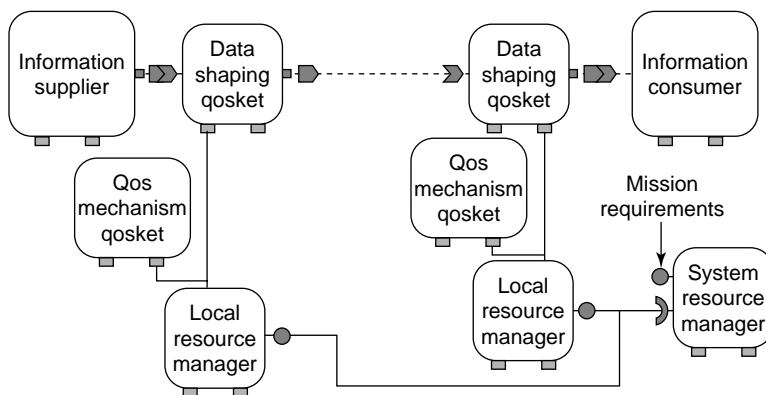


FIGURE 36.17 End-to-end QoS management elements are instantiated as qosket components and assembled with the functional components [23].

The LRM qosket components include decision-making code to decide local actions based on the policy, monitoring code to measure the effects of the QoS management, and control code to adjust levels to satisfy the policy. The LRM's control code is typically limited to setting the proper attributes on the QoS behavior for the lower-level qosket components and invoking them in the proper order.

The assembly also includes as many QoS behavior qosket components as necessary. In the example in Figure 36.17, we illustrate two types of QoS behavior qosket components, one that does data shaping and another that interfaces to an infrastructure QoS mechanism.

The base functionality of each end-to-end imagery stream consists of image capture (i.e., the UAV's camera sensor and associated processing) and image sending (i.e., communicating the imagery off-board) on the UAV; and the image receipt, display, and processing on the C2 node. The image generation rate is a configurable parameter that indicates how often an image should be pushed out, which is determined by the usage requirements of the imagery, and can be different than the rate at which it is collected. We used both CIAO [3], an implementation of the CCM, and PRiSm [28], an avionics specific component model and component successor to the Bold Stroke middleware we used in the WSOA demonstration described in Section 36.4.2. The full scope of the demonstration system includes a combination of live flight vehicles, ground vehicles, and simulated participants, and is described in Ref. 19.

We augment the functional stream with qosket components as described in Section 36.3 to get end-to-end QoS management. The full assembly for a representative imagery stream is illustrated in Figure 36.18 [23]. There is one SRM component, which we locate at the C2 node, so it is near the receivers and the command authority, both of which provide information needed to determine the mission requirements. In PCES, the SA command component keeps track of the number and role of participants and serves as the system repository. When something significant in the system state changes, such as the number or role of participants, the SA component notifies the SRM component. The SRM uses the relative weights of the roles, the importance of each UAV within a role, the number of UAVs, and the amount of resources available, to (re)compute a resource allocation for each UAV. It creates a policy structure for each participant consisting of the following:

- The UAV's role (surveillance, target tracking, or BDI)
- The UAV's importance relative to others in the role
- Allocations of resources (bandwidth, CPU, network priority)
- Minimum and maximum allowable qualities (frame rate, cropping, scale, compression, and CPU reservation)

This policy event is pushed to each of the LRMs, which in turn recompute their own local behavior.

There is an LRM component associated with the sender assembly and another one associated with the receiver assembly for each end-to-end stream. Each receives the policy sent by the SRM and updates the relevant QoS predictors with the minimum and maximum cropping, scaling, and compression levels. The LRM then queries their internal QoS Predictors to get the proper levels to set for each of the data shaping components to fit the allocated bandwidth and CPU. The adaptation strategy and trade-offs for each role is captured in a model of the system [21], and is used to determine the order of assembly and invocation of components. For example, the strategy we use for the surveillance role is to allow the rate to be reduced until the acceptable minimum (the slowest rate that does not cause gaps in surveillance); permit greater levels of compression until the maximum allowed compression; and scale the image as a last resort if further reduction is needed. The LRM then sets each of the following QoS mechanism qosket components with the proper settings from the policy and QoS predictors.

The Diffserv qosket component is responsible for setting DiffServ codepoints (DSCPs) on component containers. The LRM uses the network priority from the SRM policy to configure the Diffserv component to ensure that all packets going out have their DSCP set correctly. Routers configured to support Diffserv ensure that the packets get queued according to their DSCP priorities.

The CPU Broker qosket component is responsible for reserving CPU cycles over a period of time for a component container. The LRM uses the minimum and maximum CPU reservation and the relative importance from the SRM policy to configure the CPU Broker component. The underlying CPU

mechanisms (CPU Broker [8] and TimeSys Linux) guarantee that the container gets at least the minimum CPU cycles it needs. In the case of CPU contention, no more than the maximum CPU cycles are allocated to the container.

Once the available CPU and network resources have been allocated across UAV streams, each stream must shape its data to use the allocated resources effectively. Data shaping qosket components are a collection of individual data shaping capabilities. We assemble several data shaping qoskets that the LRM uses to accomplish matching available data transmission resources with effective use:

- Fragmentation, pacing, and defragment qosket components are combined to reduce jitter in the network by spreading the transmission of data evenly over the interval specified by its rate. The LRM configures them with the allocated bandwidth and a fragment size (the maximum transmission unit of the network is a logical choice). The fragmentation component breaks an incoming image into fixed-sized fragments and the pacing component sends the fragments over the network at regular intervals. Fragmentation on the sender side is accompanied by assembly (or defragmenting) on the receiver side. The defragment component receives fragments and, once it has received all the fragments of an image, reconstructs the image.
- The compress qosket component is responsible for compressing an image. The level of compression is set by the LRM as specified by the QoS Predictor.
- The crop qosket component removes a specified amount of the image from a set place in the image. The amount of the image that is cropped is set by the LRM as specified by the QoS predictor. In the current prototype, we crop from the center of the image, but in general, images could be cropped around an area of interest regardless of where it is in the image.
- The scale qosket component reduces the size of an image. The LRM sets the amount that the image is scaled as specified by the QoS Predictor.

Assuming there are sufficient resources, the SRM ensures that every end-to-end image stream gets at least the minimum it needs and that the more important streams get the majority of the resources. In cases where there are not enough resources for all the streams, the SRM ensures that the most important streams get the resources that are available. The LRMs in turn ensure that the allocated resources for each end-to-end stream are used most effectively for the UAV's particular role in the system. They utilize specific lower-level mechanism qoskets to effect, control, and enforce these behaviors.

Figure 36.19 illustrates an example of the design of the QoS management system using the DQME design environment. In addition to using the DQME tool to design the QoS management architecture and algorithms, we used the component assembly and deployment modeling language (CADML) tool [2] to design the assembly of the functional and QoS components in the demonstration and generate the XML CAD file used by the CIAO component infrastructure. The component assembly of the PCES capstone demonstration is shown in Figure 36.20.

36.4.3.3 Composing Separately Developed Subsystems

The PCES capstone demonstration application combined US Air Force and US Army capabilities built by several different organizations, at different times. These included C2 subsystems, sensor and weapon systems, and system element simulations. An *ad hoc*, tight coupling of these systems to integrate them for the demonstration would have gone against our goals of eliminating stovepiped system architectures, even if it would have been possible. However, even more realistic motivation for avoiding such tight integration existed: several of the subsystems were legacy systems; many of them were already large, complex systems in their own right; and they were written in different languages and hosted on different platforms.

In the integration of these, we had two primary complexities that we needed to avoid:

- Start-up dependencies—Because the subsystems were developed independently, each had application-specific ways of handling their inputs and outputs. When they were composed into the capstone demonstration, they often took on different roles relative to each other: some functioned as clients to others, some as servers, and some functioned as both clients and servers. It was

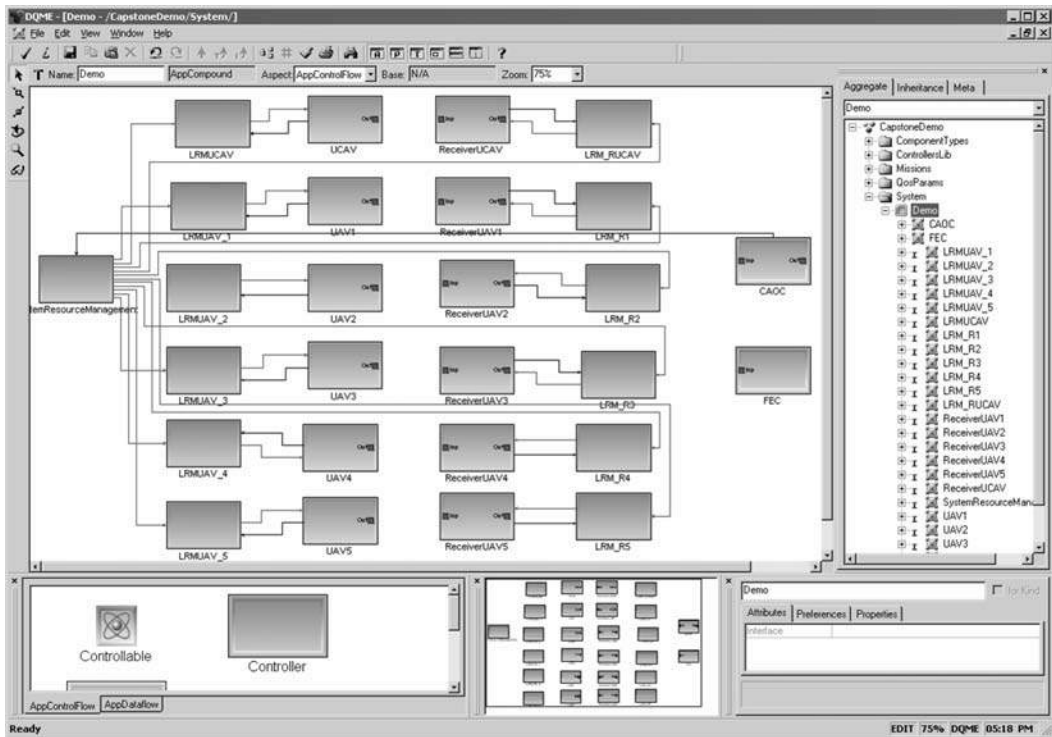


FIGURE 36.19 DQME Model of the QoS management design in the PCES capstone demonstration.

easy to fall into the trap of an accidental, but complicated imposed dependence on startup order, in which server elements of the subsystems had to be created and initialized in a particular order so that they existed before their clients looked for them.

- Runtime coupling—An *ad hoc* approach to integrating individual subsystems presented another danger, namely of making them too tightly coupled. Not only would doing so make the architecture of the composed system less modular, but it presented additional dangers for the development of the system. The capstone demonstration was large, with many interoperating pieces. During development, integration, and demonstration rehearsals, crashes of individual pieces were common. When one piece crashed, or was taken down for repair, anything functioning as a client of the now missing piece would get transient errors or crash. In a tightly coupled system, this was time consuming, requiring restarting the whole system, or large parts of it.

The traditional software engineering approach to solving the first problem is to enforce startup order using startup scripts. The second problem would traditionally be handled by introducing exception handling in each component, so that it caught and gracefully handled transient errors. However, both of these solutions introduce undesirable and avoidable tighter coupling into the system—a change to, absence of, and addition of, any component and subsystem would affect elements in other subsystems. From a software engineering perspective, individual subsystems should be able to start and function on their own or as part of a composed system and should be able to start and evolve independently, with well-defined and limited effect on other subsystems.

To achieve this goal, we used a loosely coupled design incorporating the following middleware services to integrate the pieces of the composed capstone demonstration, as illustrated in Figure 36.21:

- The CORBA Notification Service, which provides a CORBA standard publish/subscribe service that allows any interface description language (IDL)-defined structure to be published as an event.

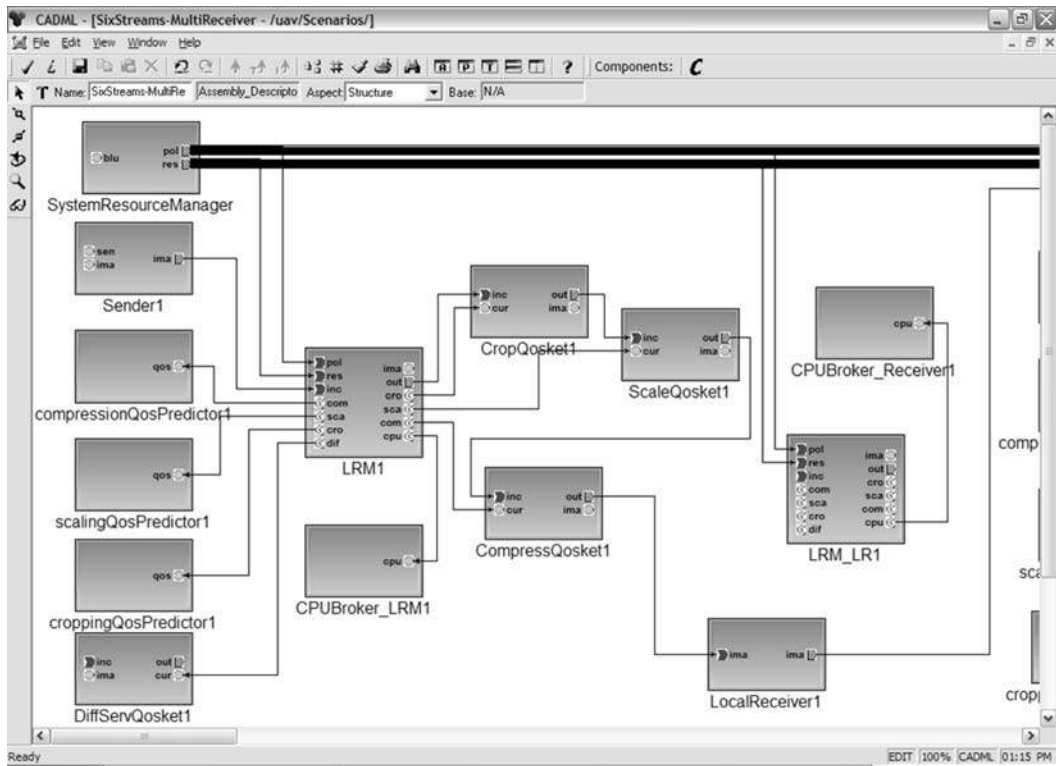


FIGURE 36.20 CADML model of capstone demonstration.

- The Global Information Grid/Joint Battlespace Infosphere (GIG/JBI), a US Air Force standard publish/subscribe information management system.
- The CORBA Real-Time Event Channel, a CORBA standards-based middleware service that supports the real-time exchange of small messages (events).

Since each piece only has references to the middleware service endpoints and not direct references to other pieces, it is not directly affected by other pieces going down or coming up. In addition, it was much easier to add a new piece or communications path to the system. Finally, it was easier to test subsystems or groups of subsystems, since each subsystem could run independently or could be replaced by a test driver that coexisted with the real pieces.

36.4.3.4 Performance of the PCES Capstone Demonstration

Figure 36.22 reflects the behavior of the PCES Capstone Demonstration from a resource management point of view over three hours of an execution. Figure 36.23 zooms in on the first hour and a half of network (Figure 36.23a) and CPU usage (Figure 36.23b) for each participant in the demonstration. The reallocation of resources upon role changes is evident from the graphs. Another important observation from the graphs is the effective control of network resource usage (Figure 36.23a) and relative lack of jitter versus the higher jitter in the CPU usage (Figure 36.23b). This is due to the fact that in the capstone demonstration, we used DSCPs and DiffServ enabled routers for network control. However, stability problems in the underlying Timesys Linux RT operating system caused us to omit the Timesys Linux RT CPU control capability for the final capstone demonstration (although we used it for shorter dry runs) and revert to using the more limited Linux OS priorities instead. Linux is not a real-time OS and, therefore, did not provide the degree of control that a real-time OS does and resulted in higher relative jitter in the Capstone Demonstration. We still managed to maintain higher priorities for the critical tasks. However, in

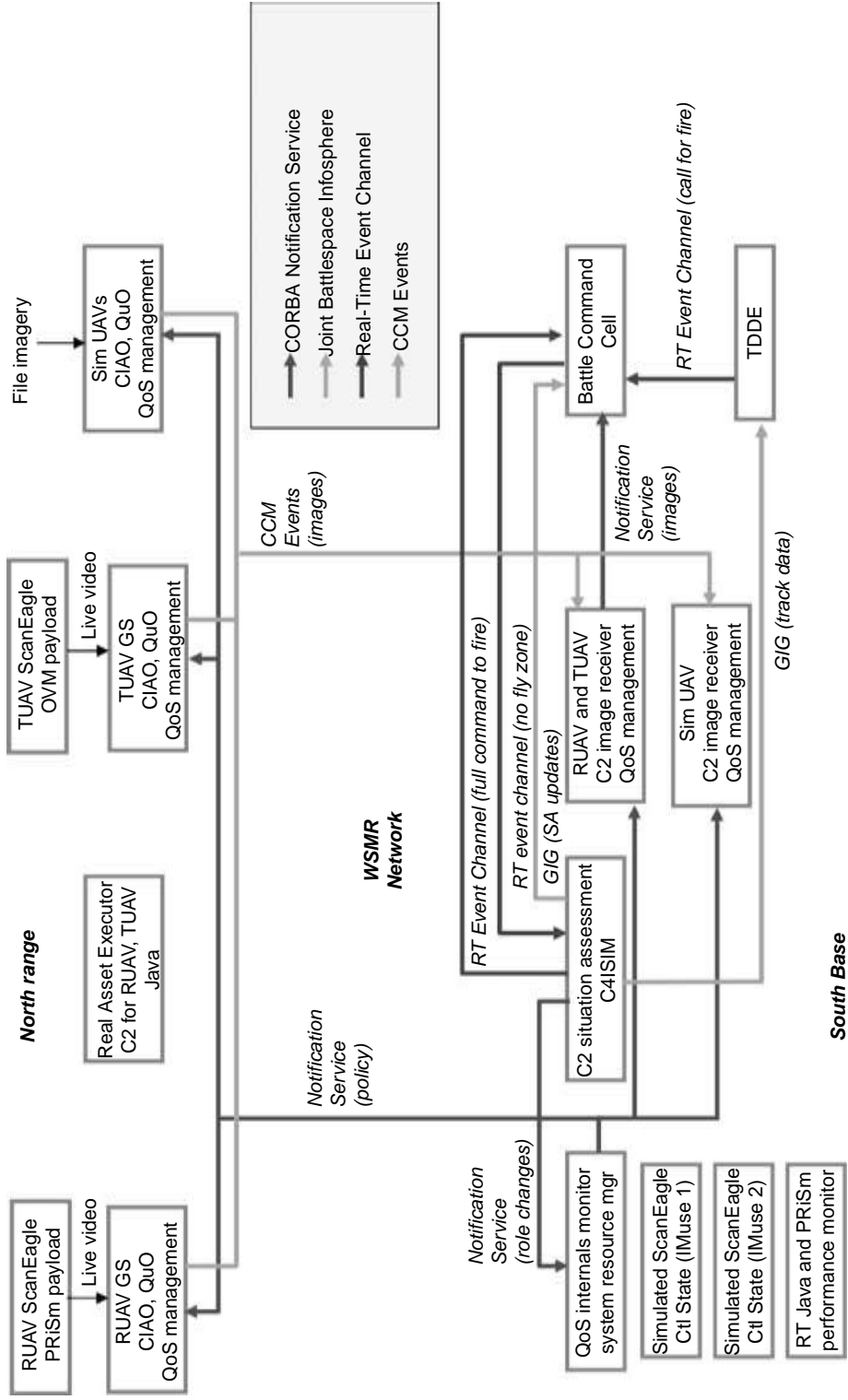


FIGURE 36.21 PCES composed subsystems utilizing middleware services.

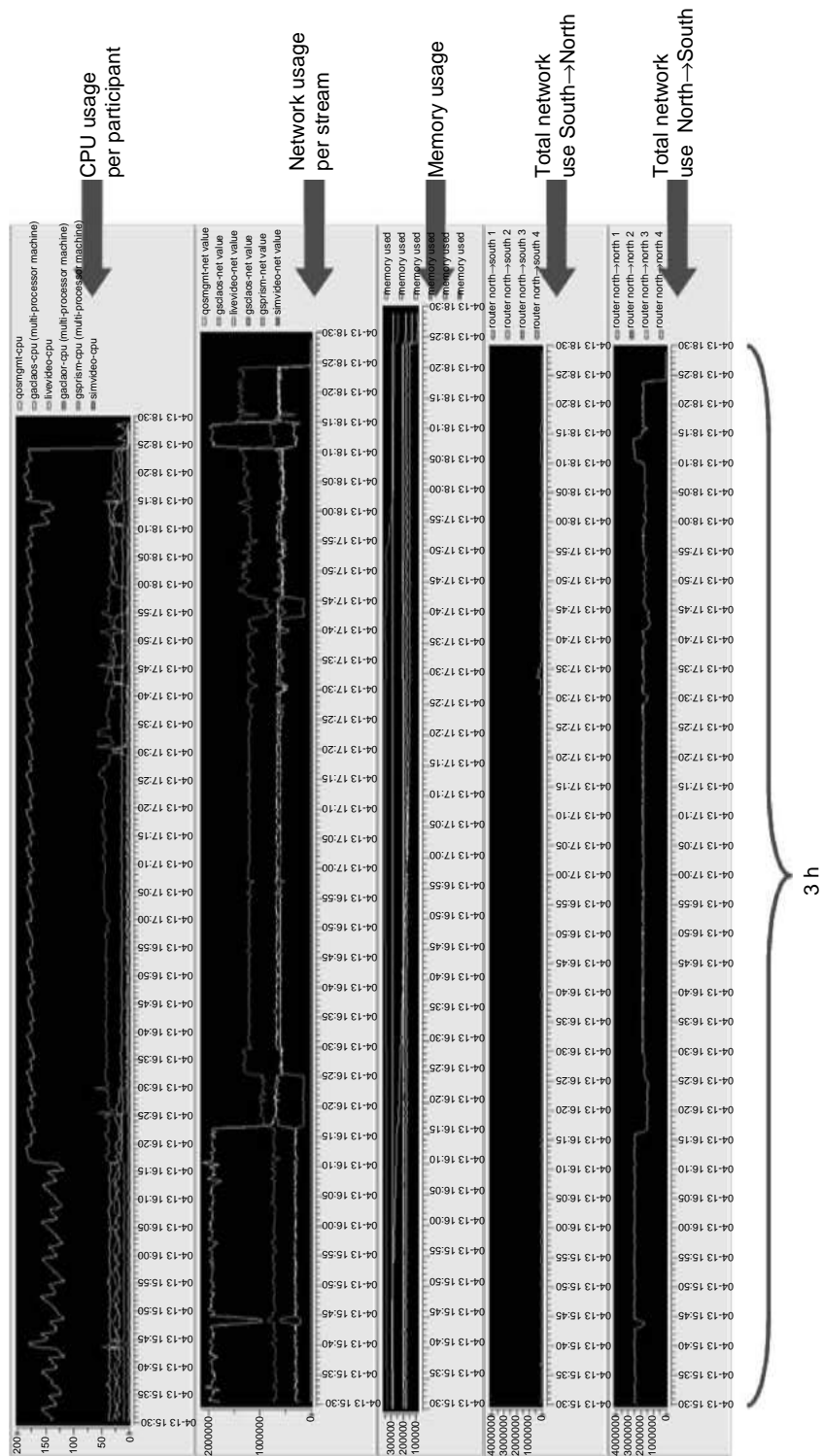


FIGURE 36.22 Resource management use during capstone execution.

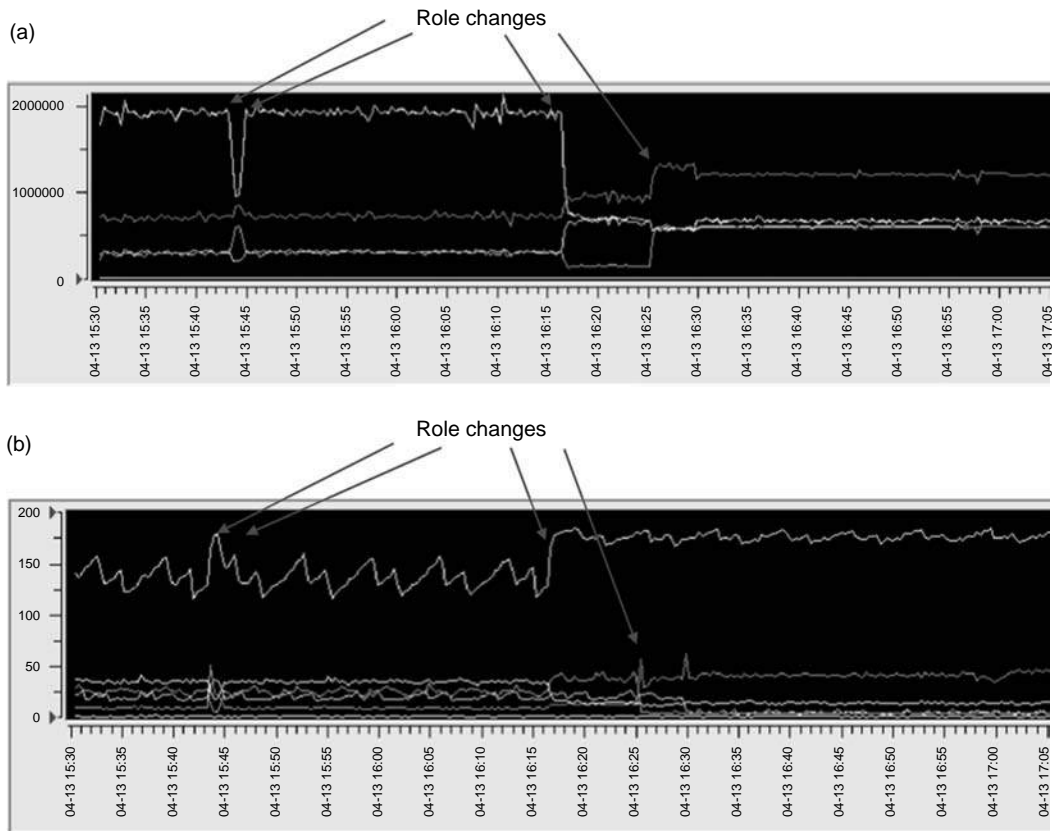


FIGURE 36.23 View of reallocation of resources due to role changes. (a) Network usage per stream. (b) CPU usage per participant.

future versions, it is obviously preferable to use a real-time OS, which means resolving the stability issues with Timesys Linux RT or replacing it with another real-time OS.

36.5 Conclusions

As the complexity and capabilities of DRE systems increase, driven both by increased networking of independent embedded systems and by the need to increase their capabilities and scope for increased usage, the ability to construct them in a manner which maintains predictable, reliable QoS must keep pace. Gone are the days when resource admission control at a single point sufficed to be called QoS management. Today's DRE systems are dynamic interoperating systems of systems, with the need for QoS management that is

- Multilayered and scalable, basing low-level allocations of resources and control of behavior on high-level mission goals.
- End-to-end, matching distinct end-user QoS needs with the integration of the various production and dissemination elements of the information system.
- Aggregate, mediating, and managing the conflicting QoS needs across competing and cooperating applications, systems, and users.
- Dynamic, adjusting QoS provision on the fly as situations, conditions, and needs change.

With enough time and budget, intelligent engineers could likely produce a system with QoS management that fulfills each of these characteristics for a specific system. However, our vision is more ambitious. We

need to develop QoS management not for a single instance of a specific system, but to develop commonly accepted and used tools and techniques that enable QoS management to be developed in many systems repeatably, so that they are well designed, reusable, and maintainable. The solution is based in middleware, because QoS management largely falls in that space where the applications interact with the platforms and environments in which they are deployed, and can more easily be made part of a common infrastructure. Current solutions additionally led to extended software engineering practices that support these goals:

- AOP and separation of concerns, to support the separation of programming application code (which is the purview of a domain expert) and QoS code (which is the purview of a systems engineer).
- Components and composition, to support the encapsulation of QoS management code into reusable bundles and the construction of new systems by composing, specializing, and configuring existing components.
- Middleware services, to support the loose integration of whole subsystems, enabling large DRE systems of systems to be constructed from existing DRE systems.

The work described in this chapter has already gone beyond the laboratory setting. The three case studies described in this chapter involved actual military and industrial organizations, systems, problems, and live flight demonstrations. Applying the QoS management middleware research to these case studies has served to validate the research results and encourage its use as a base for further increasing the capabilities while reducing the risk associated with developing the complex DRE systems emerging in real-world domains.

However, there is still a long way to go before the concepts developed and described here can become standard operating procedure or normal best practice for constructing DRE systems. Among the current issues toward that end are the following:

- Easy to use tools to automate the design process.
- Expanded shelves of reusable QoS mechanisms and management components.
- Conflict identification and resolution across various QoS dimensions.
- Approaches and tools for verifying and certifying correct operation.

References

1. A. Burns, A. Wellings, *Real-Time Systems and Programming Languages*, 3rd Edition. Addison Wesley Longmain, 2001.
2. CADML (Component Assembly and Deployment Modeling Language), <http://www.dre.vanderbilt.edu/lu/CADML/>.
3. Component Integrated ACE ORB (CIAO), <http://www.cs.wustl.edu/~schmidt/CIAO.html>.
4. CORBA Component Model, V3.0 formal specification, <http://www.omg.org/technology/documents/formal/components.htm>.
5. Cougaar—Component Model Extended from Java Bean Component Model, <http://www.cougaar.org/>.
6. B. Dasarathy, S. Gadgil, R. Vaidyanathan, K. Parmeswaran, B. Coan, M. Conarty, V. Bhanot, Network QoS Assurance in a Multi-Layer Adaptive Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework, in *Proc. of the Real-time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, March 2005.
7. G. Duzan, J. Loyall, R. Schantz, R. Shapiro, J. Zinky, Building Adaptive Distributed Applications with Middleware and Aspects, in *Proc. International Conference on Aspect-Oriented Software Development (AOSD '04)*, Lancaster, UK, March 22–26, 2004.
8. E. Eide, T. Stack, J. Regehr, J. Lepreau, Dynamic CPU Management for Real-Time, Middleware-Based Systems, in *Proc. of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, Toronto, ON, May 2004.
9. C. Gill, D. Levine, D. Schmidt, The Design and Performance of a Real-Time CORBA Scheduling Service, in *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, Kluwer, 2001.

10. G. Hardin, The Tragedy of the Commons, *Science*, 162: 1243–1248, 1968.
11. J. Huang, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao, R. Bettati, RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications, in *Proc. of Workshop on Middleware for Distributed Real-Time Systems, RTSS-97*, San Francisco, California, 1997.
12. IETF, An Architecture for Differentiated Services, <http://www.ietf.org/rfc/rfc2475.txt>.
13. M.B. Jones, D. Rosu, M.-C. Rosu, CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, in *Proc. of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pp. 198–211, October 1997.
14. D. Karr, C. Rodrigues, J. Loyall, R. Schantz, Y. Krishnamurthy, I. Pyarali, D. Schmidt, Application of the QuO Quality-of-Service Framework to a Distributed Video Application, in *Proc. of the International Symposium on Distributed Objects and Applications*, Rome, Italy, September 18–20, 2001.
15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, J. Irwin, Aspect-Oriented Programming, in *Proc. ECOOP'97 Object-Oriented Programming*, 11th European Conference, LNCS 1241, pp. 220–242, 1997.
16. J. Loyall, Emerging Trends in Adaptive Middleware and its Application to Distributed Real-time Embedded Systems, in *Proc. of the Third International Conference on Embedded Software (EMSOFT 2003)*, Philadelphia, Pennsylvania, October 13–15, 2003.
17. J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, K. Anderson, QoS Aspect Languages and Their Runtime Integration, *Lecture Notes in Computer Science*, p. 1511, Springer, in *Proc. of the Fourth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, May, 28–30, 1998.
18. J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications, in *Proc. of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, AZ, April 16–19, 2001.
19. J. Loyall, R. Schantz, D. Corman, J. Paunicka, S. Fernandez, A Distributed Real-time Embedded Application for Surveillance, Detection, and Tracking of Time Critical Targets, in *Proc. of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, March 2005.
20. J. Loyall, R. Schantz, J. Zinky, D. Bakken, Specifying and Measuring Quality of Service in Distributed Object Systems, in *Proc. of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 98)*, 1998.
21. J. Loyall, J. Ye, S. Neema, N. Mahadevan, Model-Based Design of End-to-End Quality of Service in a Multi-UAV Surveillance and Target Tracking Application, in *Proc. of the 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES)*, Toronto, Canada, May 25, 2004.
22. J. Loyall, J. Ye, R. Shapiro, S. Neema, N. Mahadevan, S. Abdelwahed, M. Koets, D. Varner, A Case Study in Applying QoS Adaptation and Model-Based Design to the Design-Time Optimization of Signal Analyzer Applications, in *Proc. of the Military Communications Conference (MILCOM)*, Monterey, California, October 31–November 3, 2004.
23. P. Manghwani, J. Loyall, P. Sharma, M. Gillen, J. Ye, End-to-End Quality of Service Management for Distributed Real-time Embedded Applications, in *Proc. of the Thirteenth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2005)*, Denver, Colorado, April 4–5, 2005.
24. Mico Is CORBA, The MICO project page, <http://www.mico.org>, <http://www.fpx.de/MicoCCM/>
25. Object Management Group, Fault Tolerant CORBA Specification, OMG Document orbos/99-12-08, December 1999.
26. Object Management Group, Minimum CORBA—Joint Revised Submission, OMG Document orbos/98-08-04, August 1998.
27. Object Management Group, Real-Time CORBA 2.0: Dynamic Scheduling Specification, OMG Final Adopted Specification, September 2001, <http://cgi.omg.org/docs/ptc/01-08-34.pdf>.

28. W. Roll, Towards Model-Based and CCM-Based Applications for Real-time Systems, in *Proc. 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, Hakodate, Hokkaido, Japan, 75–82, 2003.
29. R. Schantz, J. Loyall, C. Rodrigues, D. Schmidt, Y. Krishnamurthy, I. Pyarali, Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware, in *Proc. of the ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.
30. R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, J. Loyall, An Object-level Gateway Supporting Integrated-Property Quality of Service, in *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 99)*, May 1999.
31. P. Sharma, J. Loyall, R. Schantz, J. Ye, P. Manghwani, M. Gillen, G. Heineman, Using Composition of QoS Components to Provide Dynamic, End-to-End QoS in Distributed Embedded Applications—A Middleware Approach, *IEEE Internet Computing*, 10(3): 16–23, 2006.
32. D. Sharp, Reducing Avionics Software Cost Through Component Based Product Line Development, in *Proc. of the 10th Annual Software Technology Conference*, 1998.
33. R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, QuO's Runtime Support for Quality of Service in Distributed Objects, in *Proc. of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.
34. N. Wang, C. Gill, D. Schmidt, A. Gokhale, B. Natarajan, J. Loyall, R. Schantz, C. Rodrigues, QoS-Enabled Middleware, in *Middleware for Communications*, Qusay H. Mahmoud ed., Wiley, New York, 2003.
35. W. Wilson, Applying Layering Principles to Legacy Systems: Link 16 as a Case Study, in *Proc. of the Military Communications Conference (MILCOM)*, 526–531, 2001.
36. J. Ye, J. Loyall, R. Shapiro, R. Schantz, S. Neema, S. Abdelwahed, N. Mahadevan, M. Koets, D. Varner, A Model-Based Approach to Designing QoS Adaptive Applications, in *Proc. of the 25th IEEE International Real-Time Systems Symposium*, Lisbon, Portugal, December 5–8, 2004.
37. L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala, RSVP: A New Resource ReSerVation Protocol, in *IEEE Network*, September 1993.

37

Embedding Mobility in Multimedia Systems and Applications

37.1	Introduction	37-1
37.2	Challenges for Mobile Computing with Multimedia	37-2
	Portability • Wireless Communication • Mobility	
37.3	System-Layer Approaches	37-4
	Embedded Operating Systems and Memory Management • Mobile Multimedia Systems in a Grid Infrastructure • Quality of Service in Wireless Communication • Mobile Middleware and File Systems	
37.4	Application-Layer Approaches	37-9
	Energy Saving • QoS in Multimedia Applications	
37.5	Conclusions	37-14

Heonshik Shin
Seoul National University

37.1 Introduction

Technological advances have progressively removed the spatial limitations of the computing environment, giving rise to a new paradigm called mobile computing [1]. We can now connect to the Internet while we are on the move, using pocket-sized, battery-powered embedded devices, such as PDAs (personal digital assistants) and cellular phones, which communicate over a wireless channel. The applications of computing devices have also been changing in response to ever-improving hardware and software technologies. Today, we routinely utilize multimedia-based services instead of text-based ones, even hand-held devices. We are now moving into a new era of computing.

The burgeoning market for information, entertainment, and other content-rich services follows the rising popularity of mobile devices with high-quality multimedia capabilities [2]. But these services must adapt to a continuously changing computing environment while meeting the diverse requirements of individual users. In this case, we need to consider additional real-time and embedded properties of multimedia applications such as the deadline for each delivered packet and battery life. To address these challenges, we have developed new techniques for the implementation of networked multimedia applications across the application, system, and hardware layers of the mobile systems hierarchy.

In this chapter, we intend to present how multimedia applications can be embedded into mobile devices and how they can be optimized to support mobility. We will discuss aspects of the system layer that is implemented as software components in the system middleware, and review the facilities available to

manage mobility, connectivity, locality, and multimedia processing. We will also introduce application-layer approaches to reduce the energy requirements of mobile devices and to make the best use of the resources of the wireless mobile computing environment, such as network bandwidth, by means of improved scheduling and error correction techniques. We will illustrate these approaches with case studies, focusing on multimedia applications.

37.2 Challenges for Mobile Computing with Multimedia

The new environment we have just described brings new challenges, which can be classified into three categories: portability, wireless communication, and mobility, as shown in Figure 37.1 [1]. Each of these topics will be briefly reviewed in this section.

37.2.1 Portability

We need to reduce the size of computing devices and their energy consumption, so that they are portable and durable for everyday use with the limited power available from a battery. While the Moore's law predicts that computing power continues to increase, the energy constraint demands that we sacrifice the performance in portable devices in return for a longer operation time. In general, the focus of recent research has been on topics such as low-power system design, efficient usage of storage space, and support

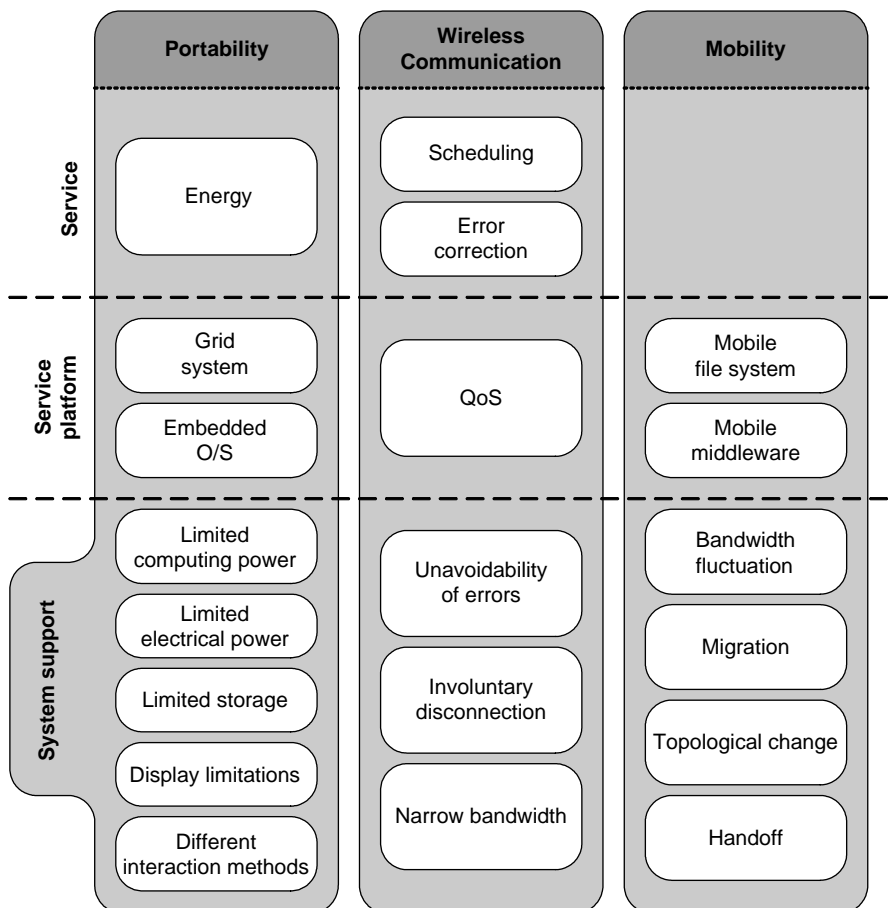


FIGURE 37.1 Novel attributes of mobile multimedia systems.

for various I/O devices [3–5]. In every case, the luxury of providing extra functionalities and redundant components is not allowed. This means that we need more sophisticated approaches to characterizing resource allocation than those taken in the traditional design process.

Almost every part of a mobile system is affected by the issue of portability. For example, it might be better to select a lower-performance CPU if it uses less power, or the fidelity of the display may have to be sacrificed to save both energy and computing resources. The most significant design considerations for portable systems can be summarized as follows:

- *Limited computing power:* The computing power of mobile devices is comparatively low, so system must be carefully tuned for computing tasks to run to completion successfully.
- *Limited electrical power:* The major energy source for a mobile device is a battery of small capacity and low voltage. To increase the time between recharges of batteries, the power consumption of each component of both hardware and software must be reduced as much as possible.
- *Limited storage:* The sizes of both the memory and the external storage are smaller than in ordinary computing devices. Storage limitations should be considered at the design stage of mobile computing applications.
- *Display limitations:* The size and the resolution of a mobile display are necessarily much less than those of a typical desktop monitor. It is essential that such a small display area is utilized effectively.
- *Different interaction methods:* Interactions with a mobile device are usually provided by means of a touch-screen, touch-pad, or microphone, which contrast with the keyboard and mouse used for desktop computers. Especially, we need to design user interactions for mobile devices to be user-friendly.

37.2.2 Wireless Communication

Recent years have witnessed the ubiquity of wireless communication for digital voice and data in various forms, such as digital cellular phones, digital cordless phones, and wireless local networks. New generation of wireless technology now focuses on digital multimedia systems and applications that involve, for example, text, images, audio, video, and sensor. To achieve the goal of pervasiveness of digital media, we need to deal with the technical issues that stem from the very characteristics of wireless systems as follows [6]:

- *Unavoidability of errors:* A wireless channel is intrinsically prone to errors. Thus, an adequate method of error recovery must be provided to maintain the usability of transmitted data.
- *Involuntary disconnection:* Frequent disconnections are expected in wireless communication. The connection status must be managed to minimize the effect of disconnections.
- *Narrow bandwidth:* Wireless communication provides a comparatively narrow bandwidth. Lower bit-rate must be taken into account in designing mobile applications.
- *Bandwidth fluctuation:* Variations in bandwidth are another characteristic of the wireless channel. It is not possible to guarantee bandwidth to any mobile devices, which means that the quality of service (QoS) requires special consideration.

37.2.3 Mobility

Mobility issues are related to the ability to change the location while remaining connected to a network. Today's subscribers want to move about, both for work and leisure, while continuing to access a network with their mobile devices. But the network environment that users experience will change dynamically as they move around, and these dynamics must be considered if an adequate service is to be provided. Mobile environments should also provide an efficient way of utilizing computing resources located at nearby computers.

To offer seamless connectivity to users, it is necessary to manage topological information about the location of their equipment on the network [7,8]. Mobility considerations include:

- *Migration:* Mobile nodes change their network connection as they move about. They need to be correctly located wherever they move to.

- *Topological change*: The network topology varies with the movement of each node. For mobile *ad hoc* networks in particular, it is necessary to provide ways of networking which will permit successful data transmission despite significant topological changes.
- *Handoff*: To achieve continuous network connectivity, a mobile device should maintain its connection with a current network until a firm connection to a new network is established. This handoff operation is the focus of mobility management.

37.3 System-Layer Approaches

In this section, we will present a scenario to describe system-layer approaches to portability, wireless communication, and mobility: Imagine a user walking along, watching the news on her PDA. After a while, she invokes video messaging to discuss an interesting news item with her friend. In this simple scenario, the user must be able to access multimedia content in real time. Mobile devices, however, have a limited ability to accept the incoming content, owing to the system limitations discussed above. To deal with such restrictions, the middleware selects an appropriate level of fidelity for the content, and makes the best of the limited resources of the mobile device. In doing so, it may, for example, exploit the distinctive characteristics of multimedia data by changing parameters such as resolution, streaming data rate, and encoding method.

Performance analysis based on the relation between resource consumption and QoS can help a mobile system to manage multimedia data efficiently. In addition to saving its resources by performance tuning, the mobile system can utilize surrogate computing facilities with more powerful resources that are attached to the wired network. Different approaches to the provision of adequate computing power are detailed in Sections 37.3.1 and 37.3.2.

One untoward characteristic of mobile computing is frequent disconnection due to the intrinsic unreliability of the wireless link. There are many ways to mitigate the effects of limited bandwidth and intermittent disconnection, based on packet scheduling and error correction techniques. The efficient use of packet scheduling optimizes the system capacity and improves the data transmission rate, while error correction recovers information corrupted by the wireless link. Section 37.3.3 describes the techniques for managing wireless communication.

In the above scenario of a user and her PDA, the services she requests will now use the network connection to deliver multimedia data. To give her access to all available services, regardless of their location and her movements, her mobile device needs to be aware of its own context, including its location, mobility pattern, the network type, and data rates. Middleware can also support transparent physical migration by means of seamless software migration techniques. An overall system architecture for mobility is presented in Section 37.3.4.

37.3.1 Embedded Operating Systems and Memory Management

An operating system (OS) is essential to simplify the design of complicated embedded systems, including portable devices that run multimedia applications. For example, MPEG software requires the services provided by the OS for activities such as receiving bitstreams, scheduling decoder processes, and synchronizing video and audio.

The designers of embedded systems have largely overlooked OS behavior and its impact on performance, because it is difficult to obtain a meaningful trace of OS activity. In fact, the effect of the OS on the overall performance of a system is critical. One of the serious problems is the poor memory performance of OSs, caused by the intermittent, exception-driven invocation of OS routines, which tends to dominate overall performance in most applications as the speed gap between the CPU and the memory widens. In particular, video encoding and decoding are computationally intensive, and they also demand exceptional memory bandwidth.

Analyzing the memory performance of applications in embedded systems is more difficult than in general-purpose systems [9]. Small changes to the design parameters of a system may have a significant

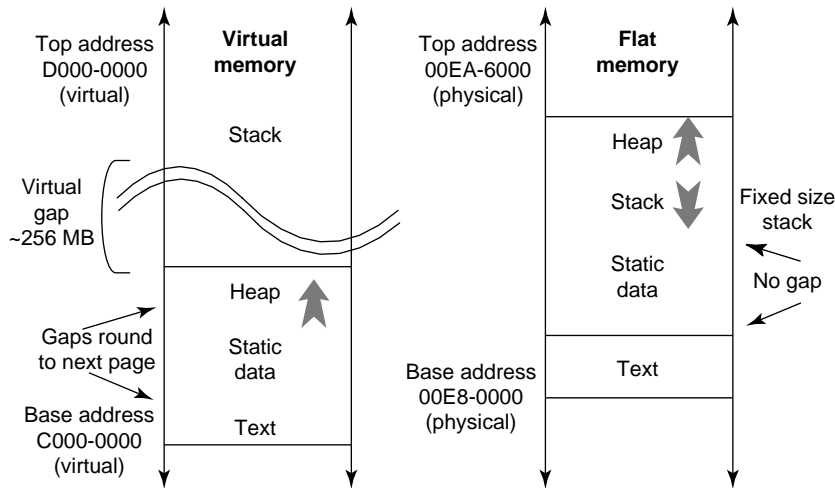


FIGURE 37.2 Memory map of virtual memory and flat memory.

effect on its performance and cost. To quantify this effect, the memory system behavior of an application running on an embedded OS needs to be evaluated, by varying both the hardware design parameters and the selection of software components.

Most modern computer systems now support virtual memory, which provides a distinct virtual address space for each process and also offers hardware-assisted memory protection for multiprogramming. A virtual memory system is maintained by the memory management subsystem of the OS, and requires hardware support from a memory management unit (MMU). The MMU helps the OS to control memory use at runtime, which improves functionality, productivity, and maintainability. However, the use of an MMU increases the execution time and overhead of application software, such as MPEG encoders and decoders, because resources are required to manage the virtual memory system as well as the MMU hardware, which also requires electrical power.

A simple but effective approach to reduce the translation lookaside buffer (TLB) overhead of virtual memory system is to modify the memory management policy so as to share the virtual address space between the kernel and the application, by running a program as a kernel-mode thread [10]. By sharing address space, the burden of memory management is reduced. For instance, separate page tables for kernel and application are merged into one page table.

Because many small embedded systems have severe restrictions on their power consumption or manufacturing cost, it is often infeasible to include an MMU. The flat memory policy [11], as depicted in Figure 37.2, shares physical addresses between the kernel and the application. This policy does not need an MMU and has the advantage that it does not need to handle per-process page tables (or TLB misses) and the associated protection required by the virtual memory system. Application programmers are still able to allocate nonoverlapping memory region to the application, but implicit restrictions are placed on memory usage. For example, the stack size must be defined carefully to avoid an overflow.

The cache performance of an MPEG decoder is often poor because typical data sets and working data sets are so huge that cache architectures for embedded systems cannot handle them. Various techniques to improve cache performance have therefore been widely adopted. For example, both software and hardware data prefetching are often employed in multimedia applications.

37.3.2 Mobile Multimedia Systems in a Grid Infrastructure

In the context of distributed supercomputing and Internet-based parallel computing, the grid paradigm can provide an infrastructure for collaborative engineering, data exploration, and other advanced value-added services [12]. In particular, users' needs for real-time streaming regardless of their locations have

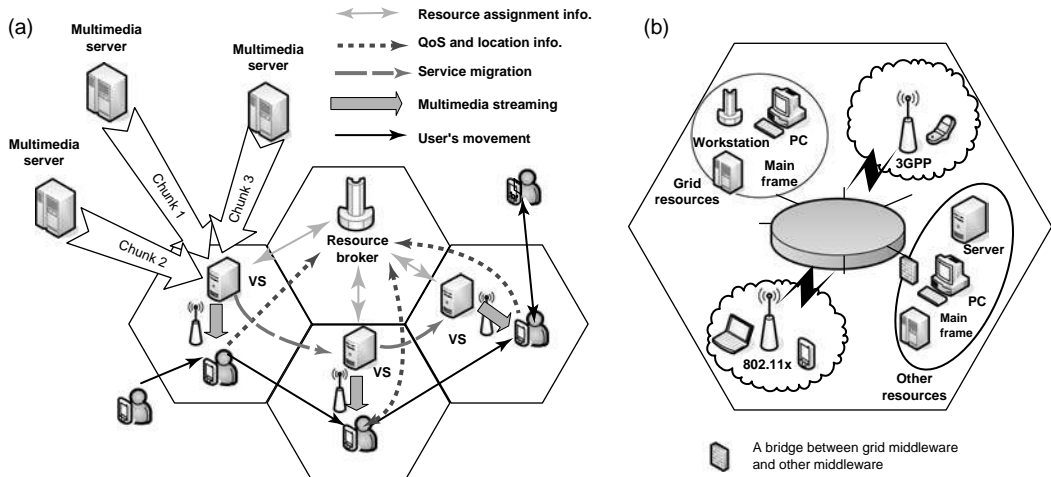


FIGURE 37.3 Mobile multimedia systems in a grid infrastructure. (a) System application scenario in a grid system, and (b) possible components within a cell.

motivated the design of the techniques which enable mobile systems to make the best of the intermittently available computing, storage, and communication resources of a grid system and achieve real-time delivery of multimedia services in a mobile environment. Two main approaches are widely accepted: the use of connection-oriented services combined with stream buffering in the service path, and transcoding of the stream from a server at a local proxy that can customize the multimedia content based on required QoS levels and mobile device capabilities.

To utilize locally available grid resources to customize multimedia applications based on the requirements and limitations of mobile devices, an effective resource allocation policy must be in place to address the trade-off between performance and quality. Making the best use of the local resources of each mobile device reduces the overall network traffic (and thus increases the average performance of all mobile devices), but misplanned local optimization tends to introduce frequent switches in the multimedia stream which may result in increased jitter and reduced QoS.

Several complications arise in the effective utilization of grid resources for mobile multimedia services, among which the most significant is the intermittent availability of grid resources in heterogeneous systems. Figure 37.3a shows a typical scenario. A mobile device (e.g., a PDA or laptop) runs a streaming multimedia application (e.g., an MPEG player) while traversing a cellular network and receiving video streams from various grid resource providers. One such provider is a grid volunteer server (VS), which is a machine that participates in the grid by supplying resources when they are not otherwise being used. A VS can be a wired workstation, server, or cluster, which provides high-capacity storage for multimedia data, and CPU power for multimedia transcoding and decompression. A VS can also be used to perform adaptive network transmissions [13] that facilitate effective dynamic power management of network interface cards. Figure 37.3b illustrates the components within a cell that may be involved in a multimedia session, including base stations, grid resources, and middleware.

The middleware components that can assist in discovering intermittently available resources for mobile multimedia applications are illustrated in Figure 37.4. The two key components are the broker that performs admission control for incoming requests and manages their rescheduling, and the directory service module that stores information about resource availability at the VSs and clients. When a mobile user sends a request, the broker passes it to the request scheduler, which executes a resource discovery algorithm based on the resource and network information retrieved from the directory service. If no scheduling solutions exist, the broker rejects the request; otherwise, the resource reservation module reserves the resource and the placement management unit replicates the data, which could be, for instance,

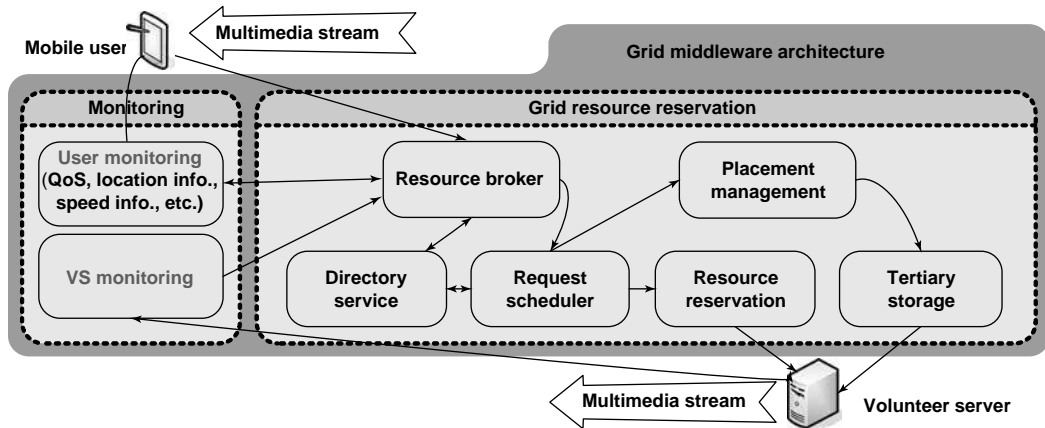


FIGURE 37.4 A grid middleware service architecture.

video segments, copying it from tertiary storage to the selected VSs. The directory service is then updated with new allocation information. If the availability of a VS changes, many preassigned requests may be invalidated. The VS monitoring module detects dynamic changes in VS availability, and informs the broker. Subsequently, the updated configuration is then registered with the directory service. The user monitoring module keeps track of the mobile clients, reporting their movements to the broker so that the necessary rescheduling processes will be triggered.

These ideas are incorporated into GRAMS (Grid Resource Allocation for Mobile Services) [14], which is a system that runs on a mobile device and allows it to find voluntary services. GRAMS not only performs a load-based adaptive grid-server selection with the aim of satisfying QoS requirements, but also it adapts to dynamic changes in user mobility and grid-server availability. There is also a power-aware version of GRAMS [15].

37.3.3 Quality of Service in Wireless Communication

A mobile user on a wireless channel can experience variations in multipath fading, path loss from distance attenuation, shadowing by obstacles, and interference from other users. Thus, the air channel condition of a user fluctuates over time and has a much higher error-rate than on a wired link.

In the case of unicast services, 3G wireless networks hold the transmission power constant and use adaptive modulation and coding (AMC) as a method of link adaptation instead of power control to improve the system capacity and peak data rate. A mobile device repeatedly measures the signal-to-interference and noise ratio (SINR) [6] of the received signal and reports it back to the base station. Changes of data rate are made in response to the channel status information by selecting an appropriate modulation and coding scheme to the information at the base station as shown in Figure 37.5. In the case of broadcast services, however, the network should serve multiple users simultaneously; so it cannot adapt to the channel condition of individual users. It is desirable to provide satisfactory services for the users with the poorest radio quality.

For error reduction, there are two well-established types of error correction used in data transmission: automatic repeat request (ARQ) and forward error correction (FEC) [16,17]. The ARQ scheme is based on the retransmission of missing data packets, triggered by ACK/NACK messages from the receiver. This approach is generally used within unicast protocols because it is very effective, simple to implement, and does not require any modification of the primary data stream. Nonetheless, ARQ does require a feedback channel and time to recover missing packets (which implies a full round-trip time implementation). These overheads may be significant over a wireless link or for real-time applications, such as a teleconferencing system that supports full duplex video and audio. Additionally, ARQ scales poorly to multicast protocols and a large number of recipients, because the sender is likely to have to deal with very large numbers

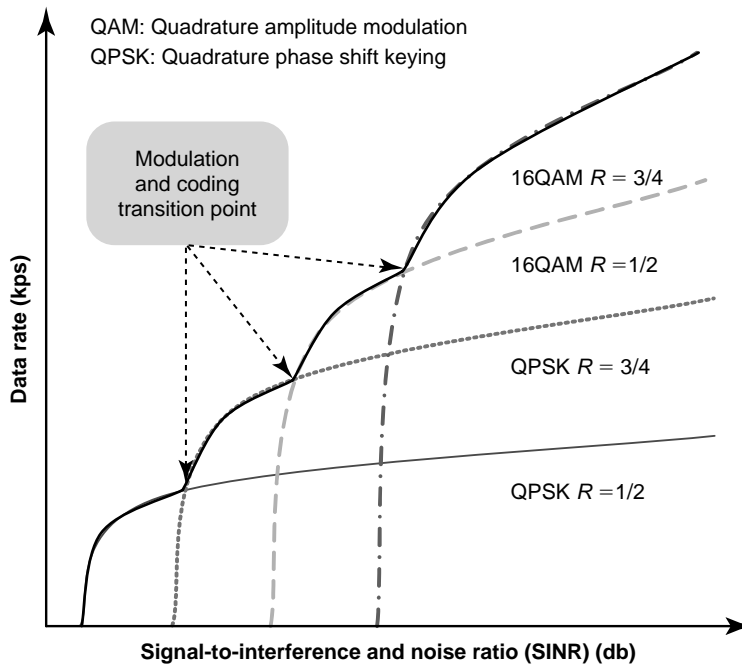


FIGURE 37.5 Modulation and coding selection in AMC by the base station.

of ACKs or NACKs, and will also have to accept additional feedback from the receivers to decide which packets to retransmit.

In contrast to ARQ, FEC can tolerate some errors. By the use of code with an appropriate level of redundancy, the level of packet loss at a receiver can be made arbitrarily small, at the price of sending extra data. Furthermore, FEC-based multicast protocols scale much better than ARQ-based protocols, since uncorrelated packet losses are handled at the receiver and make no demands on the feedback channel.

ARQ and FEC can be used in combination, with ARQ implemented on top of FEC. FEC effectively reduces the error-rate of a noisy channel and the randomness of the errors, so that they occur in relatively long bursts, which can then be corrected by requesting retransmissions. FEC is most often implemented at a low level in a network, but rarely as part of an Internet protocol.

37.3.4 Mobile Middleware and File Systems

The apparently random movement of mobile users has motivated the development of middleware to support the management of location, mobility, and disconnections. This middleware must (a) provide the context for mobile applications, (b) detect location changes, (c) associate location changes with context, and (d) determine the effects on applications. Recent research has looked at implementing these functionalities through mobile agents [18], service binding [19], and context-aware processing [20,21].

The services available in one region may become unavailable or irrelevant in another region, making it difficult to provide smooth and seamless mobile applications. As a user moves into a new area, the old service may no longer be relevant, but a different provider may offer an equivalent that relates to the new area. To allow applications to make seamless transitions across such boundaries, it is necessary to keep track of the coverage of the services offered by different providers, to sample the user's location periodically, and hence to determine whether the current services are still valid. Most of the middleware to support user mobility also facilitates the migration of software components or service bindings, as shown in Figure 37.6.

Lime [18] and EgoSpace [22] associate a change of physical location with a logical migration using mobile agents and tuple spaces. However, if a mobile device is served by a combination of Internet services

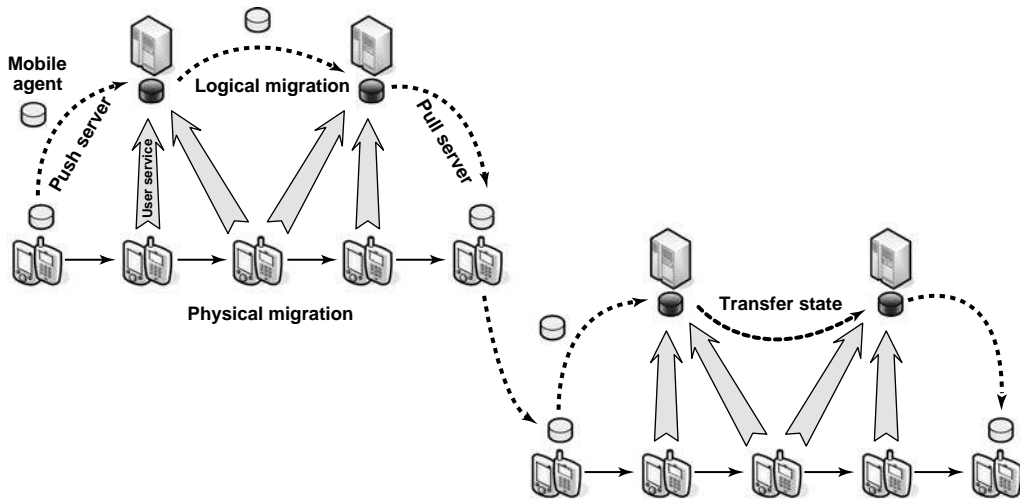


FIGURE 37.6 Middleware to support user mobility.

it will also be necessary to adapt the service binding to maintain mobility. The Atlas [19], MOCA [23], and DANSE [24] systems are oriented toward the adaptation of applications in a dynamic environment through service rebinding. Context-aware processing is also an effective way to achieve mobility, by monitoring location changes and associating them with task scheduling or the QoS control of applications [20,25]. Context-aware scheduling maximizes the overall utility of the system while satisfying the time and energy constraints of multimedia applications that are affected by factors that include location and resource availability. Middleware components can also utilize a QoS model in which applications may have several versions, each with different time and energy requirements.

File systems must also cope with frequent disconnections, although a local cache can shield a file system to some extent from the vagaries of the wireless network. Incremental updates are also effective in reducing the amount of data to be transmitted, and operation logs can be used to continue to modify files when a mobile device is disconnected. Coda [26] supports disconnected operations by caching the most recently accessed data locally. When a mobile device running Coda is disconnected, it continues to perform file operations using a local cache and saves these operations into the client modification log (CML). Once the mobile device is reconnected, these logs are sent to the server for data synchronization, or Coda can use trickle reintegration if the connection is weak, and the propagation of updates is then delayed for as long as possible. While late propagation may cause validity problems, it allows the operation logs to be optimized, because they remain accessible in the CML for a long time.

NFS/M [27] works in a similar way to Coda, operating in connected, disconnected, and reintegration phases which reflect the channel state. This technique caches data into local storage using a proxy server (PS) daemon, and a data prefetcher (DP) predicts which files will be needed next and prefetches them. Prefetching may be activated by the application or it can be triggered by the results of an analysis of the pattern of data accesses. Mobile file system (MFS) [28] compensates for unstable network bandwidth. It uses priorities that differentiate types of remote procedure calls (RPCs) to improve the response time. MFS assigns high priority to small RPCs, which would block an interactive client, and low priority to RPCs, which can be performed in the background.

37.4 Application-Layer Approaches

In this section, we intend to describe a few examples for energy saving and QoS improvement in mobile multimedia applications. Online multimedia applications require intensive computation for video

encoding and decoding. The energy requirement of video processing can be reduced by decreasing the complexity of the algorithms, or the amount of data to be processed. In mobile devices, a controlled drop in video quality is often tolerable, owing to the imperfect nature of human visual perception and the small screen size [30,31]. These approaches are presented in Section 37.4.1.

Among many issues that affect multimedia streaming services over wireless networks, unpredictable wireless channel errors cause packet loss, reducing the QoS in applications that rely on real-time traffic. This makes error correction mandatory for an efficient network multimedia service. Nevertheless, it complicates the scheduling of packets, which must both achieve fairness among flows and maximize the overall system throughput. We explore ways of improving QoS in multimedia applications in Section 37.4.2.

37.4.1 Energy Saving

In the following two case studies, we will show how to reduce the energy required for decoding a video transmitted using the widely used MPEG-4 compression standard.

37.4.1.1 Requantization

The MPEG-4 standard specifies that a frame of video is divided into basic units, called macroblocks, each of which corresponds to a 16×16 -pixel area. The MPEG-4 video decoding procedure performs variable-length decoding of an incoming bitstream and produces a macroblock, a quantization parameter (QP), and either a single motion vector or a set of motion vectors. Subsequently, each macroblock is decomposed into four blocks of luminance and two blocks of chrominance coefficients, each of size 8×8 . All of these blocks then pass through a sequence consisting of inverse quantization, inverse discrete cosine transform, and motion compensation [32].

Quantization and inverse quantization are the key data reduction processes in MPEG coding. During encoding, quantization generates a new set of coefficients by discarding the least important information from the original macroblock coefficients. The QP determines the extent of the data reduction. The decoding process reconstructs the video by inverse quantization, using the QP and the quantized coefficients, which are obtained by variable-length decoding of arriving bitstreams.

Figure 37.7 shows that both the energy consumption and the video quality are inversely proportional to the QP, where energy consumption is more sensitive to changes in the QP than video quality. This suggests that requantization using QP values, which are rather higher than the original QP, will yield far more energy saving for a minor sacrifice of video quality.

As an element of transcoding process, requantization reduces the bit-rate of a previously compressed video by applying a further quantization to each frame. The data cache miss ratio in an MPEG video

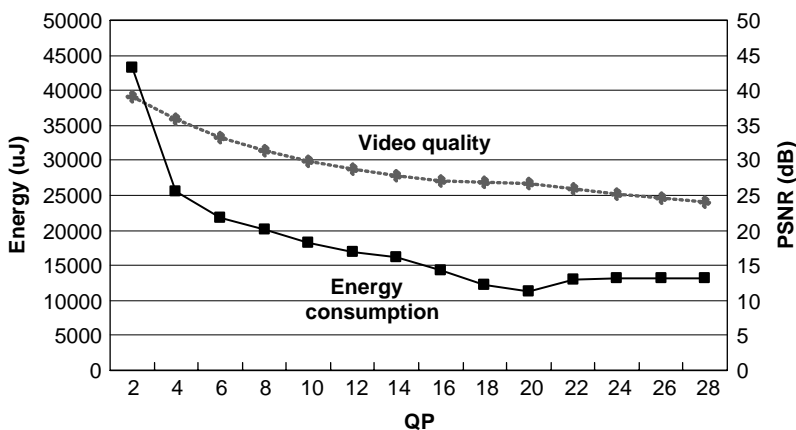


FIGURE 37.7 Video quality versus energy consumption with respect to QP.

decoder is quite high compared with other embedded applications. Therefore, it is worth reducing the amount of data involved in video decoding by changing the requantization level (Figure 37.8), especially when a mobile device is short of energy. It is, of course, necessary to maintain an acceptable video quality, although the energy saving is usually greater than the perceived degradation, depending on the amount of motion in the video [33].

37.4.1.2 Use of a Subdecoder

Current video compression algorithms enable high-quality videos to be delivered over a connection with moderate bandwidth. But the restricted size of the display on a mobile device limits the resolution of the video that is actually shown. For this reason, the downsampling process converts the original high-resolution video to a smaller video of lower resolution to fit to the mobile screen. Downsampling of the decoded video by the video processor represents a significant computing overhead, which can be reduced by integrating downsampling into the decoding process, within a decoding framework called a subdecoder (Figure 37.9). This reduces the load on the video processor, degrading the picture quality to a lesser extent, depending on the amount of motion in the video [34].

37.4.2 QoS in Multimedia Applications

The following two case studies show how packet scheduling algorithms and error correction techniques can overcome the error-prone nature of mobile wireless networks to deliver multimedia over 3G broadcast services.

37.4.2.1 Real-Time Packet Scheduling for Multimedia Services

Wireless channels are subject to unpredictable, location-dependent, and bursty errors. The packet scheduling of multimedia services must adapt dynamically to the channel status by using the scheduling algorithms, such as earliest deadline first (EDF) or greatest degradation first (GDF) [35]. EDF will maximize the

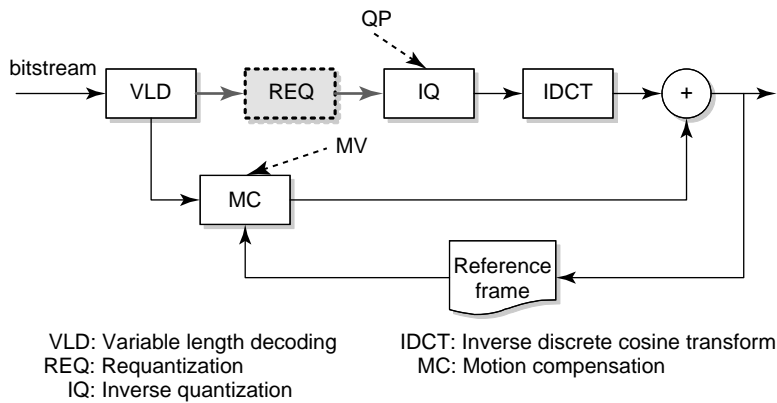


FIGURE 37.8 The process of requantization in MPEG video decoding.

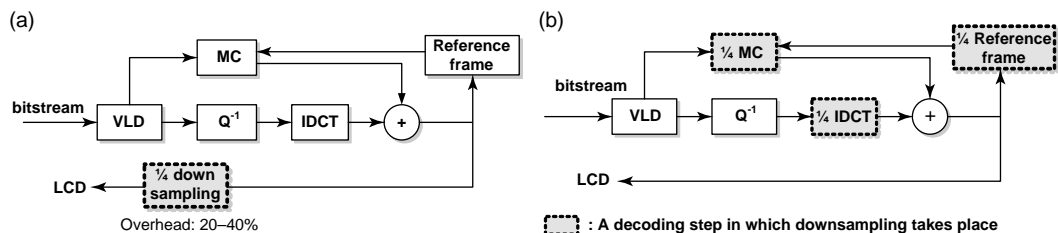


FIGURE 37.9 Comparison of downsampling with a subdecoder. (a) Typical downsampling and (b) subdecoder.

throughput when there are no channel errors because the packet with the earliest deadline is always scheduled first, despite the possible development of starvation problem. Conversely, GDF seeks to minimize the maximum degradation of any flow by scheduling first the next packet belonging to the flow which is currently most degraded.

While EDF and GDF try to meet deadlines, the weighted fair queueing (WFQ) [36] and proportional fair queueing (PFQ) [37] scheduling algorithms aim for fairness among flows. WFQ is a weighted version of fair queueing that provides services of guaranteed bandwidth by an approximation to general process sharing. Packets are stored in queues and the dispatch time of each packet is determined from its arrival time and the number of flows that are sharing its channel. Then WFQ sends the packet with the closest dispatch time. PFQ reflects temporal variations of the channel condition in its scheduling decisions. It receives the current channel condition from every receiver, computes the ratio between the current and average channel condition for every active flow, and then assigns the next time slot to the flow with the maximum data rate.

In the case of a video streaming service, a scalable video code such as MPEG-4 FGS (fine-granularity scalable) [38] can be utilized to improve quality. FGS allows the data rate of decoded videos to be dynamically changed and can provide several levels of reduced resolution. FGS encoding produces a base layer and several enhancement layers. The base layer is essential for decoding, while the enhancement layers successively improve the visual quality of the final picture. An FGS bitstream is partially decodable at any bit-rate within an allowable range and the resulting video signal has optimal quality for that bit-rate. It is clearly advantageous to provide the base layer with more protection against errors than the less important enhancement layers.

Recently, the 3GPP-2 group laid the foundation of a specification for broadcast and multicast services (BCMCS) [39,40] in a cdma2000 network, which allows multiple receivers to share the same broadcast channels. It specifies that broadcast video streams are scheduled statically and that the resulting schedule is announced to receivers. To maximize the number of receivers and to simplify implementation, the standard forbids receivers from sending acknowledgments of received packets or reporting channel status. Therefore, the coordinator of broadcast services has limited knowledge of the receivers' channel status. With the minimum acceptable SINR predetermined, modulation and coding schemes are selected to maximize the data rate.

The current static scheduling algorithm incorporated in the BCMCS specification cannot adapt to an environment in which content streams change dynamically. When a new item of content is added empty time slots must be found to service it. The timing constraints that apply to multimedia streams mean that a video frame is useful only if it arrives at a mobile before its playback time. This suggests that EDF would be a suitable real-time scheduling algorithm and significantly improve performance.

BCMCS specifies Reed–Solomon (RS) coding [41] to correct errors and this requires parity information. When the channel condition is good without eliciting any errors, many slots for parity will not serve the purpose, wasting the resource. Additionally, the capacity for error recovery in the RS scheme declines rapidly as the channel condition deteriorates. We can address these problems by adding a retransmission scheme to the EDF scheduling algorithm, using the slots saved by reducing the number of parity bits [42]. This will improve playback quality by protecting the important base-layer packets.

In an alternative approach [43] to improve channel efficiency in broadcast services, a video session is encoded in several layers and the time slots are partitioned among the layers, using different modulation and coding schemes. The layers are received with an affordable error rate that depends on the channel status. The more layers arriving at the receiver, the better quality the video shows. The modulation and coding schemes are chosen to maximize the total utility for heterogeneous receivers with varying QoS requirements. This approach also considers competing video sessions and allocates time slots among them in a way that reflects the popularity of each session. The allocation of time slots and the selection of modulation and coding schemes are performed by a modified greedy algorithm with a polynomial time complexity that permits real-time operation. Simulations have shown that the algorithm produces the near-optimal allocation that outperforms a BCMCS single-layer video broadcast with fixed modulation and coding (FMC).

37.4.2.2 Error Correction

Real-time traffic, such as voice and video streaming, is very sensitive to delay, but it can stand a certain level of packet loss. FEC is therefore preferable for multimedia broadcasts over a wireless network, and we will now describe more fully the RS FEC scheme, which we have already mentioned, and its implementation with an error control block (ECB) as shown in Figures 37.10 and 37.11, respectively.

The broadcast framing protocol in the cdma2000 1xEV-DO standard specifies how the upper-layer packets are fragmented at the access network. Whereas the broadcast security protocol specifies the encryption of framing packets, the broadcast MAC protocol defines the procedures used to transmit over the broadcast channel and specifies an additional outer code which, in conjunction with the physical-layer turbo code, forms the product code. The protocol is completed by the structure of the broadcast physical layer, which provides the broadcast channel [39,44].

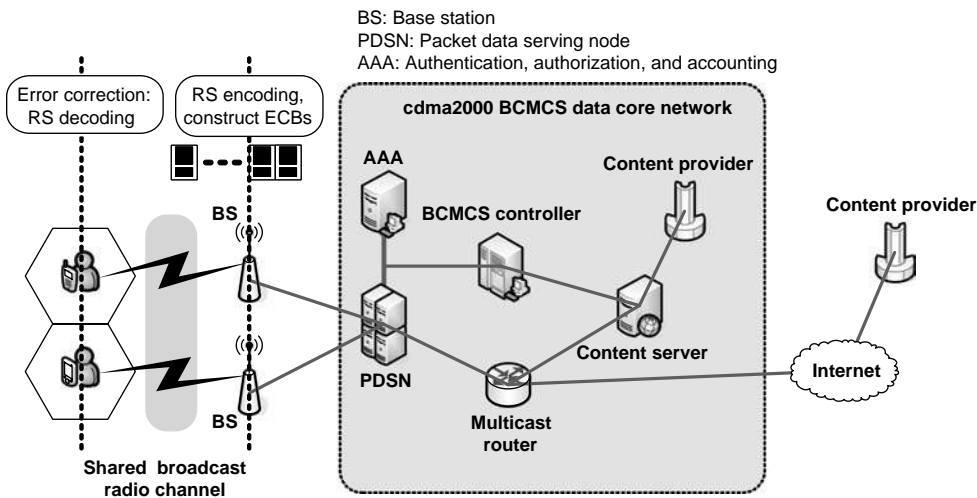


FIGURE 37.10 The BCMCS architecture.

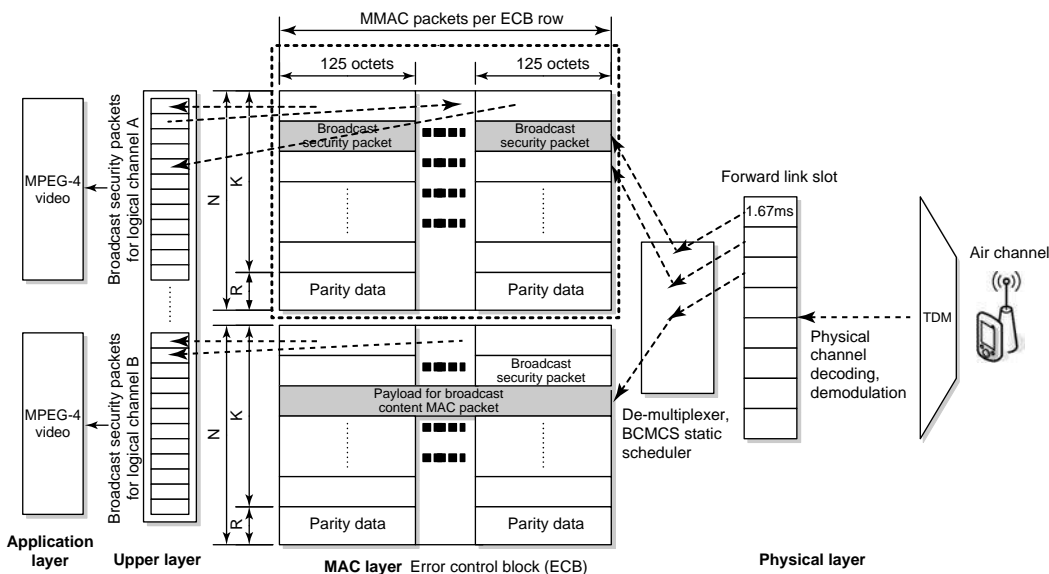


FIGURE 37.11 Error recovery structure in BCMCS.

Analysis of RS error correction shows that its performance degrades significantly when mobile devices experience bad channel conditions. This problem can be resolved by increasing the amount of parity data in the RS codes; however, that reduces the effective bandwidth. One alternative is the Hybrid-ARQ scheme [45], which uses reduced-parity codes and redeploys the saved bandwidth to retransmit base-layer packets of MPEG-4 FGS encoded video.

We must also consider video timing constraints. An MPEG-4 system permits a maximum delay of 100 ms from input to system decoder. The corruption of this timing constraint will sequentially corrupt all the timing references and cause a system reset, in which the system decoder will stop all the decoding processes, initialize every buffer and timing count, and then force the system decoder to be reinitialized [46]. The delay incurred during RS decoding, however, varies with the channel conditions and takes no account of this constraint.

Static, dynamic, and video-aware dynamic error control schemes (ECS) have been suggested [47] to address this problem. These schemes are used to bypass RS decoding if there is not enough time to correct the errors. Static ECS simply bypasses RS coding completely while dynamic ECS corrects as many errors as possible within the time available. Video-aware ECS fixes errors in a priority-driven manner which considers the characteristics of MPEG-4 FGS video [38]. In this scheme, the base-layer and enhancement-layer packets are segregated within the ECB in such a manner that all the MAC packets in each ECB subblock correspond to the same layer. This gives priority to base-layer packets when the scope for error recovery is restricted by the timing constraints of MPEG-4. Alternatively, a buffer can be used to prevent violation of the timing constraints, where the buffer size can be determined by analyzing the execution time and jitter of the RS code under various air-channel conditions.

The energy required by video decoding is another challenge to respond to the error recovery in BCMCS. We can save energy by appropriately choosing an RS coding scheme and ECB size. Less energy is required as the amount of parity information in the RS code and the ECB size decreases. This must be balanced against the concomitant reduction in error recovery capacity which will impact the QoS.

37.5 Conclusions

In this chapter, we have discussed some of the major challenges for mobile multimedia computing, affecting both the system and the application layers of the software stack.

The system layer of a mobile device is of critical importance in supporting mobile multimedia services because it provides platform for all applications. As for the OS, we have shown how the memory performance can be characterized for embedded multimedia applications. In coping with the limited resources of portable devices, a solution is suggested to selectively offload computing tasks to static servers with the help of appropriate middleware components within a grid infrastructure. Another approach presented is to balance the requirement for computational resources against the QoS.

Our discussion of application-layer techniques for mobile computing has focused on multimedia, and in particular MPEG-4 video. We examined the techniques of saving energy in multimedia applications using requantization and a subdecoder. We then investigated the effect of the wireless channel on video broadcasts, considering the merits and demerits of several scheduling algorithms for the MPEG applications. Finally, we dealt with error correction, showing how we can work with the layered structure of fine grain scalable MPEG video to deliver pictures of better quality.

The roll-out of 3G applications and services have already motivated a variety of developments in mobile computing technologies. We believe, however, that we are still awaiting a host of great ideas and implementations to be discovered and created, expanding the horizon of mobile multimedia computing.

References

1. G. Forman and J. Zahorjan, "The challenges of mobile computing," *IEEE Computer*, Vol. 27, No. 4, April 1994, pp. 38–47.

2. K. Tachikawa, "A perspective on the evolution of mobile communications," *IEEE Communications Magazine*, Vol. 41, No. 10, October 2003, pp. 66–73.
3. A. Vahdat, A. Lebeck, and C. S. Ellis, "Every Joule is precious: The case for revisiting operating system design for energy efficiency," *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000, pp. 31–36.
4. E. Pitoura and B. Bhargava, "Data consistency in intermittently connected distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 6, November 1999, pp. 896–915.
5. M. Weiser, "Some computer science issues in ubiquitous computing," *Communications of the ACM*, Vol. 36, No. 7, July 1993, pp. 75–84.
6. S. Haykin, *Communication Systems*, 4th Edition, Wiley, New York, 2001.
7. C. E. Perkins, "Mobile IP," *IEEE Communications Magazine*, Vol. 40, No. 5, May 2002, pp. 66–82.
8. H. Elaarag, "Improving TCP performance over mobile networks," *ACM Computing Surveys*, Vol. 34, No. 3, September 2002, pp. 357–374.
9. E. Park, T. Kim, and H. Shin, "Comparative analysis of aperiodic server approaches for real-time garbage collection," *Journal of Embedded Computing*, Vol. 1, No. 1, 2005, pp. 73–83.
10. C. Crowley, *Operating Systems: A Design-Oriented Approach*, Irwin Publishing, Toronto, Canada, 1997.
11. J. deBlanquier, "Supporting new hardware environment with uclinux," *Journal of Linux Technology*, Vol. 1, No. 3, 2000, pp. 20–28.
12. D. Bruneo, M. Guarnera, A. Zaia, and A. Puliafito, "A grid-based architecture for multimedia services management," *Proceedings of the First European Across Grids Conference*, Santiago de Compostela, Spain, February 2003.
13. S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "Integrated power management for video streaming to mobile handheld devices," *Proceedings of the ACM Multimedia Conference*, Berkeley, USA, November 2003.
14. Y. Huang and N. Venkatasubramanian, "Supporting mobile multimedia services with intermittently available grid resources," *Proceedings of the International Conference on High Performance Computing*, Hyderabad, India, December 2003.
15. Y. Huang, S. Mohapatra, and N. Venkatasubramanian, "An energy-efficient middleware for supporting multimedia services in mobile grid environments," *Proceedings of the International Symposium on Information Technology: Coding and Computing*, Las Vegas, USA, April 2005.
16. H. Liu, H. Ma, M. E. Zarki, and S. Gupta, "Error control schemes for networks: an overview," *Mobile Networks and Applications*, Vol. 2, No. 2, October 1997, pp. 167–182.
17. A. M. Michelson and A. H. Levesque, *Error-Control Techniques for Digital Communication*, Wiley, New York, 1985.
18. A. Murphy, G. P. Picco, and G.-C. Roman, "LIME: a middleware for physical and logical mobility," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Phoenix, USA, April 2001.
19. A. Cole, S. Duri, J. Munson, and D. Wood, "Adaptive service binding middleware to support mobility," *Proceedings of the IEEE International Conference on Distributed Computing Systems Workshops*, Providence, USA, May 2003.
20. C. A. Rusu, R. Melhem, and D. Mosse, "Maximizing the system value while satisfying time and energy constraints," *IBM Journal of Research and Development*, Vol. 47, No. 5/6, September/November 2003, pp. 689–702.
21. E. Park and H. Shin, "Cooperative reconfiguration of software components for power-aware mobile computing," *IEICE Transactions on Information and Systems*, Vol. E89-D, No. 2, February 2006, pp. 498–507.
22. G.-C. Roman, C. Julien, and Q. Huang, "Network abstractions for context-aware mobile computing," *Proceedings of the International Conference on Software Engineering*, Orlando, USA, May 2002.

23. J. Beck, A. Gefflaut, and N. Islam, "MOCA: A service framework for mobile computing devices," *Proceedings of the International Workshop on Data Engineering for Wireless and Mobile Access*, Seattle, USA, August 1999.
24. T. Itao, T. Nakamura, M. Matsuo, and T. Aoyama, "Context-aware construction of ubiquitous services," *IEICE Transactions on Communications*, Vol. E84-B, No. 12, December 2001, pp. 3181–3188.
25. G. Welling and B. R. Badrinath, "An architecture for exporting environment awareness to mobile computing applicaitons," *IEEE Transactions on Software Engineering*, Vol. 24, No. 5, May 1998, pp. 391–400.
26. P. J. Braam, "The Coda distributed file system," *Linux Journal*, June 1998. Available at www.linuxjournal.com/article/2390
27. J. C. S. Lui, O. K. Y. So, and T. S. Tam, "NFS/M: An open platform mobile file system," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Amsterdam, Netherlands, 1998.
28. B. Atkin and K. P. Birman, "MFS: An adaptive distributed file system for mobile hosts," Tech. Rep., Department of Computer Science, Cornell University, 2003.
29. K. Choi, K. Dantu, W. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a mpeg decoder," *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, 2002, pp. 227–231.
30. M. Mesarina and Y. Turner, "Reduced energy decoding of mpeg streams," *Multimedia Systems*, Vol. 9, No. 2, 2002, pp. 202–213.
31. P. Symes, *Video Compression: Fundamental Compression Techniques and an Overview of the JPEG and MPEG Compression Systems*, McGraw-Hill, New York, 1998.
32. I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia*, Wiley, New York, 2003.
33. S. Park, Y. Lee, J. Lee, and H. Shin, "Quality-adaptive requantization for low-energy MPEG-4 video decoding in mobile devices," *IEEE Transcations on Consumer Electronics*, Vol. 51, No. 3, 2005, pp. 999–1005.
34. J. Lee, Y. Lee, S. Park, and H. Shin, "Maintaining video quality for a down-sampling decoder in a low-power mobile device," *Proceedings of the 24th IASTED*, 2006, pp. 73–78.
35. M. Adamou, S. Khanna, I. Lee, I. Shin, and S. Zhou, "Fair real-time traffic scheduling over a wireless LAN," *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, December 2001, pp. 279–288.
36. J. Bennett and H. Zhang, "Worst case fair weighted fair queuing," *Proceedings of IEEE INFOCOM*, April 1995, pp. 120–128.
37. T. S. E. Ng, I. Stoica, and H. Zhang, "Packet fair queueing algorithm for wireless network with location-dependent errors," *Proceedings of IEEE INFOCOM*, April 1998, pp. 1103–1111.
38. W. Li, "Overview of fine granularity scalability of MPEG-4 video standard," *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 11, No. 3, March 2001, pp. 301–317.
39. P. Agashe, R. Rezaiifar, P. Bender, and QUALCOMM, "Cdma2000 high rate broadcast packet data air interface design," *IEEE Communications Magazine*, Vol. 42, No. 2, February 2004, pp. 83–89.
40. J. Wang, R. Shinnarajaj, T. Chen, Y. Wei, E. Tiedemann, and QUALCOMM, "Broadcast and multicast services in cdma 2000," *IEEE Communications Magazine*, Vol. 42, No. 2, February 2004, pp. 76–82.
41. R. E. Blahut. *Theory and Practice of Error Control Codes*, Addison-Wesley, Reading, Massachusetts, 1983.
42. K. Kang, J. Cho, and H. Shin, "Dynamic packet scheduling for cdma2000 1xEV-DO broadcast/multicast services," *Proceedings of the IEEE Wireless Communications and Networking Conference*, Vol. 4, March 2005, pp. 2393–2399.
43. J. Kim, J. Cho, and H. Shin, "Resource allocation for scalable video broadcast in wireless cellular networks," *Wireless and Mobile Computing, Networking and Communications*, Vol. 2, August 2005, pp. 174–180.
44. 3GPP2, C.S0054 v2.0, cdma2000 High Rate Broadcast-Multicast Packet Data Air Interface Specification, 2004.

45. Y. Cho, K. Kang, Y. Cho, and H. Shin, "Improving MAC-layer error recovery for 3G cellular broadcasts," *Proceedings of the Advanced Information Networking and Applications Conference*, Vol. 2, April 2006, pp. 418–422.
46. ISO/IEC 14496-2, Coding of Audio-Visual Objects—Part 2, 2004.
47. K. Kang, J. Lee, Y. Cho, and H. Shin, "Efficient error control for scalable media transmission over 3G broadcast networks", *LNCS*, Vol. 4096, August 2006, pp. 416–426.

38

Embedded Systems and Software Technology in the Automotive Domain

38.1	Introduction	38-1
38.2	The History	38-2
38.3	State of Practice Today	38-2
	The Role of Software in Cars • Embedded Software as Innovation Driver • Deficits in Engineering Software in Cars	
38.4	The Domain Profile	38-3
38.5	The Future	38-4
	Innovation in Functionality • Cost Reduction • Innovative Architectures	
38.6	Practical Challenges	38-6
	Competency and Improved Processes • Innovation in Architecture • Development and Maintenance Processes • Hardware and Technical Infrastructure • Cost Issues	
38.7	Research Challenges	38-11
	Comprehensive Architecture for Cars • Reducing Complexity • Improving Processes • Seamless Model-Driven Development • Integrated Tool Support • Learning from Other Domains • Improving Quality and Reliability • Software and System Infrastructure	
38.8	Comprehensive Research Agenda	38-16
	Foundations: Distributed System Models • Adapted Development Process • Tool Support	
38.9	Conclusion	38-18

Manfred Broy

Technische Universität München

38.1 Introduction

Today, in many technical products, software plays an increasingly dominant role. In cars, this applies even to the extreme. Today, software in cars is a crucial success factor for the car industry, however, bringing various problems, but being nevertheless decisive for competition.

One can easily see that the amount of software in cars has been growing exponentially over the last 30 years, and one can expect this trend to continue for another 20 years at least.

The first bit of software found its way into cars only at a time about 30 years ago—in fact, software became a major issue in only more or less four generations of cars. From one generation to the next, the

software amount was growing by a factor of 10, or even more. Today, we find in premium cars more than 10 million lines of code and we expect to find in the next generation up to 10 times more.

Many new innovative functions in cars are enabled and driven by software. Recent issues are energy management and the current step into hybrid solutions, which can only be realized in an economic way by plenty of software. It is mainly the application domain-specific innovations in cars with their stronger dependencies and feature interactions that ask for cross-application domain considerations and organizations.

In the following, we shortly describe the history of software in cars as far as it is relevant to understand the current challenges. Then we sketch the state of practice with its problems, challenges, and opportunities. Based on a short estimation of the future development we describe our expectation of how the field will develop in the coming years. Finally, we describe current research in the domain of automotive software engineering (see also Ref. 15).

38.2 The History

Just 30 years ago, software was first deployed into cars to control the engine and, in particular, the ignition. These first software-based solutions were very local, isolated, and unrelated. The hardware/software systems were growing bottom-up. This determined the basic architecture in cars with their dedicated controllers (electronic control units or ECUs) for the different tasks as well as dedicated sensors and actuators. Over the time to optimize wiring, bus systems (see Ref. 29) were deployed into the cars by which the ECUs became connected with the sensors and actuators.

Given such an infrastructure, ECUs got connected and could exchange information. As a result, the car industry started to introduce functions that communicated or even were realized distributed over several ECUs connected by the bus systems. Traditionally, such functions were built bottom-up. A systematic top-down design was never used. If we would not go forward in evolutionary steps but redesign the hardware/software systems in cars from scratch today, we would certainly come up with quite different solutions for the architecture of embedded systems in cars.

38.3 State of Practice Today

Today, premium cars feature not less than 70 ECUs connected by more than five different bus systems. Up to 40% of the costs of a car are due to electronics and software.

38.3.1 The Role of Software in Cars

Within only 30 years, the amount of software in cars went from 0 to more than 10,000,000 lines of code. Today, more than 2000 individual functions are realized or controlled by software in premium cars. Of the development costs of the software/hardware systems 50%–70% are software costs. For a view on the network structure in a car see Figure 38.1.

Software as well as hardware became enabling technologies in cars. They enable new features and functionalities. Hardware is becoming more and more a commodity—as seen by the price decay for ECUs—while software determines the functionality and therefore becomes the dominant factor [8].

38.3.2 Embedded Software as Innovation Driver

Software is today the most crucial innovation driver for technical systems, in general. By software the industry realizes innovative functions, it finds new ways of implementing known functions with reduced costs, less weight, or higher quality; by embedded software we save energy and, what is in particular important, we combine functions and correlate them into multifunctional systems.

This way software allows for completely new solutions of, in principle, known tasks. Most important, of course, is embedded software as an enabling technology that step-by-step passes the road introducing new

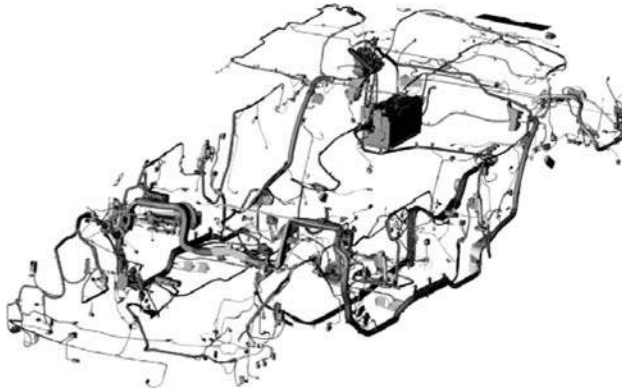


FIGURE 38.1 Onboard network.

functions as part of a multifunctional system. What has been said here for embedded systems, in general, applies to cars, in particular.

38.3.3 Deficits in Engineering Software in Cars

Today, the engineering of software in cars is still in its infancy. The quick increase of software and the traditional structures of the car industry make it difficult for this old economy to adapt fast enough to the quite different requirements of a software-intensive system in which cars become more and more. New skills and competencies are required at all levels of the Companies.

Following its tradition to find its own proprietary solutions (see Ref. 7), the car industry developed to a large extent its own approaches also in the software domain. It is amazing to see the amount of proprietary technology in the software in cars. This applies to operating system, communication protocols, tools, architecture and even methodology, in fact, basically all aspects of software in cars. Of course, automotive software engineering has its own domain-specific requirements (see below). Nevertheless, the car industry could have done much better by benefiting from existing experiences and technology from other domains, in particular, telecommunication and avionics.

Life-cycle management of software in cars is in its early stage. Many suppliers and even some OEMs are not even at CMM level 2. This is, in particular, a problem in a situation where systems are developed by distributed concurrent engineering and the software is highly complex, multifunctional, distributed, real time, and safety critical.

Reuse of solutions (see Ref. 21) from one car to the next is insufficient and only done in a consequent way in some limited areas. In many subdomains, the functionality from one car generation to the next is only changed and enhanced by 10%, while more than 90% of the software is rewritten. One reason is a low-level, hardware-specific implementation, which makes it difficult to change, adopt, and port existing code. Another reason is insufficient cross-product software life-cycle management.

Finally, the amount of automation in software production for software in cars is quite low. Tools are only used in an isolated manner. There is neither a properly defined design flow nor seamless tool chains for distributed functions.

38.4 The Domain Profile

Traditionally, the car industry was highly vertically organized. In software engineering, we would say it is modular with respect to the functions. The mechanical engineers worked hard for over 100 years to make the various subsystems in cars in their development and production quite independent. This facilitates independent development and production of the subparts and allows for an amazing division of labor.

As a result, suppliers could take over a considerable part of the engineering, the development, and also the production by a consequent outsourcing. Ideally, the parts of cars are produced by a chain of suppliers and finally more or less only assembled by the car manufacturer (called OEM in the following). Thus, a large amount of the engineering and production is outsourced and the cost and risk distribution can be optimized. A car is (or better was) considered as a kit of subparts that are assembled by the OEM.

With software becoming a major force of innovation this situation changed drastically:

- Traditionally, quite unrelated and independent functions (such as braking, steering, or controlling the engine) that were freely controlled by the driver get related and start to interact. The car turns from an assembled device into an integrated system. Phenomena such as unintentional feature interaction become issues.
- Assembling subparts becomes system integration.
- The behavior of cars becomes much more programmable. Certain properties, such as comfort or sportive handling are no longer solely determined by mechanics but to a growing fraction also by the by software.
- The costs of cars get more and more influenced by development costs of software, for which the traditional cost models dominated by the cost-by-part paradigm are no longer fully valid.

Size and structure of the embedded software/hardware systems in cars are enormous. The application software is built on top of real-time operating systems and bus drivers. Most of the software is hard real-time critical or at least soft real-time critical.

The requirements for the embedded software systems in cars are quite specific and show high requirements on quality:

- Wide range of different users (not only drivers and passengers, but also maintenance)
- Specific maintenance situation
- Safety-critical functions
- Specific context of operation of the systems
- Heterogeneity of functions (from embedded real-time control to infotainment, from comfort functions like air condition to driver assistance, from energy management to software download (flash) functionality, from air bags to onboard diagnosis and error logging)

As a result, the complexity and spectrum of requirements for onboard software is enormous.

38.5 The Future

The increase of software and functionality in cars is not close to an end, in the contrary. We can expect a substantial growth in the future. The future development is driven by the following trends:

- High demand of new innovative or improved functionality
- Quickly changing platforms and system infrastructures
- Rapid increase in development cost in spite of a heavy cost pressure
- Demand for higher quality and reliability
- Shorter time-to-market
- Increased individualization

As a result, there is a high demand for dedicated research on software and systems engineering on automotive systems.

38.5.1 Innovation in Functionality

Software will remain the innovation driver for new functionalities in cars for the next two decades. We will see many new software-based functions in cars in the near future. Each new software-based function that comes into the cars enables several further features. This accelerates the development and innovation.

38.5.1.1 Crash Prevention, Crash Safety

Today, the safety standards in cars are very high. The rates of people seriously injured or killed in their cars in accidents are decreasing in spite of increased traffic. Statistically, software in cars helped impressively to prevent accidents and many severe injuries in accidents. Nevertheless, the systems are far from being perfect by now. New generations of crash prevention, precrash, and crash-mitigating functions are in preparation.

One of the promising areas of crash prevention is context-aware system adaptation and advanced driver assistance (see below). By comprehensive sensor fusion and global data management, the situation of the car, the driver, and the traffic can be recognized and extended crash prevention can be realized.

38.5.1.2 Advanced Energy Management

Hybrid cars are just at their beginning. In future cars we can expect a technical infrastructure that takes care of many in-car issues like energy consumption, car calibration, and management of the electric energy available.

Energy consumption is one of the most relevant issues in cars today and even in the future. In current years, engines have become most efficient while the size, weight, and power of cars as well as the electronic equipment have increased the energy consumption. This is not only a question of economy or ecology. It is also an issue of providing enough electrical energy by batteries. Moreover, if safety-critical functions are based on embedded software, the reliable provision of electrical power becomes safety critical by itself.

Today, we find more and more functions in cars that aim at a sophisticated energy management, aiming at the same time at saving energy and making sure that enough energy is available for safety-critical functions.

38.5.1.3 Advanced Driver Assistance

The complexity of the software systems in cars for their drivers, passengers, and also for maintenance is too high. What can help is a variety of driver assistance functions at all levels, not only supporting instantaneous driver reactions but also providing short-term driving assistance in, for instance, lane departure or tour planning.

Apart from crash prevention, driver assistance may also provide a much higher comfort for drivers and passengers. It can help considerably to reduce the complexity of the driver's task today not only by assisting driving, but also by managing related tasks such as communicating and navigating while driving.

38.5.1.4 Adaptable HMI

What seems less far in the future are integrated seamless adaptive HMI (human-machine interaction) systems in cars. Cars get more complex also owing to their software-based functions—but they get safer owing to that software and they get more convenient. But to get an easy access to this convenience we have to offer those functions to drivers and passengers in a way where they do not have to operate all this complexity explicitly. Adaptive context-aware assistance systems which grasp the situations and are able to react within a wide range without too much explicit interaction by the driver or the passengers can lead to a new quality of HMIs.

Another issue is the wide range of different car users and their different expectation. Here, personalization and individualization are key goals.

38.5.1.5 The Programmable Car

Equipped with various actuators and sensors, as premium cars are today, we get close to a point where we can purely by programming and by introducing new software, create new functions for cars. This comes close to the vision of the programmable car.

Adding new functionalities by downloads opens new business areas; this way a car can be updated and new functionalities can be added.

38.5.1.6 Personalization and Individualization

A promising issue is the personalization and individualization of cars. Drivers are quite different. When cars get more and more complex, of course, it is crucial to adapt the ways cars have to be operated to the individual demands and expectations of the users.

Individualization is very much a trend, today. Consumers like to have products that individually address their needs.

38.5.1.7 Interconnection Car Networking

Another notable line of innovation is the networking of on- and off-board systems. Using wireless connections, in particular, peer-to-peer solutions, we can connect cars, which gives many possibilities to improve safety issues in the traffic or to find new solutions in the coordination of traffic far beyond the classical road signs of today. For instance, in the long-term future when we can imagine that all road signs are complemented by digital signals between the cars, we can have a completely different way of traffic coordination.

Peer-to-peer communication is one of the areas where a lot of new innovations can be expected. An example is the communication of traffic load information (about traffic jams, accidents, or simple bad road conditions or the weather) between adjacent cars.

38.5.2 Cost Reduction

The software costs in car increase enormously. These are not only pure development costs. Sometimes even more significant are maintenance costs and especially warranty costs.

Looking at the often exponential increase in the costs there is some urgent need to develop ideas on how to handle reduction in these costs by advanced development technology. Here there are basically two options: automating development by better methods and tools as well as improving reuse. An improvement of the systematics of development can also be achieved by a better infrastructure that reduces the hardware dependency of the implementation.

38.5.3 Innovative Architectures

The car of the future will certainly have much less ECUs in favor of more centralized multifunctional multipurpose hardware, less communication wires and less dedicated sensors and actuators, but more general sensors for gaining information via sensor fusion. Arriving today are more than 70 ECUs in a car, further development will rather go back to a small number of ECUs by keeping only a few dedicated ECUs for highly critical functions and combining other functions into a smaller number of ECUs, which then would be rather not special-purpose ECUs, but very close to general-purpose processors.

It would affect the way software is packaged and deployed. Such a radically changed hardware would allow for quite different techniques and methodologies in software engineering.

38.6 Practical Challenges

Software issues hit the car industry in a dramatic way. In a period of only 30 years, the amount of software-related development activities went from 0% to 30% or even 40%. If we assume that an engineer works about 35–40 years in an industry, it is obvious that the companies were not able to gather sufficient competencies quickly enough. A second problem is that there are not enough software engineers educated by the universities in the skills needed in the embedded and especially the automotive domain.

The high intensity of software in cars puts the car industry under stress and a high change pressure. The reasons are manifold. First of all, an important issue is the dependencies between the different functions in the car leading to all kinds of wanted or unwanted feature interactions.

This is quite different from what the automotive industry was used to be before software came into the cars. Over a hundred years the car industry managed to make their different functionalities as independent

as possible such that cars could be developed and produced in a highly modular way. With the growth of software-based functions in the cars this independence disappeared. Today, a car has to be understood much more as a complex system where all the functions act together closely. So software engineering in cars needs to take a massive step into systems engineering. Cars have become typically multifunctional systems.

38.6.1 Competency and Improved Processes

The growth of software in cars means that the car industry needs new competencies. It has to master the competency management to build up software competencies as fast as needed.

However, the car industry needs completely new development processes. Processes that—in contrast to those used today—are better in dealing with and are much more influenced by software issues. It is fascinating to see how the processes and models of software engineering influence more and more what is going on in mechanical engineering in the automotive domain.

38.6.1.1 From Software to Systems Engineering

In the end, the whole structure and organization of the automotive industry starts to change. One issue is the amount of development done by the OEM. The general tendency is that basically all implementation is done via outsourcing. However, as long as the OEMs are interested to do the integration work themselves it is obvious that the OEM has to gain a deep understanding of the software-intensive systems.

In systems engineering with concurrent and multiple distributed development activities, the understanding of the architecture becomes crucial. The architecture not only determines the quality of software systems but also the structuring of the development process [20,25,26].

38.6.1.2 The Role of Control Theory

Traditionally, control theory plays a prominent role in the car development. However, today a lot of the software in cars is not actually control theoretic but event based. The challenge here is to find the right theory and methodology to combine control theory and the engineering of discrete-event systems.

So far, the interaction between the models of control theory (differential equations) and the models of discrete-event systems (discrete state machines) are not sufficiently understood. There does not exist an integrated theory and methodology for developing distributed event-based systems with a high content of control theory.

38.6.1.3 Chances and Risks

The speed of the development, the complex requirements, the cost pressure, and the insufficient competency in the field bring enormous challenges and risks but also high potentials and opportunities for improvement. Mastering these challenges needs technical, management, and economic skills.

So far there is not enough data and analysis available to be able to have a proper understanding of what the main problems are. A deeper analysis and a systematic data collection can help to substantially understand where to improve the technology and methodology.

38.6.2 Innovation in Architecture

The enormous complexity of software in cars asks for an appropriate structuring by architectures in layers and levels of abstraction [22].

38.6.2.1 Functionality

One of the most interesting observations is the rich multifunctionality that we find in a car today. In premium cars, we find up to 2000 or more software-based functions. Those functions address many different issues including classical driving questions but also other features in comfort and infotainment and many more. Most remarkably, these functions do not stand alone, but show a high dependency between each other. In fact, many functions are very sensitive with respect to other functions operated at the same time.

So far, the understanding of these feature interactions between the different functions in the car is insufficient. We hope to develop much better models to understand how to describe a structured view on multifunctional systems like those found in cars.

38.6.2.2 HMI

What is obvious today and what is well understood by now is that the human machine interfaces (HMI) in cars have to be done in a radically different way. It was BMW that was brave enough to do a first step in the right direction. Their iDrive concept is very much influenced by the interaction devices of computer systems like the concept of a computer mouse or a touch pad we are used to by our computers, today. BMW got a lot of criticism for that step. Meanwhile, all its competitors have followed the same road.

But quite obviously, these are merely the first steps. Multifunctionality of cars needs flexible ways of addressing as well as operating and interacting with all those functions.

What makes the situation more difficult than in classical computers is, of course, that car drivers cannot pay as much attention to the system functions as computer users would, but rather must concentrate on the traffic and driving; thus, drivers should get a user interface, which allows them to deal with the many functions in a car in a way that takes not too much of their attention compared to the attention given to the traffic.

38.6.2.3 Complex Comprehensive Data Models

Currently, in cars there is only a very distributed uncoordinated data management. Each of the ECUs contains and manages its own data. But we should not think about these data as a distributed database that is well organized based on some kind of data integrity. Instead, all the different ECUs and functions keep a large portion of their data separately including redundant information that is not synchronized. This can lead to a kind of schizophrenic situation in cars where some ECUs think, according to their local data, that the car is moving, while others believe that the car has stopped.

It would be an interesting exercise to design the software architecture of a car in a way that there is an integrated interfunction data model that includes sensor fusion and overall car data management.

38.6.3 Development and Maintenance Processes

Certainly, the process for the development of software systems gets more and more complex. What we need is a suitable, well-structured, and methodologically well-supported process that reduces complexity, enables innovation, saves costs, is transparent, and addresses outsourcing.

38.6.3.1 Requirements Engineering

One of the biggest problems in automotive software engineering is tailored requirements engineering, and that this is essential is quite obvious because a lot of the functions in cars are innovative and completely new. When introducing new functions, of course, we have no experience with them. What is the best way to work out the detailed functionality, what are the best dialogs to access the functions, what are the best reactions of the systems? By software we get much larger design spaces for solutions than we had in cars before. Therefore, requirements engineering is one of the crucial issues (see Refs. 18 and 19).

In addition, some of the requirements engineering has to be done inside the OEMs and the supplementary requirements engineering has to be added by the suppliers, which usually carry out the implementation of the functions. Therefore, the communication between OEMs and suppliers has to be organized via the requirements documents, which nowadays are often not precise and not complete enough.

This brings us to the issue of distributed concurrent engineering. Typically, in the automotive industries we have a change. The more costly and complex systems become in their development, the more important it is to use good product models to support the integrity of the information exchange within the supplier chains.

38.6.3.2 Designing the Hardware/Software Architecture

Designing the architecture in an IT system in a car means to determine the hardware architecture consisting of ECUs, bus systems and communication devices, sensors, actuators, and the HMI. The software infrastructure is based on this hardware structure including the operating system, the bus drivers, and additional services. This system software forms, together with the hardware, the implementation platform.

The application software is based on the platform and consists of the application code. This underlines the significance of the platform for many typical software engineering goals such as suitable architecture, separation of concerns, portability, and reusability.

38.6.3.3 Coding

Suppliers carry out most of the coding, today. Only in extraordinary cases the OEM, for instance, produces code for some of the infrastructure (such as bus gateways).

A lot of the code is still written by hand, although some tools generate good-quality code. Code generation, however, is often considered not efficient enough to exploit the ECUs in the optimal way. Highly optimized code, however, makes reuse and maintenance quite hard.

38.6.3.4 Software and Systems Integration

Since today, during their design, architecture and the interaction between the subsystems are not precisely specified, and since the suppliers realize the subsystems in a distributed process, it is not surprising that integration is a major challenge.

First of all a virtual integration and early architecture verification is not done today due to lack of methodology. Appropriate tooling is not possible, today, also due to the lack of precise requirements, architecture, and interface specifications. Second, in turn the subsystems delivered by the suppliers do not fit together properly and thus the integration fails. Third, when trying to carry out the error correction owing to the missing guidelines of architecture, there is no guiding blueprint to make the design consistent.

38.6.3.5 Quality Assurance

A critical issue of the car industry is quality. Since the car industry is so much cost aware, quality issues are often not observed in the way advisable for a software system (see Ref. 13). This and the application of established certification processes in the avionic industry are the reasons for airplanes' reliability outmatching cars' reliability by far.

But not only is reliability a critical aspect for cars with respect to quality, others concern maintainability in long-term operation. We have to understand much more in detail, which quality aspects are of importance for software in cars and how we can support and improve them.

38.6.3.6 Maintenance

A critical issue is, of course, that cars are in operation over more than two or three decades. This means we have to organize long-term maintenance. Maintenance includes a number of issues:

Compatibility. Maintenance of the software in cars is not so easy. Today new versions of software are brought in during maintenance by flashing techniques, that is, replacing the software of an ECU by another. But doing this, one has to be sure that the new versions interoperate with the old version. In other terms, we have to answer the question whether the new version is compatible (see Ref. 34) with the one we had before. A lot of the problems we see today in cars in the field are actually compatibility problems.

To overcome these problems we need a comprehensive tractable theory of compatibility and practical methods to apply it. Compatibility has to become a first-class development goal.

Defect diagnosis and repair. An interesting observation says that today more than 50% of the ECUs that are replaced in cars are technically error-free. They are just replaced because the garage could not find a better way to fix the problem. However, often the problem does not lie in broken hardware but rather ill-designed or incompatible software.

Actually, we need much better adapted processes and logistics to maintain the software in the cars. Understanding how we maintain and a further development of the software architecture in cars, understanding the configurations and version management, and making sure that not very well-trained people in garages can really handle the systems is a major challenge.

Changing hardware. Hardware has to be replaced in cars if it is broken. Moreover, over the production time of a car model, which is about 7–8 years, not all the ECUs originally chosen are available in the market for the whole period. Some of them will no longer be produced and have to be replaced by newer types. Already after the first 3 years of production 20%–30% of the ECUs in the car typically have to be replaced due to discontinued availability of the original ECUs. As a result, the software has to be reimplemented, since it is tightly coupled with the ECU. Also, for these reasons portability and reusability become more and more important for car industry.

Again here compatibility is a key question. Software is needed that is easily portable and compatible with new hardware devices.

38.6.4 Hardware and Technical Infrastructure

Today, in cars we find a very complex technical infrastructure. We have up to five bus systems and more. We find real-time operating systems, a lot of systems and software as part of the technical infrastructure on which the applications are based. This is why relatively simple applications get wildly complex since they have to be distributed and executed to communicate over a complex infrastructure.

One of the problems of this infrastructure is that on the bus level there is a lot of multiplexing going on. The same holds for the ECUs where there are tasks and schedulers. Actually, we can find all the problems of distributed systems—in a situation where physical and technical processes have to be controlled by the software, some of them highly critical and hard real-time.

What creates the big problems owing to the multiplexing going on in the cars? In the transmission time of the messages, there is some jitter and delay such that systems appear to be nondeterministic and that in many cases time guarantees cannot be given. This is one of the reasons why we do not have more X-by-wire solutions in cars today. On the one hand, the reliability today is not good enough, and on the other, the time guarantees are not good enough. Therefore, a lot of interesting potentials for improvement looking at drive-by-wire systems are not realized so far.

Today, new infrastructures are suggested that support the time-triggered paradigm such as FlexRay. Introducing such new techniques, however, requires new development techniques, as well.

38.6.5 Cost Issues

Traditionally, the car industry is very cost aware. Competition is to a large extent determined by prices on one side and by brand image on the other. Image is determined by design, quality, comfort, and innovation. The last three factors and also the cost structure is heavily influenced by software in cars.

38.6.5.1 Software Cost Control

Software cost control is today, of course, closely related to the traditional cost per piece and production-centric cost models in the car industry. This is not very adequate for software where development cost is dominant. However, we observed an exponential growth of software costs in cars in recent years.

At the moment most of the software in cars is reimplemented over and over again. The reasons for that are to a large extent the optimization of the costs per part. The car industry always tries to use the cheapest processors they can find and to exploit more than 80% of their power and capacity. As a result, late changes bring those processors close to their limits and thus the software has to be highly optimized. This causes that the software cannot be reused from one ECU to the next.

38.6.5.2 New Players in Field

Software will become an independent subpart in the automotive domain. This is due to the fact that more and more ECUs are no longer dedicated to one application, but are or will be multiplexing several subapplications. This means that suppliers no longer produce encapsulated solutions where ECUs, sensors,

actuators, software, and hardware as well as the mechanical devices are developed as one integrated piece. The software then has to be delivered separately running on an ECU not delivered by the same supplier. As a result, the software is becoming more rather like a device driver of the mechanical device. In fact, then there is no real reason why the software has to come from the supplier producing the mechatronic part, it may come separately from a software house. This way, software becomes an independent subproduct and subsystem for the car industry.

Owing to this observation it seems very likely that new players come into the field. Sometime ago a software house would not be an interesting first-tier supplier for the embedded systems of the car industry. This is about to change radically.

Another interesting role may be that of a systems engineer. Systems engineering such as the decomposition of systems and their functionality as well as the integration are major issues of engineering software-intensive systems in cars.

38.6.5.3 Long-Term Investment

An economically interesting question is who will, in the long run, own the investment that is created by the development of software in the car. It is not clear at all who will own the intellectual property for that investment and therefore who will, on the long run, be the dominant player in the industry.

The software in cars is already such a high-cost investment today that very soon it will be quite difficult for newcomers to enter the market. A critical question is whether we will observe monopolization like in other software fields for software in cars on the long run.

38.6.5.4 Reuse and Product Lines

One of the big hopes for cost reduction in software development for the automotive domain is product line approaches. These offer a domain-specific organized way of reuse. But so far, product line engineering is only used by a few suppliers and not systematic at all by the OEMs.

In the end, the product line approach has to be applied and specialized for the particular requirements of software in cars. The cooperation between the suppliers and the OEM are of particular interest here.

38.7 Research Challenges

As explained, the car industry is facing many challenging issues for software in cars. This opens a wide field for research.

38.7.1 Comprehensive Architecture for Cars

Owing to the multifunctionality and all the related issues, we need a sophisticated domain-specific structural view onto the architecture in cars that addresses all the aspects that are relevant. In such an architecture (see Ref. 35), we distinguish ingredients that we briefly explain in the following.

38.7.1.1 Functionality Level—Users View

The usage view aims at capturing all the software-based functionality offered by the car to the users. Users include not only drivers and passengers but also the people in a garage and maintenance staff, perhaps even the people in the production and many more. We call this the functionality level (see Ref. 28).

In any case, the functionality level provides specifically a perspective onto the car that captures and structures its family of services and aims at understanding how the services are offered and how they depend on and interfere with each other. The so-called feature or function hierarchies can model this best. The individual functions can then be modeled by state machines and their relationships by specific coordination languages.

38.7.1.2 Design Level—Logical Architecture

The design level addresses the logical component architecture for the software-based functions in cars. In a logical architecture, the functional hierarchy is decomposed into a distributed system of interacting components.

At the design level we describe the distributed architecture of a system, independent from the question whether the components are implemented by hardware or software. The logical architecture can be described by a number of interfaces for communicating state machines with input and output that realize the functions that are found in the system. Via their interaction, they realize the observable behavior described at the functionality level. The logical architecture describes abstract solutions and to some extent the protocols and abstract algorithms used in these solutions [17].

38.7.1.3 Cluster Level

In the clustering, we rearrange the logical architecture in a way that prepares the deployment and the step toward the software architecture. This is a step from the logical architecture, where application-specific issues are dominant, toward technical issue. This step prepares the structuring into a software architecture as a basis of deployment.

38.7.1.4 Software Architecture

The software architecture consists of the classical division of software into platforms comprising operating systems and bus drivers on one side and the application software represented by tasks, which are scheduled by the operating system, on the other. This software has to be deployed onto the hardware.

The application software forms an architecture of its own. It uses the infrastructure software and depends on it. The application software has a source code architecture, which is used to structure the code at design time. Furthermore, there exists a runtime architecture that structures applications in tasks and processes.

38.7.1.5 Hardware Level—Hardware Architecture

The hardware architecture consists of all the devices including sensors, actuators, bus systems, communication lines, ECUs, MMI, and many more. It forms the basis for the execution of the software that defines the specific functions.

There is, of course, a close interrelationship between the hardware and the software. To make both more independent on one side and easier to integrate on the other is one of the long-term goals.

38.7.1.6 Deployment—Software/Hardware Codesign

Finally, we need a deployment function that relates hardware to software. The hardware/software and the deployment function together have to represent a concrete realization of the logical architecture that just describes the interaction between the logical components.

One ambitious goal is to make deployment decisions more flexible. As the ultimate goal, we would like to have a dynamic deployment where the flexibility is kept for the execution level.

38.7.1.7 Architecture Modeling and Description

A modeling approach is needed that is expressive enough to deal with all the mentioned aspects of architecture. Such a fully worked out modeling approach does not exist until now.

There is no concrete architecture description technique which captures at least the most important views onto architectures.

38.7.2 Reducing Complexity

One of the biggest problems in the car industry today is the overwhelming complexity of the systems they face today. How can we reduce complexity?

Of course, we can use classical techniques from software engineering, which is structuring, separation of concerns, and abstraction. Structure is what we have achieved if we get an appropriate architectural view with the levels as described above. Abstraction is what we gain if we use appropriate models. Model orientation remains one of the big hopes for the car industry to improve its situation.

38.7.3 Improving Processes

A key issue is process orientation and software development processes. So far the processes in the car industry are not adapted to the needs of software-intensive systems and software engineering. Process orientation aims at a much higher awareness for the significant role of processes.

38.7.3.1 Maturity Levels

A good example for process orientation is the introduction of Spice and CMMI techniques into the car industry, which has already helped a lot to improve the awareness and the competencies there. However, such a proceeding has to go hand in hand with improving the quality of the models and tools.

Actually, a deep process orientation on the long run would need a well-understood handling of the product models. The product data of course need a comprehensive coherent and seamless model chain. Here, we find an exciting dependency between engineering support software and embedded onboard software.

38.7.4 Seamless Model-Driven Development

One of the great hopes for improvement is seamless model-driven development. This means that we work with a chain of models for the classical activities:

- Requirements modeling
- Design modeling
- Implementation modeling
- Modeling test cases

A vision would be an integrated modeling approach where the relationship between all the models is captured and parts of the next models are generated from the models worked out before.

38.7.4.1 The Significance of Models

Models, if they are formalized and have a proper theory, are useful in many respects. Starting with documentation, formalization, and making informal descriptions precise we can go on with analysis, reuse, transformation, code generation, and finally enter into product lines. In particular, well-chosen models are the key to tooling and automation (Figure 38.2).

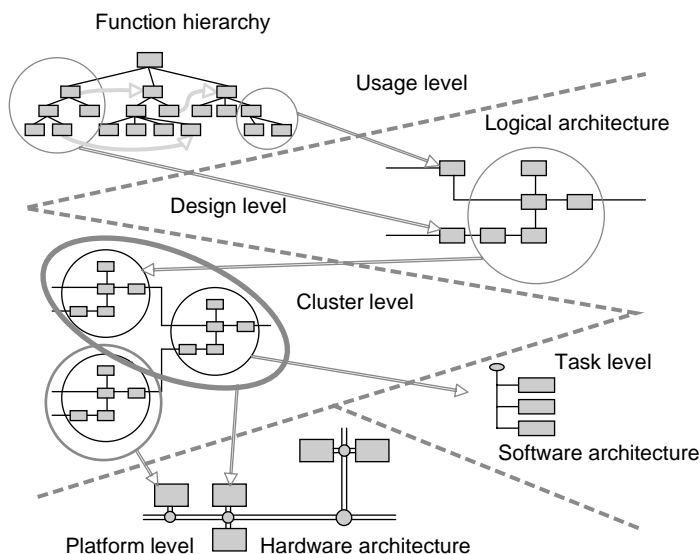


FIGURE 38.2 Comprehensive architecture.

Finally, models capture development knowledge, know-how about the products, and give guidelines for the development processes.

A key issue is the modeling of architecture at all levels as described above. This needs powerful modeling techniques for structuring the functionality and the design. This includes issues of timing and interfaces.

38.7.4.2 Weaknesses of Modeling Today

Today models and model-based development are used to some extent in the automotive industry but their use is fractal. Modeling is applied only at particular spots in the development process. We speak of “isolated islands of modeling.” By not being able to follow a seamless modeling in the development a lot of the benefits of formal modeling gets lost that could be exploited if we had integrated model chains.

Since the models are only semiformal and the modeling languages are not formalized, a deeper benefit and tool support is not achieved. Typical examples are consistency checking or the generation of tests from models (see Ref. 14). Another issue is that models could help very much in the communication between the different companies such as OEMs and first- and second-tier suppliers; also here models are used only in a limited way so far.

Pragmatic approaches such as UML (see Ref. 31) and other modeling approaches used in industry are not good enough. Such approaches are not based on a proper theory and sufficient formalization. As a result, tool support is weak, possibilities to do analysis with models are weak, and the preciseness of the modeling is insufficient.

It is important to keep in mind that modeling is more than documentation. UML should rather be called UDL (unified documentation language). So far UML is not a full modeling language. It does not offer a good basis for abstraction, automation, and refinement.

Modeling can help to improve the quality of the automotive software systems. As it is well known, quality has a wide spectrum of different aspects and facets. Unfortunately, what we can find in modeling today cannot help so much as to improve the quality of the development processes and of the software-intensive products, because it does not give precise and uniform views onto systems based on theories. As a result, possibilities for automatic analysis are missing and tool support remains weak.

38.7.4.3 Potentials of Modeling

Using models in an integrated seamless way, we come up with a vision of a strictly model-based process in the automotive industry. The idea is as follows:

After the first business requirements are captured and the most important informal requirements are brought together, we split the requirements into functional and nonfunctional requirements. Functional requirements are formalized by a function hierarchy where all the atomic functional features are described by interacting state machines or by interaction diagrams between the different functions. In the function hierarchy, dependency relations are introduced.

The nonfunctional requirements are process requirements and quality requirements for the product. For the product quality requirements a quality profile is worked out. We have to understand how the quality requirements determine properties of the architecture. This way we form a quality-driven architecture onto which we map the functional requirements.

Along this line we arrive at system decomposition. In the decomposition, we have to specify a logical architecture and the interfaces (see Ref. 32) of its logical components. At that level we can already do a proof of concept in terms of a virtual verification of the logical architecture as long as we have formal models for the logical components and their composition. This step already proves that at the application level the architecture is correct.

From here on we do the decomposition of the components in software parts, we design the hardware architecture, and we design the deployment.

38.7.5 Integrated Tool Support

Tool support is a key issue. Today, we find a rich family of tools in use but unfortunately these tools are not integrated. Therefore, there are a number of attempts to create pragmatic tool chains by connecting

the tools commercially available in a form where the data models of one tool are exported and imported by the next tool. However, this does not help if there is not a semantic understanding how the different tools are based on joint concepts.

Unfortunately, each of the tools uses its own concept of a system, its own abstractions and models. Thus, it is very difficult or even impossible to connect the tools and to transform the model content of one tool in a correct way into the model content of the next tool.

The development of integrated tool chains that support the whole development process in a seamless way based on a proper methodology and system modeling techniques is one of the unsolved challenges—not only in automobile software engineering.

38.7.6 Learning from Other Domains

In the automotive industry we basically find all the challenging problems that we find in software engineering, in general, sometimes even in a more crucial way. It is an appealing question, what we can do to bring solutions and ideas from software engineering in general into the automotive domain to try them out there, to improve them and to get some feedback, and maybe improvements and new ideas for software engineering as such.

Avionics is an area that has a lot in common with the automotive domain. There software-intensive systems and function have been introduced one or two decades earlier. So a lot of progress has been made there that is interesting to the automotive domain. However, there are significant differences between the automotive and the avionics industries. They are found in the difference in customers, prices per unit, production numbers per unit, and in the production time per model.

38.7.7 Improving Quality and Reliability

Today the reliability of software in cars is insufficient. In avionics, reliability numbers of 10^9 h mean time between failures and more are state of the art. In cars we do not even know these figures. Only adapted quality processes can improve the situation. As an example, Toyota is quite successful with reviews based on failure modes.

38.7.7.1 Automatic Test Case Generation

The amount of quality assurance in the development of cars is enormous. Today, the industry relies very much on hardware and software as well as on system in the loop techniques (HIL/SIL). There subsystems are executed under simulated environments in real time. However, this technology is coming to its limits because of the growing combinatorial complexity of software in cars.

More refined test techniques can help here, where test cases are generated based on models. The deep benefits of such an automatic test generation from models can only be captured, however, if modeling is integrated into the entire development process. Then the creation of models pays back in many ways—not only for testing.

38.7.7.2 Architecture and Interface Specification

A critical issue for the precise modeling of architectures is the mastering of interface specifications. So far interface specification methods in software engineering in general are not good enough. This fact is, in particular, a disaster for the car industry, because of its distributed mode of development.

The OEM has to do the logical architecture and to decompose the system into components and then distributes the development of the components that correspond to the components of the logical architectures to the suppliers. The suppliers do the implementation, even supply the hardware and bring back pieces of hardware and software that then have to be integrated into the car and connected to each other via the bus systems. As long as the interfaces are not properly described, the integration process gets into a nightmare. A lot of testing and experimentation have to go on, a lot of change processes are needed to understand and finally work out the integration process.

38.7.7.3 Error Diagnosis and Recovery

Today, the amount of error diagnosis and error recovery in cars is limited. In contrast to avionics, where hardware redundancy and to some extent also software redundancy is used extensively, in cars we do not find a comprehensive error treatment. In the CPUs some error logging takes place, but there is neither any consideration nor logging of errors at the level of the network and the functional distribution. In fact, there is neither a comprehensive error diagnosis nor any systematic error recovery.

There are some fail safe and graceful degradation techniques found in cars today, but there is not a systematic and comprehensive error treatment.

One result of this deficiency is problems in maintenance. Today—as mentioned above—in the cause of repair of a defect more than 50% of the hardware devices that are replaced in garages are physically and electrotechnically without defects. They are replaced, since a successful diagnosis, error tracing, and error location did not work and thus by replacing the CPU software is replaced, too, such that the error symptom disappears—but often further, different errors show up later.

38.7.7.4 Reliability

Another critical issue is reliability. We do not actually know how reliable our cars are in terms of mean time to failure. In fact, there are no clear reliability numbers and figures. In contrast to the avionic industry there is no serious reliable calculation for cars.

The high reliability in the avionic field is of course, to a large extent, due to the fact that they use very sophisticated error-tolerance methods, redundancy techniques, and at the same time they work with a sophisticated way of error modeling (such as FMEA). In cars today, errors are only captured within processors in the so-called error logging stores. FMEA today, as far as it is done at all, is quite separated from general modeling. Also here, a methodological integration is needed.

What we need on the long run are comprehensive error models in cars and also software that is prepared for the detection of errors. On such models, we can base techniques to guarantee fail safe and graceful degradation and in the end also error avoidance by the help of software redundancy. Another issue that can be attached by error models is more sophisticated error logging to get into a better error diagnosis for maintenance.

38.7.8 Software and System Infrastructure

An important step in car industry is to go away from proprietary solutions and to develop standards. A promising step in that direction is the AUTOSAR project (see Refs. 4, 5, and 30) that defines a uniform platform architecture. AUTOSAR is only one small step, however.

In the long run, we need not only a standard infrastructure but also application frameworks on top of which we build applications based on techniques of reconfiguration and code generation. This can improve the cost situation greatly.

38.8 Comprehensive Research Agenda

In our research group at TUM we started research in the area of automotive software engineering more than 10 years ago. Over one decade, we developed, step by step, very tight research cooperations with major OEMs and suppliers leading to a kind of strategic partnership.

On the one hand, our research agenda is strongly influenced by urgent questions in industry. On the other, our research results are accepted as helpful contributions and input to improve the situation. This also enables a lot of fruitful joint research and development.

38.8.1 Foundations: Distributed System Models

Software in car forms a concurrent, distributed, interacting, often hard or at least soft real-time system. Modeling and understanding such systems lies in the center of software and systems engineering. What

we need there is a comprehensive theory of modeling as a basis for capturing the functionality and the architecture. We have worked out a large part of such a theory of modeling in recent years (see Refs. 10, 11, 12, and 23).

Such a theory of modeling has to contain a theory of functions, components, features, interfaces, state machines, function combination and component composition, property, interface as well as time granularity refinement, hierarchical decomposition and architecture, communication, levels of abstraction, architectural layering, and interaction processes and how all these are connected on a conceptual basis which includes a notion of time.

If we work out such a theory in the right way we gain all what is needed to solve many of the issues, which we need to understand to be able to model and develop automotive software systems in a systematic way.

38.8.2 Adapted Development Process

Based on our observations we aim at improving the development process, in particular, adding roles for the principal and the supplier into a software life-cycle model, the V-Model XT (see Refs. 16, 24, and 27) developed by our group for the German Government.

38.8.2.1 Requirements Engineering

Requirements engineering has to address the needs of multifunctional systems and concurrent distributed engineering. In particular, specific models are needed to support requirements engineering.

38.8.2.2 Architecture

We have developed a comprehensive view onto architectures defining levels and layers of abstraction along the lines described above (see Ref. 6).

38.8.2.3 Methodology

In a strictly model-based development, we support all the steps in the development process via models and techniques of validation, verification, transformation, and generation.

38.8.2.4 Testing

Automatic test generation is a very promising approach. Once models have been created, we can generate test cases from them, leading to a much better coverage and much deeper test impact.

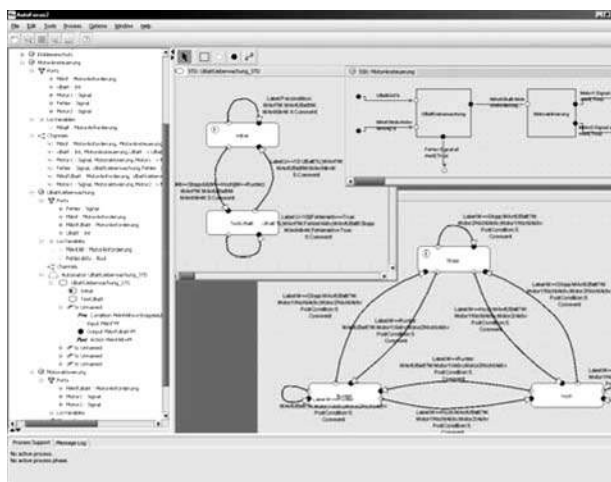


FIGURE 38.3 Screenshot from the Tool AutoFocus 2.

38.8.2.5 Verification

Today, we are at the point where we do verification for industrial-type software systems in cars. An ambitious approach is provided by the Verisoft project (see Ref. 9), where we verify as a demonstration the automatic emergency call function, which is found in cars today, in detail. The example of the emergency call function is verified by modeling it together with the used infrastructure like FlexRay, the real-time operating system OSEKtime, and the code running on processors in detail. In the Verisoft project, we use AutoFocus as the modeling tool and Isabelle as the verification engine.

38.8.3 Tool Support

We have developed prototyping tools like AutoFocus for design and its enhancement AutoRaid (see Ref. 33) for requirements engineering (see Refs. 1, 2, and 3).

Figure 38.3 shows a screenshot of the AutoFocus tool, with extensions like AutoFlex especially targeting the development of embedded control applications.

38.9 Conclusion

In fact, software in cars is one of the big challenges and at the same time one of the interesting fields of research for the software engineering community. Much can be gained in this area, much can be learned in this area, and much can be contributed by well-targeted research. We consider automotive software engineering as one of the very challenging fields where we have the chance to get into a mutually fruitful dialog between application domain and software engineering experts.

Acknowledgment

I am grateful to a number of people, including Martin Rappl, Doris Wild, Christian Kühnel, Jan Jürjens, Alexander Pretschner, Christian Salzmann, Hans Georg Frischkorn, Bernhard Schätz, Florian Deissenböck as well as engineers from BMW, Daimler Chrysler, Ford, Honda, Porsche, Audi, Volkswagen, ESG, Validas AG, Bosch, and Siemens VDO for stimulating discussions on topics of automotive software engineering.

References

1. AutoFOCUS. <http://autofocus.in.tum.de>, 2005.
2. AutoFOCUS 2. <http://www4.in.tum.de/~af2/>, 2006.
3. AutoRAID. <http://www4.in.tum.de/~autoraid/>, 2005.
4. AUTOSAR Consortium. www.autosar.org, 2005.
5. AUTOSAR Development Partnership. www.autosar.com.
6. A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, D. Ziegenbein, Auto-MoDe—Notations, Methods, and Tools for Model-Based Development of Automotive Software. In: Proceedings of the SAE 2005 World Congress, Detroit, MI, Apr. 2005. Society of Automotive Engineers.
7. M. Bloos, Echtzeitanalyse der Kommunikation in KfZ-Bordnetzen auf Basis des CAN Protokolls, Dissertation TU, München, 1999.
8. BMW Group, Annual Report, 2004.
9. J. Botaschanjan, L. Kof, Ch. Kühnel, M. Spichkova, Towards Verified Automotive Software. ICSE, SEAS Workshop, St. Louis, Missouri, May 21, 2005.
10. M. Broy, Automotive Software Engineering. Proc. ICSE 2003, pp. 719–720.

11. M. Broy, Modeling Services and Layered Architectures. In: H. König, M. Heiner, A. Wolisz (Eds.), *Formal Techniques for Networked and Distributed Systems. Lecture Notes in Computer Science 2767*, Springer, Berlin, 2003, pp. 48–61.
12. M. Broy, Service-Oriented Systems Engineering: Specification and Design of Services and Layered Architectures—The JANUS Approach. In: M. Broy, J. Grünbauer, D. Harel, T. Hoare (Eds.), *Engineering Theories of Software Intensive Systems*, Marktoberdorf, August 3–15, 2004, Germany, NATO Science Series, II. Mathematics, Physics and Chemistry—Vol. 195, Springer, Berlin/Heidelberg.
13. M. Broy, F. Deißeböck, M. Pizka, A Holistic Approach to Software Quality at Work. 3rd World Congress for Software Quality (3WCSQ) J. Dannenberg, C. Kleinhans, *The Coming Age of Collaboration in the Automotive Industry*, Mercer Management Journal 18: 88–94, 2004.
14. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (Eds.), *Model-Based Testing of Reactive Systems*, Advanced Lectures, Springer, Berlin, 2005.
15. M. Broy, A. Pretschner, C. Salzmann, T. Stauner, Software-Intensive Systems in the Automotive Domain: Challenges for Research and Education. In: SP-2028, *Vehicle Software and Hardware Systems*, SAE International, April 2006, <http://www.sae.org/technical/books/SP-2028>.
16. M. Broy, A. Rausch, Das Neue V-Modell XT, *Informatik Spektrum A* 12810, Band 28, Heft 3, June 2005.
17. M. Broy, K. Stølen, Specification and Development of Interactive Systems—FOCUS on Streams, Interfaces and Refinement, Springer, Berlin/Heidelberg, 2001.
18. A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, D. Wild, Concretization and Formalization of Requirements for Automotive Embedded Software Systems Development, In: K. Cox, J.L. Cybulski, L. Nguyen, J. Lamp, R. Smith (Eds.), *The Tenth Australian Workshop on Requirements Engineering (AWRE)*, Melbourne, Australia, 2005, pp. 60–65.
19. E. Geisberger, Requirements Engineering eingebetteter Systeme—ein interdisziplinärer Modellierungsansatz, Dissertation TU München, 2005.
20. K. Grimm, Software Technology in an Automotive Company—Major Challenges. *Proc. ICSE*, 2003, pp. 498–505.
21. B. Hardung, T. Kölzow, A. Krüger, Reuse of Software in Distributed Embedded Automotive Systems. *Proc. EMSOFT'04*, pp. 203–210, 2004.
22. H. Heinecke, Automotive System Design—Challenges and Potential. *Proc. DATE'05*.
23. D. Herzberg, M. Broy, Modeling Layered Distributed Communication Systems. *Applicable Formal Methods*. Springer, Berlin/Heidelberg 17(1), 2005.
24. M. Kuhrmann, D. Niebuhr, A. Rausch, Application of the V-Modell XT—Report from A Pilot Project. In: M. Li, B. Boehm, L.J. Osterweil (Eds.), *Unifying the Software Process Spectrum*, International Software Process Workshop, SPW 2005, Beijing, China, May 25–27, pp. 463–473, Springer, Berlin/Heidelberg, 2005.
25. A. Pretschner, C. Salzmann, T. Stauner, SW Engineering for Automotive Systems at ICSE'04. *Software Engineering Notes* 29(5): 5–6, 2004.
26. A. Pretschner, C. Salzmann, T. Stauner, SW Engineering for Automotive Systems at ICSE'05. *Software Engineering Notes*, 30(4): 79, 2005.
27. A. Rausch, Ch. Bartelt, Th. Ternité, M. Kuhrmann, The V-Modell XT Applied—Model-Driven and Document-Centric Development. In: 3rd World Congress for Software Quality, Volume III, Online Supplement, 2005, pp. 131–138.
28. S. Rittmann, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, D. Wild, Integrating Service Specifications on Different Levels of Abstraction, In: IEEE. (Ed.), *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, IEEE, 2005.
29. Robert Bosch GmbH. CAN Specification Version 2.0, 1991.
30. C. Salzmann, H. Heinecke, M. Rudorfer, M. Thiede, T. Ochs, P. Hoser, M. Mössmer, A. Münnich. Erfahrungen mit der technischen Anwendung einer AUTOSAR Runtime Environment, VDI Tagung Elektronik im Kraftfahrzeug, Baden-Baden, 2005.

31. C. Salzmann, T. Stauner, *Automotive Software Engineering. Languages for System Specification: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specification from FDL'03*, pp. 333–347, Kluwer, Norwell, MA, USA, 2004.
32. B. Schätz, *Interface Descriptions for Embedded Components*. In: O. Niggemann, H. Giese (Eds.), *Postworkshop Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-time Systems (OMERS)*, HNI-Verlag, Schriftenreihe, Paderborn, Band 191, 2005, Heinz-Nixdorf-Institut, Universität, Paderborn.
33. B. Schätz, A. Fleischmann, E. Geisberger, M. Pister. *Model-Based Requirements Engineering with AutoRAID*. In: *Informatik 2005*, Springer, Berlin/Heidelberg, 2005.
34. T. Stauner, *Compatibility Testing of Automotive System Components*. 5th International Conference on SW Testing (ICSTEST), Düsseldorf, 2004.
35. D. Wild, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, *An Architecture-Centric Approach towards the Construction of Dependable Automotive Software*, SAE Technical Paper Series 2006, Detroit, 2006.

39

Real-Time Data Services for Automotive Applications

Gurulingesh Raravi
Indian Institute of Technology

Krithi Ramamritham
Indian Institute of Technology

Neera Sharma
Indian Institute of Technology

39.1	Introduction	39-1
39.2	Real-Time Data Issues in Automotive Applications	39-2
39.3	Adaptive Cruise Control: An Overview	39-5
	Real-Time Issues in ACC	
39.4	Our Goals and Our Approach	39-7
39.5	Specifics of the Dual Mode System	39-8
39.6	Specifics of the Real-Time Data Repository	39-10
	Task Models • Scheduling of On-Demand Tasks	
39.7	Robotic Vehicle Control: Experimental Setup	39-12
39.8	Results and Observations	39-13
	Basic Experiments • Two-Level Real-Time Data Repository Experiments • Mode-Change Experiments	
39.9	Related Work	39-17
39.10	Conclusions and Further Work	39-18

39.1 Introduction

Computer-controlled functions in a car are increasing at a rapid rate, and today's high-end vehicles have up to 80–100 microprocessors implementing and controlling various parts of the functionalities [1]. Some of the features currently controlled by microprocessors include electronic door control, air conditioning, cruise control (CC), adaptive cruise control (ACC), antilock braking system (ABS), etc. It is predicted that 80% of the automobile innovation will be in the area of electronics, controls, and software. Automobile industries are aiming toward 360 degrees of sensing around the vehicle leaving no blind zones for the driver. Sophisticated features such as collision avoidance, lane keeping, and *by-wire* systems (steer-by-wire, brake-by-wire) are on the verge of becoming a reality. The number of sensors required to sense the environment for all these sophisticated applications is also increasing at a rapid rate and so is the amount of real-time data that need to be managed.

Applications such as *Collision Avoidance* or *ACC* are safety enhancers that deal with critical data having stringent timing requirements on freshness and involving deadline bound computations [2]. Therefore, there is a need for effective real-time data services to handle the data to provide real-time guarantees for such applications.

The practice of implementing each application as an independent *black-box* excludes any possibility of sharing the applications among the microprocessors. Increasing the microprocessors in relation to individual applications will be a difficult proposition both in terms of cost and integration complexity. Hence, the microprocessors must be effectively used to make *fully electronically controlled vehicle* a reality.

Minimizing the number of microprocessors in an automobile by effectively using them and at the same time providing real-time guarantees to the safety-enhancing applications by providing better real-time data services is our goal. Most automotive applications that are safety-enhancing tend to be control applications (a particular target area being that of by-wire controls). These are typically developed using high-level functional modeling languages such as Simulink and Stateflow (SL/SF). These models are interconnected networks of computational blocks that exchange information using signals. One of the challenging steps is the refinement of the high-level functional models into computational tasks that can be executed on a distributed platform. This involves the following steps:

1. Decomposition of functional model into tasks
2. Distribution of tasks to different computing nodes in the system
3. Generation of schedules for each of the computing nodes for executing their assigned tasks
4. Assignment of slots on the communication bus to communicating tasks

Three important requirements for such high-integrity applications are:

- Correctness (both functional and timing) of the model to be preserved across the refinement from functional models to computational tasks
- Efficient utilization of resources
- Stability and performance analysis of the controller

The focus of this chapter is on efficient utilization of resources by providing real-time data services without affecting the performance and safety of such applications. To this end, we discover efficient methods to allow sharing of multiple applications among the processors and at the same time guarantee real-time responses. This would be a critical need, for the cost-effective development of fully electronically controlled vehicle in the future and, in this chapter, we have attempted to address this need. The refinement of the high-level functional models into computational tasks and stability analysis of the designed controller is planned in the future work. We have chosen ACC, a safety-enhancing application as the case study for our approach.

In this chapter, we have (i) identified the data-related issues in automotive applications, (ii) identified the data and the task characteristics of ACC and shown how to map them on a real-world (robotic vehicle) platform, (ii) facilitated a real-time approach toward designing the ACC by the application of mode-change and real-time data repository concepts for reducing the CPU capacity requirement, and (iii) provided the scheduling strategies to meet the timing requirements of the tasks. The experiments carried out show that CPU requirements can be effectively reduced without compromising the real-time guarantees for safety-enhancing applications.

The rest of the chapter is structured as follows. Section 39.2 briefs about the real-time data issues in automotive applications. Section 39.3 briefs about the features of ACC and the current practice followed to develop this application. The issues that need to be addressed to provide computational support for the ACC applications are described in Section 39.4. The mainstay of our approach, namely, a two-level real-time data repository and mode-change design are detailed in Sections 39.5 and 39.6. The experimental setup is described in Section 39.7 and the results of the evaluations from the prototype model is shown under Section 39.8. Section 39.9 presents the related work followed by conclusions and future work.

39.2 Real-Time Data Issues in Automotive Applications

There is a strong need in the automotive industry for better tool support to handle the increasing functionality and complexity of computer-controlled software. The amount of data to be managed in

a vehicle has increased drastically owing to the increasing law regulations, diagnosis requirements, and complexity of electronics and control software. There are various ways to characterize data items that automotive applications deal with depending on:

- Methods of their update
 - *Continuous data items*: They reflect values from an entity, such as a sensor, that changes continuously in the external environment.
 - *Discrete data items*: They are updated at discrete intervals; for example, cruise control set speed is updated when the driver increases or decreases it using a manual switch.
- Source of origin
 - *Base data items*: They are sensor values reflecting an external environment and can be of the type either *continuous* or *discrete*.
 - *Derived data items*: They are actuator values or intermediate values used by several computations; for example, a vehicle's longitudinal speed based on wheel sensors' angular speed.
- Their timing requirements
 - *Safety critical (SC)*: The access and processing of these data items will have timing requirements whose violation might lead to catastrophe, e.g., leading vehicle's distance from the host vehicle for ACC application.
 - *Non critical (NC)*: They will either have no timing requirements or soft timing requirements which when failed to meet will not lead to a catastrophe but meeting them will add some value to the system. For example, logging driver actions and vehicle state by event-data recorder for offline processing to rate the driver.

Different preventive safety functions are now introduced in vehicles to assist the driver for controlling the vehicle in a better way and to reduce the risk of accidents. The stringent timing requirements on the data items handled by such applications have to be met for realizing such applications. Hence, the time has come for automotive researchers to apply much of the research carried out in real-time (database) systems to provide efficient data services to effectively handle the data.

In providing real-time data services to guarantee the required timeliness of data and tasks, there are various issues that must be considered. Below is a discussion of some of the concerns that have been the subject of research in real-time systems that automotive researchers are looking at for developing safety applications.

Data, task, and system characteristics. The need to maintain coherency between the state of the environment and its reflection in the database or data repository leads to the notion of temporal consistency. This arises from the need to keep the controlling system's view of the state of the environment consistent with the actual state of the environment. Since many vehicular functionalities, such as lane changing ACC and collision avoidance, are safety enhancers, a real-time solution must maintain not only the logical consistency of data and the tasks handling them, it must also satisfy the task timing properties as well as data temporal consistency. *Task timing constraints* include deadlines, earliest start times, and latest start times. Tasks must be scheduled such that these constraints are met. *Data temporal consistency* imposes constraints on how old a data item can be and still be considered valid.

Task scheduling. Task scheduling techniques that consider task and data characteristics have been the major part of research in real-time (database) systems. Real-time data base researchers have considered the inherent trade-offs between quality of data versus timeliness of processing. If logical consistency is chosen, then there is the possibility that a data item may become old, or that a task may miss a deadline. On the contrary, if temporal consistency is chosen, the consistency of the data or of the task involved may be compromised. It is often not possible to maintain both the logical consistency of a database and the temporal consistency. Therefore, there must be a trade-off made to decide which is more important. In dynamic systems, where it is not possible to know and control the real-time performance of the database, it may be necessary to accept levels of service that are lower than optimal. It may also be necessary to trade off the quality of the data to meet specified timing constraints.

Tracking dynamically varying data. Dynamic data is characterized by rapid changes and the unpredictability of the changes, which makes it hard to effectively track at predetermined times considering the computational resource limitations in automobiles. The validity interval of such data items is not necessarily fixed during the execution of the system. To have a fixed sampling time as in existing systems requires a worse-case design, leading to over-sampled data and ineffective use of the computational resource such as a processor.

Handling of mode-specific data and tasks. Processes controlled by hard real-time computer systems typically exhibit mutually exclusive phases/modes of operation and control [3]. The change in the system's mode of operating might change the task sets and the data processed by them, affecting task characteristics such as their timing requirements. A design process that requires all modes of a system to be considered and designed once leads to poor utilization of computational resources, scheduling overhead. For these reasons supporting different modes of operation is of major concern.

Distribution of data and processing. Several applications are obviously distributed in nature to satisfy fault tolerance requirements and localized availability of data. Sensing, computation, and actuation tasks are not located on a single computing resource owing to the nature of distribution of physical entities that carry out these activities. Rather, they are distributed, and may require that the real-time data be distributed as well. Issues involved with distributing data include data replication, replication consistency, and distributed concurrency control. When real-time requirements are added, these embedded systems become much more complex.

Dynamic data dissemination. With the technological advancements in automotive electronics, information and mobile communication technology, many data-intensive applications are emerging, e.g., Vehicle-to-Vehicle (V2V) communication such as cooperative adaptive cruise control (CACC) and Vehicle-to-Infrastructure (V2I) such as real-time navigation. Many of these applications deal with large amount of real-time data sensed to capture the environment status. Access to these data items and their processing might be associated with soft or even hard deadlines depending on the application under consideration. For example, vehicles equipped with CACC communicate messages to each other continuously to avoid probable collision at blind crossings, and a real-time traffic navigation system should continuously track the vehicle till it reaches its destination after receiving a request for the best path for its destination. The data involved in these kind of applications will have timing requirements and need to be satisfied to provide better service and to even avoid catastrophes in certain cases. Hence, there is a need for effective data dissemination methods for meeting the timing requirements and to provide accurate information in both V2V and V2I applications.

Sensor data fusion. The automobile industry is moving toward a large number of sensors connected through a network to achieve the goal of 360 degree sensing leaving no blind zones for the driver around the host vehicle. Various technologies are under consideration: sensors, radars, vision system, and the combination of these. The challenge here is environment recognition and reconstruction so as to be able to alert the driver of possible threats and prevent collisions. The need for such systems exists in many applications involving tasks in which timely delivery of information streams is of critical importance to avoid possible collisions. The obstacle or possible threat recognition techniques require a great deal of accuracy and reliability since an erroneous application of emergency braking caused by false alarms makes it difficult for the driver to accept such software-controlled safety applications. Automotive researchers would not deny that such actions caused by false alarms might even result in a catastrophe. Hence, there is a need to study accurate and *real-time* fusion techniques to fuse the heterogeneous data coming from different sensors distributed across the vehicle to identify the correct potential threats well in time.

Supervisory control module. Computer-controlled functions in a car are increasing at a rapid rate. Some of the features currently controlled by microprocessors include electronic door control, air conditioning, cruise control, ACC, antilock braking system, etc. Sophisticated features such as collision avoidance, lane keeping, curve speed control, and *by-wire* systems (steer-by-wire, brake-by-wire) are on the verge of becoming a reality. These applications will use sensors data to compute the desired actions and issue commands to brake, throttle, and steering subsystems to carry out them. Its very difficult to (i) avoid conflicts in commands issued by different applications, (ii) properly distribute commands between these

subsystems, (iii) coordinate each feature to improve overall vehicle performance, and (iv) properly deal with signal noise, latency, multiple dynamical rates at which different applications issue commands. Hence, it is necessary to have a supervisory control module to handle these issues effectively to enhance vehicle performance without affecting the safety requirements.

Fault tolerant architecture. Without doubt, fault tolerance is one of the major challenges that must be met for the design of dependable or safety critical electronic systems, such as *drive-by-wire* applications. These applications are obviously distributed in nature to satisfy fault tolerance requirements and localized availability of data. A reliable architecture (such as time triggered architecture [TTA] [4]) is required to implement such fully fault-tolerant real-time systems. These systems also require a communication service (such as time triggered protocol [TTP] [5] or FlexRay, www.flexray.com) that transports messages from one node to another with nearly constant delay and minimal jitter.

39.3 Adaptive Cruise Control: An Overview

ACC is an intelligent application that automatically adjusts vehicle (a.k.a. *following vehicle*) speed to maintain a *safe distance* from the vehicle moving ahead in the same lane (a.k.a. *leading vehicle*). When there is no vehicle ahead, it tries to maintain the *safe speed* set by the driver. The *safe distance of separation* (DoS) that needs to be maintained from the leading vehicle is a function of the *host vehicle velocity* and a driver specified parameter, *timegap* (a.k.a. time separation) and it is given by [6]

$$V_h * T_g + \delta \quad (39.1)$$

where V_h is the host vehicle speed, T_g the timegap, and δ the separation distance when the leading vehicle is at a standstill. This ensures that safety distance requirement is not violated under the extreme stopping condition where the leading vehicle can come to a halt instantaneously.

ACC, also called next generation Cruise Control, uses forward-looking radar to detect the speed and distance of the leading vehicle. It continuously monitors the host vehicle's speed, the status of cruise control, the driver's inputs, and the leading vehicle's speed and distance. Using all the data, ACC does intelligent computations to determine the desired acceleration of the host vehicle and achieves the safe DoS with the help of throttle and braking system. Actions taken by the controller are displayed on the driver console to keep the driver informed about the same.

The various components of the ACC system are shown in Figure 39.1 and are described below.

Sensors and fusion algorithms: Wheel sensors are used to measure the angular velocity of the wheels. The road-tire friction is also estimated using this data. Sensors are also used to know the current position of the throttle and brake pedals. The sensor fusion algorithms help to get the relevant and accurate information

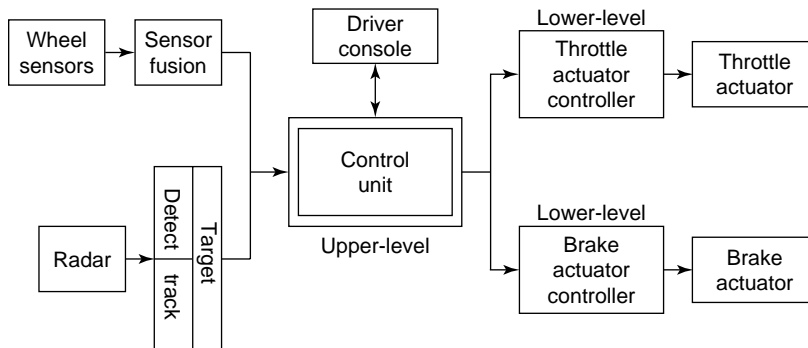


FIGURE 39.1 Block diagram of ACC components.

from the sensors. For instance, they calculate the vehicle speed by fusing data coming from individual wheel sensors i.e., rotational speed is converted into longitudinal speed.

Radar and filter algorithms: Radar is used to detect the vehicles ahead and measure the relative velocity and distance between the leading vehicle and the host vehicle. The algorithms used here help to identify and track the most relevant vehicle from the set of vehicles detected by the radar. It helps the controller to compute necessary action by providing the DoS and speed of the relevant leading vehicle.

Driver console: This unit takes care of the communication to and from the driver. The driver should be aware of the environment and action being taken by the ACC feature.

Controller: The controller maintains up-to-date information about the environment with the help of sensors and radar. It performs necessary control computations to take appropriate actions to maintain the safe distance from the leading vehicle. It has a collection of subtasks that run on an embedded real-time operating systems (RTOS) such as RTLinux and OSEK. These subtasks have information about the environment. The subtasks also interact with other subsystems, and decide actions to be taken, and commands the actuators to perform the required actions. The controller is typically designed in a hierarchical manner consisting of an *upper-level controller* and a *lower-level controller*. The upper-level controller computes the desired acceleration based on the values of the current velocity, acceleration, and measured DoS. The lower-level controller in the system is a subloop controller that manipulates the brake and engine outputs to achieve the desired acceleration/deceleration. At this lower level, an adaptive control algorithm is required that closely monitors the brake and engine control units to achieve the desired acceleration value calculated by the upper-level controller. The upper-level controller has been the focus point for most of the research work in ACC.

The upper-level controller can operate in one of the two basic modes:

- *Distance control mode:* When there is a leading vehicle ahead, controller tracks its DoS and speed and maintains the safe DoS as set by the driver.
- *Speed control mode:* When there is no leading vehicle, controller maintains the safe speed set by the driver.

We exploit the difference between the two modes for efficiency of processing requirement.

Actuators: The vehicle speed is controlled by brake and throttle actuators. They are electronically controlled components driven by the commands from the lower-level controllers for the requested acceleration computed by the high-level controller.

39.3.1 Real-Time Issues in ACC

One of the challenges faced when trying to maintain safe distance is that both the vehicles in the environment are moving. The task of the control system installed in the host vehicle (a.k.a. follower) is to continuously track the leading vehicle (a.k.a. leader) and adapt its velocity accordingly. The tracking rate should be fast enough to have accurate information about the leading vehicle and slow enough to ensure that the system is not overloaded with redundant operations, i.e., operations that do not lead to any change of control actions. The goal is to achieve correct functionality and meet timing requirements, while using the resources optimally. However, there are several factors that can make it very difficult to achieve these requirements. These include the rate at which vehicles are approaching or separating, vehicles cutting in from adjacent lanes to the host car's lane, and weather conditions. To illustrate the timing requirements in the system, consider a host vehicle moving with a velocity of 20 m/s and it detects a vehicle ahead moving with a uniform velocity of 15 m/s, at a distance of 50 m. Let us assume that the driver has set a timegap to be maintained as 2 s. ACC has to adapt its velocity from 20 to 15 m/s by the time the DoS reduces to 30 m. The calculation of desired speed, acceleration, and the adjustment of throttle or brake should be completed within this time to maintain the safe distance. This practical scenario shows that the system must be able to work with timing constraints to guarantee that all the data collection and its processing, as well as subsequent decisions and actions happen in time.

In current ACC systems, all the sensors sense the data (leading vehicle speed, distance, and host vehicle velocity) *periodically*. Also, these sensors provide the *raw data* that should be processed to convert them into application-specific data (or meaningful information) known as *derived data*. The raw items reflect the external environment and can either change continuously (e.g., wheel speed sensor) or change at discrete points in time (e.g., selecting safe speed). The derived items are derived from raw items and other derived items, i.e., each derived item d has a read set denoted as $R(d)$ that can have members from both the raw and the derived set [7]. For instance, the host velocity is derived from the angular velocity of the wheels obtained from four wheel sensors. In this case, the angular velocity is raw data item and the host velocity is derived data item. The tasks that process raw data to get derived items are also run periodically in the existing systems. The periodicities of all the tasks in the system are set to handle the worst-case scenario.

39.4 Our Goals and Our Approach

Our goal is to provide real-time data services that address the following:

Effective tracking of dynamically varying data. A data item reflects the status of an entity only for a limited amount of time. When this time expires, the data item is considered to be stale and not valid anymore. This validity interval of a data item is not necessarily fixed during the execution of the system. For instance, consider a leading vehicle that accelerates for a certain amount of time and then travels with uniform velocity. The validity interval (and hence the sampling period) for the data item *leading distance* will be small when the leading vehicle is accelerating and it can be large when it is moving with a uniform velocity. To have a fixed sampling time as in existing systems requires a worse-case design, leading to over-sampled data and inefficient use of the processor.

Timely updates of derived data. The data derivation should be complete before the derived item's read set becomes invalid. The derived item needs to be updated only when one or more data items from its read set changes more than a threshold value. For example, the host velocity is derived only when the angular velocity of wheels changes more than the threshold value. In existing systems, the derived values are updated periodically causing unnecessary updates. This will again lead to over sampling and hence inefficient use of the processing power.

Handling mode-specific task sets. Processes controlled by hard real-time computer systems typically exhibit mutually exclusive phases of operation and control [3]. ACC system performs different tasks while following a close vehicle when compared to following a vehicle which is far away. For instance, we can have a task that determines how much time is left before the safety criterion is violated when the DoS is small, to help the controller to take alternate actions such as warning the driver or changing the lane. Similarly, we can have a task that can adjust the driver-set parameters—safe speed and timegap—depending on the weather and road conditions when the DoS is large. The task characteristics such as periodicity may also vary in different modes. Such changes in the modes of operation affect the task timing requirements, their dependencies, and execution times. In current approaches, all the tasks are executed throughout. This leads to poor CPU utilization, scheduling overhead, and contradicts the principles of modularization.

Our approach to address the above-mentioned issues by providing two real-time data services that exploit two well-known design techniques from real-time system domain: mode-change protocol and real-time update protocols. Both the approaches help to design the application that leads to effective utilization of the CPU capacity by understanding the needs of the system's task and data characteristics. The mode-change protocol is a *task-centric* approach that allows the designer to vary the task sets and characteristics over a period of time. At any point of time, the system will have and schedule only the necessary tasks without wasting the CPU capacity on unnecessary tasks. The real-time data repository model is a *data-centric* approach that decides the task characteristics from the freshness requirements of the base and derived data items. Certain periodic tasks from the mode-change approach are made aperiodic to facilitate the *on-demand (OD) updates to data items*.

39.5 Specifics of the Dual Mode System

Processes controlled by hard real-time computer systems typically exhibit mutually exclusive phases/modes of operation and control. A mode change will typically lead to either:

- Adding/deleting a task or
- Increasing/decreasing the execution time of a task or
- Increasing/decreasing the frequency of execution of a task

For instance, ACC performs different tasks while *following* a *close* leading vehicle compared to one that is *far*. In different modes, we can have the sensing tasks execute at different frequencies to deal with dynamically varying data and we can have different set of tasks active in different modes. Hence, we do not need to have all the tasks active at all the times. The design of ACC application with this approach requires answers to the following questions: (i) How many modes should the design have? (ii) What condition/event should trigger mode change in the system? (iii) When can we switch modes and how much time can a mode-change operation take? and (iv) How should the tasks be scheduled to meet their timing requirements? We have explained below how these issues are handled while designing ACC application.

(i) *Two mutually exclusive phases of operation for ACC*

Noncritical mode (NC mode): In this mode, the environment status does not change rapidly. For instance, when the host vehicle is following a leading vehicle at uniform velocity, the parameters like DoS, host velocity, leading vehicle velocity do not change rapidly. The rate at which the parameters of the system change decides the periodicity of the tasks. The sensor tasks in this mode can execute less frequently.

Safety critical mode (SC mode): In contrast to NC mode, here the system parameters vary rapidly. For example, consider the case when the leading vehicle is applying maximum brake (say 4.9 m/s^2) and the host vehicle under ACC is decelerating (say 2 m/s^2). In this case, the DoS between the two vehicles is reducing at a rapid rate. Hence, the radar task that senses this separation should begin to execute more frequently to give the controller fresh data, helping it to determine the right decision at the right time. The system is classified into these two modes of operation based on the following parameters:

- Distance between the two vehicles (can take the values FAR, NEAR, FOLLOW)
- Rate of change of distance (RoD) (can take the values Incr-Fast, Incr-Slow, Decr-Fast, Decr-Slow)

Task sets in different modes are shown in Table 39.1. The tasks in NC mode, namely, FrictionT, AdaptT, and EDiT perform noncritical operations such as determining road-tire friction, adapting the driver set parameters depending on weather conditions, and logging the data for offline processing, respectively. Similarly, the tasks in SC mode, namely, TimeLeftT, AdjLaneT, and SuggestT carry out critical operations such as determining the time left to avoid collision, sensing adjacent lanes to determine if the lane can be changed to avoid collision, and suggesting the driver to take some alternative actions when it becomes difficult for ACC to take control of the situation, respectively. The common tasks WheelT and SpeedT are sensor-related tasks and RadarT, DistT, and LeadVelT are radar tasks. The SpeedT task determines the host vehicle's speed using the data provided by the wheel sensor task, WheelT. The tasks DistT and LeadVelT calculate the leading vehicle DoS and velocity, respectively, making use of the data provided by the radar task, RadarT. The CruiseT, AccT and BrakeT, ThrottleT are upper-level and lower-level controller tasks and SwitchT is the task that performs mode change. DriverT monitors the driver inputs and ExceptionT is the task to handle exceptions in the system. All these tasks are *periodic* in nature. The regions FAR, FOLLOW, and NEAR are characterized in Figure 39.2. The radar detection range is given by the dotted curve. *Safe_Dist* is calculated as per the Equation 39.1 and *Follow_Dist* = *Safe_Dist* + Δ , where Δ is additional gap for improved safety margins. *Safe_Dist*⁺ is given by *Safe_Dist* + δ , which gives enough time for the controller to raise the alarm and for the driver to take over the control. The vehicle will lock-on to a leading vehicle and follow it when the DoS is equal to the *Follow_Dist*. The regions NEAR and FAR are used to determine the mode of operation (SC versus NC) as shown in Table 39.2. The *Foll_Dist*⁺

TABLE 39.1 Task Sets in Different Modes of ACC System

Mode	Task Set
NC only	WeatherT, FrictionT, AdaptT, EDrT
SC only	TimeLeftT, SuggestT, AdjLaneT
Both modes	WheelT, SpeedT, CruiseT, AccT, RadarT, LeadVelT, DistT, DriverT, BrakeT, ThrottleT, SwitchT, ExceptionT

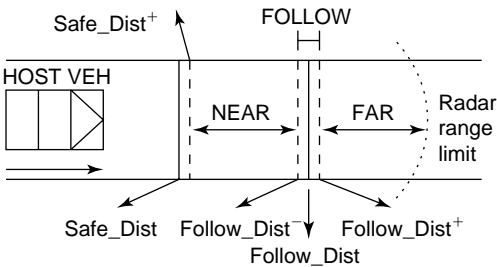


FIGURE 39.2 Environment partitioned into regions.

TABLE 39.2 The State of the System in Different Modes

LeadDist	RoD	Mode
FAR	Decr-Fast	SC
FAR	Incr-Fast	NC
FAR	Decr-Slow	NC
FAR	Incr-Slow	NC
NEAR	–	SC
FOLLOW	–	Retain mode

and $Foll_Dist^-$ are used to create the FOLLOW region whose importance is explained when switching condition is described.

(ii) *Details of the modes.* Intuitively, we should switch from NC to SC mode, the moment the leading vehicle enters the NEAR region. The controller should resume NC mode as soon as the leading vehicle accelerates to the FOLLOW region. The switching condition is satisfied more frequently leading to rapid mode switching, a phenomenon called *chattering*. Since the controller should have fresh information about the environment, considering only the DoS for switching modes will not yield good results, e.g., when the leading vehicle is in the FAR region and decelerating fast. To tackle this case, we also consider the RoD as one of the parameters while deciding the mode-switch condition. Finally, to avoid the chattering phenomenon, FOLLOW region is used as hysteresis. The decision on the current mode of the system is taken based on two parameters DoS and the RoD and the change in their values triggers the mode-change phenomenon. Table 39.2 shows all the possible situations for the system to operate in each of the modes.

The system should satisfy the following conditions depending on its mode of operation. In SC mode:

$$\begin{aligned} & (Safe_Dist^+ < Curr_Dist \leq Follow_Dist^-) || \\ & (Follow_Dist^+ < Curr_Dist \leq Radar_Dist \&\& RoD == Decr_Fast) || \\ & (Follow_Dist < Curr_Dist \leq Follow_Dist^+ \&\& Curr_Mode == SC) \end{aligned}$$

In NC mode:

$$\begin{aligned} & (Follow_Dist^+ < Curr_Dist \leq Radar_Dist \&\& RoD \neq Decr_Fast) || \\ & (Follow_Dist < Curr_Dist \leq Follow_Dist^+ \&\& Curr_Mode == NC) \end{aligned}$$

(iii) *Switching modes.* A mode-change request is generated from either the radar task, when DoS parameter satisfies the condition for mode change, or another periodic task, which tracks the RoD to satisfy the condition. Once the mode-switch condition is satisfied, mode change needs to be carried out by a task. This process involves deleting the tasks in the current mode, allocating stack to the new tasks and creating the new tasks ensuring schedulability at any given point of time throughout this process. The periodic task *SwitchT* performs these operations in our implementation. The mode-switch operation is initiated upon the termination of the task that makes the mode-switch condition true.

Mode-change delay, defined as the delay between the time at which mode change is requested and the time at which all the tasks that need to be deleted have been deleted and their allocated processor capacity becomes available, should be small. As we initiate the mode-change operation once the current task finishes its execution, the delay in reclaiming the processor capacity is bounded by the period of low-priority task in the system. In rate monotonic algorithm (RMA), this is given by longest period [8].

(iv) *Scheduling tasks in different modes.* The modes in a system are characterized by a number of active tasks. The tasks and their characteristics (periodicity, WCET) are known *a priori*. Hence, static priority scheduling is the obvious choice for the system with different modes. RMA is used to schedule the tasks in our implementation.

39.6 Specifics of the Real-Time Data Repository

In this section, we describe our real-time data repository model, which consists of two levels of data store. The concept of two levels of data store is motivated by (a) the presence of raw and derived data items and (b) the fact that a small change in raw data, i.e., sensor values might not affect the action of the ACC controller. As we discussed in Section 39.5, the wheel sensor task *WheelT* periodically senses the angular velocity of the wheels which is used by another periodic task *SpeedT* to determine the linear velocity of the vehicle. We realize that the periodic execution of the task *SpeedT* may be skipped in some cases. For instance, in a case where the host vehicle is following a leading vehicle moving with a uniform velocity maintaining the safe DoS, the host vehicle will travel with the same velocity until the leading vehicle starts accelerating or decelerating. In such cases, we can choose to skip the execution of *speedT* until we observe a considerable change in the value sensed by the task *WheelT*. Similarly, the tasks *DistT* and *LeadVelT*, which derive the DoS and leading vehicle velocity from the raw data provided by the task *RadarT*, need not execute periodically. These tasks should execute only when significant changes are observed in data collected by *RadarT*. This approach of avoiding the unnecessary updates in the system would result in a best utilization of CPU capacity. Our real-time data repository model is a data-centric approach, in which we explore the possibility of making some of the periodic tasks in our task set aperiodic, to reduce the number of unnecessary updates in the system. In this approach, we describe the temporal characteristics of the data items in the system, which help us decide the temporal characteristics of the updating tasks associated with the data item.

We use two levels of data store in our approach: *environment data repository* (EDR) and *derived data repository* (DDR) as shown in Figure 39.3. The upper- and lower-level controllers in the Figure 39.3 correspond to the controllers shown in Figure 39.1. EDR is an active entity, storing the data pertaining to the controlled ACC system. EDR contains base data items and the procedures for data derivation task. If smart sensors are employed in the vehicle, the process of data derivation can be done at these sensors themselves, causing some of the procedures implemented in EDR to be omitted. Collected sensor values might contain some noise. Techniques such as in-network aggregation and averaging may be

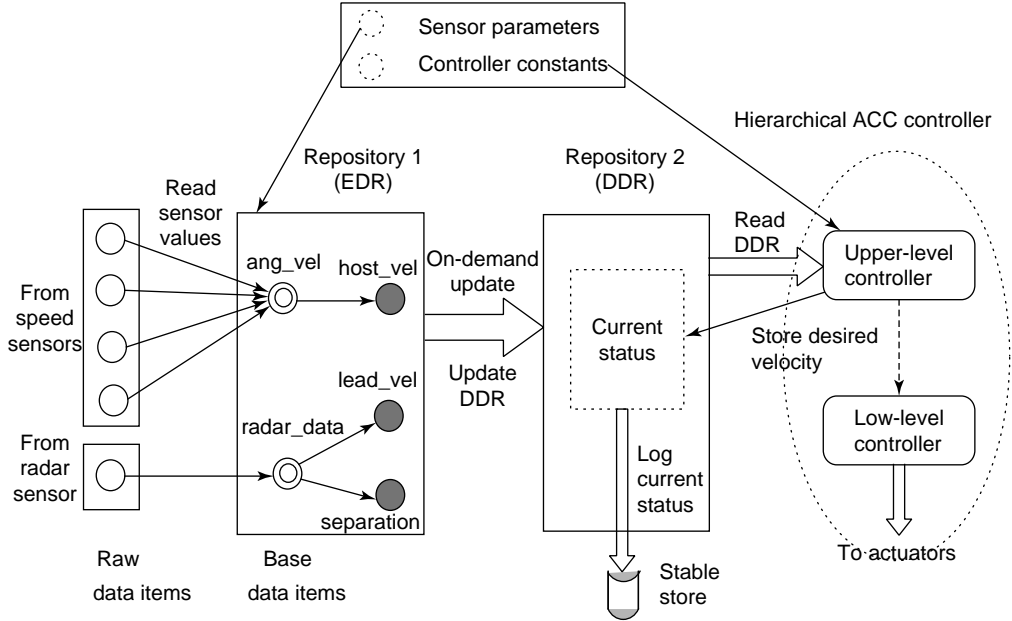


FIGURE 39.3 Real-time data repository for adaptive cruise control.

implemented in EDR to filter noisy values. The second repository DDR in the model acts as a global database for the system. ACC controller communicates with DDR to get the current values of vehicle parameters.

39.6.1 Task Models

As shown in Figure 39.3, circular nodes represent data items and arrows acting on these nodes represent tasks operating on data items. Effectively, each data item is associated with a value and a task manipulating the value. We identify the following classes of tasks in the system:

- *Sensor reading (SR) tasks*: Data collected from sensors are *temporally consistent*. They are valid if

$$CurrentTime - TS(d_i) \leq VI(d_i) \quad (39.2)$$

where $TS(d_i)$ and $VI(d_i)$ denote time stamp and validity interval of data object d_i , respectively. The tasks *WheelT* and *RadarT* read the values from wheel sensors and radar periodically and update EDR. These tasks have a known period T_i , a computation time C_i , and a relative deadline D_i equal to its period. The period of these tasks is determined by the validity interval of data objects associated with them.

- *On-demand (OD) update tasks*: The derived data items are calculated and updated in the DDR only when one of the data items from the read set, $R(d)$, changes more than the threshold value, δ_{th} . The tasks *SpeedT*, *DistT*, and *LeadVelT* fall under this category. The validity condition of data item d_i in DDR can be written as

$$|v_i(inEDR) - v_i(inDDR)| \leq \delta_{th} \quad (39.3)$$

where v_i is the value of the data item d_i and δ_{th} the threshold value that determines the frequency at which OD update tasks in the system are invoked.

- *Lower-level controller tasks*: They give command to the mechanical system of the vehicle to achieve the desired velocity. These tasks are performed by the hardware of our prototype model.

- *Other tasks*: They are the lower-priority tasks such as WeatherT and FrictionT for monitoring weather and road conditions. These tasks are executed only when there are no critical tasks pending in the system.

39.6.2 Scheduling of On-Demand Tasks

The second repository update task and upper-level controller tasks are modeled as OD aperiodic tasks. A well-known conceptual framework to guarantee the service on aperiodic tasks is the *aperiodic server technique*. This models the behavior of aperiodic tasks by reserving a share of processor bandwidth for each of the aperiodic tasks and associates a controlling server thread to maintain each reserved bandwidth. We use constant bandwidth server (CBS) [9] for scheduling aperiodic tasks. A CBS ($S = (C_s, T_s, B_s)$) is characterized by three parameters: maximum capacity (C_s), period (T_s), and bandwidth ($B_s = C_s/T_s$). Since we do not know the execution time (bandwidth requirement) of OD aperiodic (e.g., SpeedT) task *a priori* we implement CBS that adapts bandwidth dynamically using feedback control: we start with a high value for bandwidth, which reduces at each step and finally stabilizes to a constant value. The stabilized value for bandwidth is kept more than the actual required value. At each step, we calculate CBS scheduling error:

$$\epsilon = \text{CBS_Deadline} - \text{Task_Deadline}$$

where $\text{CBS_Deadline } d_i$ is set to $d_{i-1} + \text{server period}$. Task_Deadline is calculated using task rate. If R_i is the task rate and $a_{i,j}$ is the arrival time of j th instance of i th aperiodic task, task deadline will be: $1/R_i + a_{i,j}$. At each step, we adjust CBS bandwidth by changing server capacity C_s using the following feedback control law [10]:

$$\delta C_s = \frac{\epsilon}{T_s} * C_s \quad (39.4)$$

when ϵ is 0, the aperiodic task is guaranteed to respect its deadline. The objective of the system is to maintain ϵ near 0.

39.7 Robotic Vehicle Control: Experimental Setup

This section describes the implementation details of both hardware and software used to demonstrate the concept. Since our aim of this implementation is to prove the concept, we have implemented the essential parts of the system, abstracting out some real world factors. However, the implementation is scalable so that these factors can be accommodated as and when required. The robot on which ACC was implemented is shown in Figure 39.4 and had the following features:

- Obstacle detection (i.e., radar) range: 2 m
- Maximum speed: 0.50 cm/s
- Maintains path through white-line following
- Closed-loop controller(s)

The leading vehicle was simulated by manually moving an obstacle ahead of the robot and making the robot maintain the safe DoS from it. The experiments were performed using this manually controlled obstacle, maintaining constant DoS simulates the scenario of leading vehicle moving with uniform velocity. The case where leading vehicle varies its velocity over a period of time is simulated by increasing or decreasing the DoS. The robot was controlled by a PC running RTLinux . The PC performed all the computations and issued commands to the robot through the printer port. The standard printer port is used as the communication medium. The software was written using C on RTLinux platform. The logging functions were also written in C. The controller polled the data values from different sensors and performed computations to decide the action to be taken and drove the actuators to carry out the appropriate actions. The sensors were used to measure the host vehicle speed and the leading vehicle distance. The task structure and data items in real-time repository are shown in Figure 39.5. The data

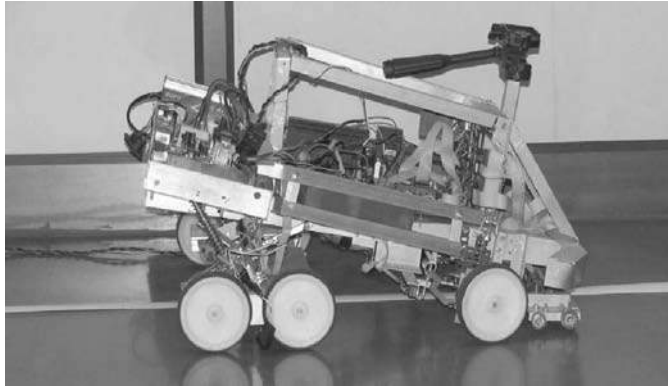


FIGURE 39.4 Experimental robotic vehicle platform.

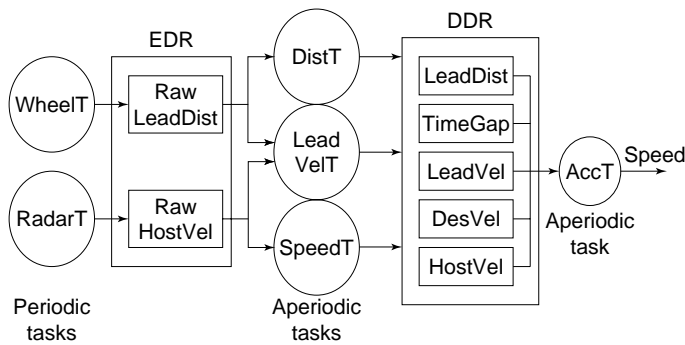


FIGURE 39.5 Task and data structure in real-time repository implementation.

items are represented by rectangular boxes and the tasks by circular boxes. The tasks updating EDR are periodic and those updating DDR are aperiodic in nature, ensuring *OD updates*.

The CBS is used to guarantee task completion before its deadline and we used CBS patch for RTLinux developed by Lipari et al. [11] for this purpose. The controller task (AccT) that reads the data items from DDR and computes the desired speed of the host vehicle also executes aperiodically for the above-stated reason. In SC and NC modes, the tasks and data items listed in Section 39.5 exist (common tasks) along with an additional task *SwitchT* to carry out mode-change operation. The tasks shown in Table 39.1 that exist in one of the two modes are implemented as dummy tasks with different periodicities. In the SC mode, the common tasks execute at double the speed as compared to the NC mode.

39.8 Results and Observations

Experiments were conducted in three stages. First experiments were meant to observe whether the robot was achieving the desired functionalities of the ACC system. This basic implementation did not incorporate either the mode-change or the real-time repository concepts. Second, experiments were conducted to test the behavior of the two-level real-time data repository design. These experiments were used to observe the reduction in the number of update tasks executed. The second-level tasks were executed only when necessary, as opposed to periodic execution in the initial design without the two-level repository. This saves the CPU processing power that can be effectively utilized to execute other lower-priority tasks in the system, if any. Finally, experiments were conducted to study the system behavior under mode-change design.

39.8.1 Basic Experiments

Four different tests were carried out to observe the system behavior by logging the host velocity and the DoS, with time. Figure 39.6 shows the velocity response of the host vehicle when there was no leading vehicle ahead and safe speed was set to 25 cm/s. We can observe that the controller is able to maintain the velocity in the range: 22 ± 3 cm/s.

The velocity response of the host vehicle and time separation maintained are shown in Figures 39.7 and 39.8, respectively, when the leading vehicle is at a constant DoS (i.e., moving at a uniform velocity). The variation in the velocity response of the host vehicle in the time window 4.5–5.0 s (in Figure 39.7) is due to the variation in DoS in the time interval 4.0–4.5 s. This delay of about 0.5 s in the velocity response of the host vehicle to the changes in the environment can be attributed to its physical characteristics and control loop delay.

The next experiment simulated a scenario, where leading vehicle exhibits varying velocity. The DoS was increased gradually between time interval 1 and 6 s, kept constant between 6 and 8 s, then gradually decreased from 8 to 12 s and again kept constant between 12 and 14 s. Figure 39.9 shows the velocity response and Figure 39.10 shows the time separation maintained in this case.

39.8.2 Two-Level Real-Time Data Repository Experiments

The second set of experiments tested the two-level real-time data repository design. These experiments captured the system behavior when the data update tasks of the two-level repository design are executed only when necessary.

The velocity response of the host vehicle with the use of the real-time repository when the leading vehicle is moving at a constant distance is shown in Figure 39.11. Invocations of the task updating the DoS (DistT) in DDR are observed (stems in the graph indicate the time of invocation). This task is invoked only when the DoS changes by a preset threshold value, 5 cm. The DoS is calculated once at the beginning of the experiment and twice in the time interval 10–12 s when the distance is decreased by 5 cm and then set to the original value. The task deriving the DoS and updating the DDR executes only when necessary. Similar results can be observed in Figure 39.12 where the leading distance increases in time intervals 0–5 and 6–7 s, kept constant between 5–6 and 7–8 s and then gradually reduced from 8 to 12 s. We can observe from the graph that during the time interval 5–6 and 7–8 s when the DoS was constant, the OD task is not invoked and during other time intervals, it is invoked whenever the DoS changes by threshold value. Table 39.3 shows the DistT task's invocation in both the models (with and without two-level real-time repository model) over different time windows. The periodicity of the task was set to 0.3 s in the

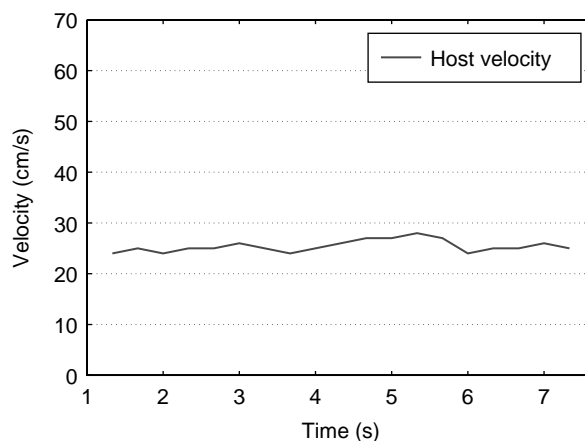


FIGURE 39.6 Host velocity: cruise control case, set velocity = 25 cm/s.

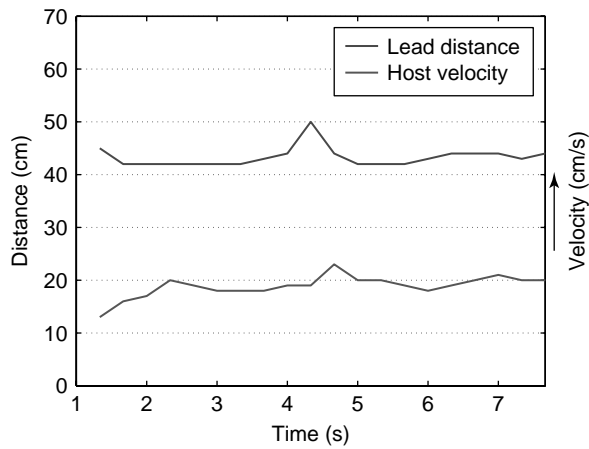


FIGURE 39.7 Host velocity: leading vehicle with uniform velocity.

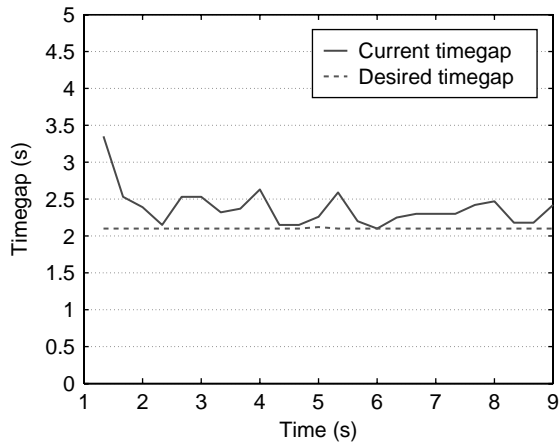


FIGURE 39.8 Timegap: leading vehicle with uniform velocity.

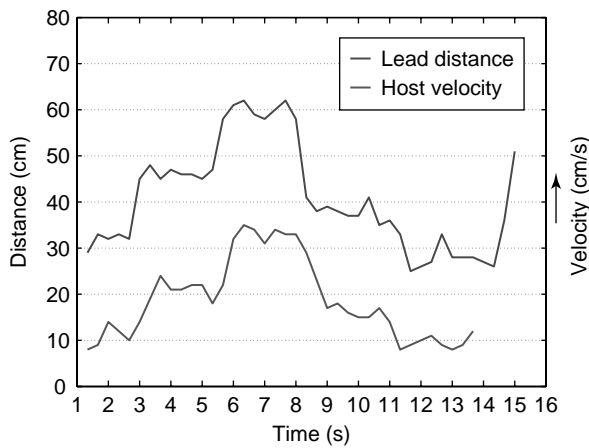


FIGURE 39.9 Host velocity: leading vehicle with varying velocity.

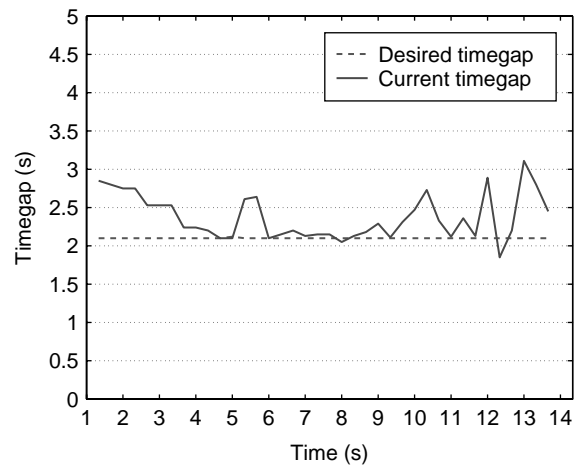


FIGURE 39.10 Timegap: leading vehicle with varying velocity.

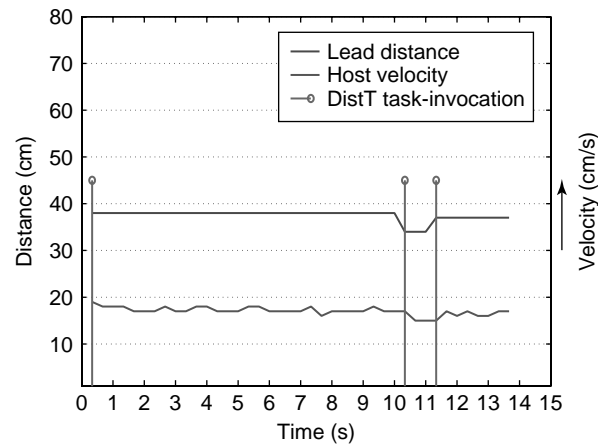


FIGURE 39.11 Host velocity: constant leading distance case—on-demand updates.

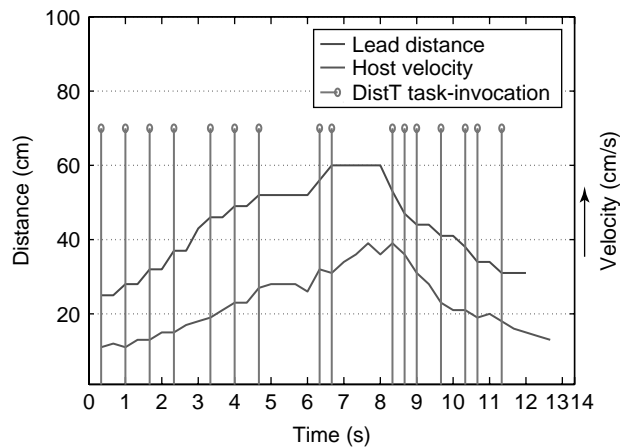


FIGURE 39.12 Host velocity: leading vehicle with varying velocity case—on-demand updates.

TABLE 39.3 Number of Invocations of Lead Distance Updating Task in the Two Models

Lead Distance	Time Window	DistT Task Invocation	
		With Two-Level	Without Two-Level
Const	0 to 12	3	40
Incr-Decr	0 to 12	16	40

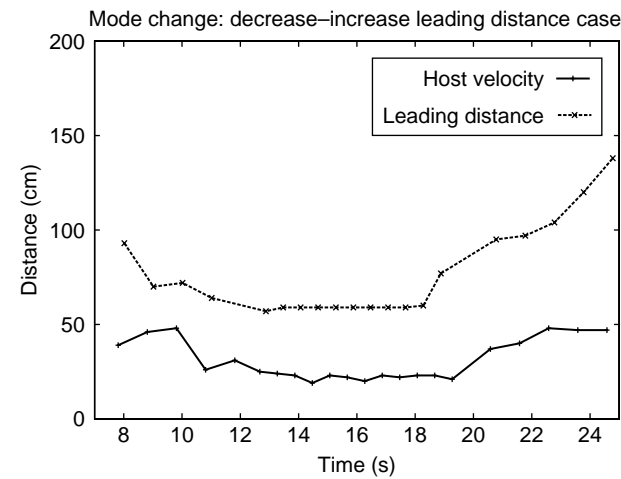


FIGURE 39.13 Host velocity response: Mode-change case, leading vehicle with varying velocity.

experiments discussed in Section 39.8.1. We can observe that this approach requires less processing power compared to the conventional approach.

39.8.3 Mode-Change Experiments

Finally, we study the effect of mode-change delay on data freshness and safety of the system in terms of maintaining the desired time separation. The velocity response of the host vehicle and time separation maintained with the mode-change implementation, when leading vehicle is moving with varying velocity are shown in Figures 39.13 and 39.14, respectively. The DoS is kept as the criterion for changing the mode, i.e., if *leading distance* \leq 65 cm enter SC mode else NC mode. The frequency of execution of tasks in NC mode is half that of the frequency in SC mode: the periodicity of the tasks was set to 0.3 s in SC mode and 0.6 s in NC mode. We can observe from the graph that the system is operating in NC mode between time interval 8–12 s, where the distance is gradually decreasing from 90 to 65 cm and then it enters SC mode at time 13 s and again switches back to NC mode at 19 s. The desired timegap is violated couple of times in Figure 39.14, which can be attributed to the inertia of the vehicle and mode-change delay. This suggests that a conservative approach should be taken while deciding the safe DoS or time separation by taking these two factors into account. In this approach too, we can observe that half the CPU capacity is saved when the system operates in NC mode compared to conventional approach.

39.9 Related Work

ACC is a well-studied research topic in control systems. The design techniques and simulation results for ACC equipped vehicles are reported in Refs. 12 and 13. The chapters discuss a design of an autonomous

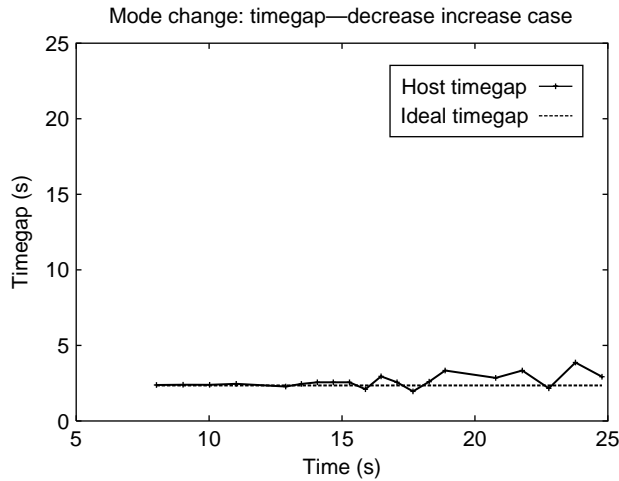


FIGURE 39.14 Timegap response: mode-change case, timegap in leading vehicle with varying velocity.

intelligent cruise control and a constant time-headway rule for a worst-case stopping scenario. Prior work discusses control aspects of the application not the real-time aspects. A data-centric approach to the architectural design of performance critical vehicular applications has been examined before in Refs. 14 and 15. In particular, Gustafsson and Hansson [14] address the issues in the design and the implementation of an active real-time database system for EECU (Electronic Engine Control Unit) software. A set of OD updating algorithms: on-demand depth first traversal (ODDFT) and on-demand top bottom (ODTB) are presented in Ref. 7. These algorithms optimistically skip unnecessary updates and hence provide increased performance. Methods for the specification and runtime treatment of mode changes are discussed in Ref. 3. An approach to handle mode changes in the time triggered, strictly periodic, preruntime scheduled system MARS, is studied in Ref. 16. Sha et al. [8] attempt to address the problem of analyzing *a priori* a single-processor system, scheduled according to the rate monotonic scheduling policy, with tasks able to lock and unlock semaphores according to the priority ceiling protocol. The mode-change protocol they propose is one, where tasks are either deleted or added across a mode change.

39.10 Conclusions and Further Work

In this chapter, we have discussed the real-time data issues involved in developing automotive applications. We have also presented the issues involved in developing real-time support for ACC and shown the effectiveness of two real-time data services in handling the data and optimally utilizing computational resources. By using a two-level real-time data repository model to update the derived data only when necessary and designing ACC with different modes, each containing different task sets with different characteristics, we have utilized processor capacity effectively, compared to the existing approaches. We have shown that these approaches can enable system designers and developers to build a safe and predictable system making effective use of the CPU capacity even if there are demanding validity requirements to be satisfied by the applications.

We are working on possible extensions to the research described here. First, more analysis of the system design is being carried out with different conditions for mode switching, periodicity of the tasks in different modes, and conditions for triggering second-level update tasks. Second, application needs are being mapped to a distributed platform (as it is the case in the real-world) and the real-time

communication issues between the processors are being studied using FlexRay and CAN like communication infrastructures. Third, the impact of mode-change and real-time data repository design concepts on the controller's stability and performance from control theoretical perspective is being studied. Fourth, we are also looking at other real-time data services to address issues described in this chapter such as sensor data fusion and efficient dissemination of data in V2V and V2I communication applications.

Acknowledgment

Our special thanks are due to Sachitanand Malewar for designing and realizing the required hardware.

References

1. H. Kopetz, "Automotive electronics," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1992, pp. 132–140.
2. R. Gurulingesh, N. Sharma, K. Ramamritham, and S. Malewar, "Efficient real-time support for automotive applications: A case study," in *IEEE Conference on Real-Time Computing Systems and Applications*, Aug. 2006.
3. G. Fohler, "Flexibility in statically scheduled hard real-time systems," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 1994.
4. H. Kopetz and G. Bauer, "The time-triggered architecture," in *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Oct. 2001.
5. H. Kopetz and G. Grünsteidl, "Ttp—a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, 1994.
6. J. Zhou and H. Peng, "Range policy of adaptive cruise control vehicles for improved flow stability and string stability," in *IEEE Transactions on Intelligent Transportation Systems*, vol. 6, pp. 229–237, June 2005.
7. T. Gustafsson and J. Hansson, "Dynamic on-demand updating of data in real-time database systems," in *Proceedings of the ACM Symposium on Applied computing*, NY, USA, 2004, pp. 846–853.
8. L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," Tech. Rep., Amherst, MA, USA, 1989.
9. L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, vol. 19. Washington, DC: IEEE Computer Society, 2–4 Dec. 1998, pp. 4–13.
10. L. Abeni and G. Buttazzo, "Adaptive bandwidth reservation for multimedia computing," in *Sixth International Conference on Real-Time Computing Systems and Applications*, pp. 70–77, Dec. 1999.
11. L. P. G. Lipari and L. Marzario, "Wp4 resource management components," OCERA Consortium, Tech. Rep., Apr. 2003.
12. P. A. Ioannou and C.-C. Chien, "Autonomous intelligent cruise control," in *IEEE Trans. on Vehicular Technology*, June 1993, pp. 657–672.
13. U. Palmquist, "Intelligent cruise control and roadside information," *IEEE Micro*, pp. 20–28, 1993.
14. T. Gustafsson and J. Hansson, "Data management in real-time systems: A case of on-demand updates in vehicle control systems," in *Proceedings of the 10th IEEE RTAS*, 2004, pp. 182–191.
15. D. Nyström, A. Tesanovic, C. Norström, J. Hansson, and N.-E. Bänkestad, "Data management issues in vehicle control systems: A case study," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, 2002, pp. 249–256.
16. G. Fohler, "Realizing changes of operational modes with pre run-time scheduled hard real-time systems," in *Proceedings of the 2nd International Workshop on Responsive Computer Systems*. Saitama, Japan: Springer, Oct. 1992.

17. K. Ramamritham, "Real-time databases," *Distributed and Parallel Databases*, vol. 1, no. 2, pp. 199–226, 1993.
18. K. Ramamritham, S. H. Son, and L. C. DiPippo, "Real-time databases and data services." *Real-Time Systems*, vol. 28, no. 2-3, pp. 179–215, 2004.
19. B. Adelberg, H. Garcia-Molina, and B. Kao, "Applying update streams in a soft real-time database system," in *SIGMOD'95: Proceedings of the International Conference on Management of Data*. New York, NY: ACM Press, 1995, pp. 245–256.
20. B. Kao, K.-Y. Lam, B. Adelberg, R. Cheng, and T. Lee, "Maintaining temporal consistency of discrete objects in soft real-time database systems," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 373–389, 2003.

Index

A

- absolute temporal constraints, defined, **10-2**
- abstraction levels, granularity refinement, **32-16–32-17**
- ACCORD modeling environment, COMET
 - automated configuration and analysis, **28-12–28-16**
- ACP-style synchronization, process algebras, **31-3**
- actions, process algebras, **31-2**
- Ada programming language
 - Ada 95, basic properties, **13-3**
 - Ada 2005 new features, **13-3–13-4**
 - Baker's preemption level protocol, protected objects, **13-15**
 - deadline support, **13-14–13-15**
 - earliest deadline first scheduling, **13-15–13-19**
 - constraints, **13-18**
 - dispatching rules, **13-16–13-18**
 - task set examples, **13-19**
 - execution-time clocks, **13-4–13-5**
 - execution-time server programming, **13-8–13-13**
 - execution-time timers, **13-6**
 - group budgets, **13-7–13-8**
 - mixed dispatching systems, **13-19–13-20**
 - timing events, **13-4**
- bounded-execution-time programming, **11-8–11-10**
- Java comparisons with, **13-20–13-21**
- adapted development process, automotive embedded systems, **38-17**
- adaptive cruise control (ACC) (automotive), real-time databases
 - development of, **39-5–39-7**
 - dual mode systems, **39-8–39-10**
 - mode-change experiments, **39-17**
 - on-demand task scheduling, **39-12**
 - overview, **39-1–39-2**
 - repository specifics, **39-10–39-11, 39-14–39-17**
 - research issues, **39-3–39-5**
 - robotic vehicle control experiment, **39-12–39-17**
 - task models, **39-11–39-12**
- adaptive factor, real-time databases, **27-4–27-5**
- adaptive segmentation, DeeDS NG architecture, **29-12–29-13**
- admission control policy
 - COMET database platform, **28-10–28-11**
 - USB subsystem architecture, **17-8**
 - polling tree structure, **17-13–17-15**
- Advanced Configuration and Power Interface (ACPI)
 - specification, dynamic power management, **6-12**
- advanced driver assistance, automotive embedded systems, **38-5**
- aggregate quality-of-service (QoS) management, distributed real-time embedded systems, **36-4–36-5**
- AGR algorithm, intertask voltage scaling, **6-8**
- AlarmNet system, **20-7–20-9**
- alphabet of P, process algebras, **31-2**
- analysis-based memory allocation, interrupt systems, stack overflow management, **16-4–16-5**
- analyzer process, deterministically networked real-time systems, **11-19**
- aperiodic recurrence, temporal control, real-time systems, **10-4**
- aperiodic task handling
 - automotive adaptive cruise control, **39-12**
 - real-time resource management and scheduling, **2-7–2-9**
 - real-time specification for Java, **12-8–12-9**
 - sporadic task model relaxation, **3-14–3-15**
- API-UDM, metaprogrammable model manipulation, **33-6–33-7**
- application-adaptive scheduling (ASC), sensornet messaging architecture, **21-5–21-6**
- application-adaptive structuring (AST), sensornet messaging architecture, **21-4–21-6**
- application-defined schedulers, real-time specification for Java, **12-9**

- application-layer approaches, multimedia mobile computing, **37-9–37-14**
 - energy conservation, **37-10–37-11**
 - quality-of-service management, **37-11–37-14**
- application-oriented networking (AON), sensor-net messaging architecture, **21-17–21-18**
- approximated solutions, priority-driven periodic real-time systems, steady-state backlog analysis, **9-14–9-15**
- APPROXIMATE query processor, real-time databases, quality-of-service (QoS) support and analysis, **24-13–24-14**
- arbitrary-deadline systems, multiprocessor sporadic task systems, **3-5**
 - dynamic priority scheduling, **3-5**
 - fixed-job priority scheduling, **3-6**
 - fixed-task priority scheduling, **3-11–3-13**
 - global scheduling, **3-8–3-10**
- architectural platforms
 - automotive embedded systems
 - adapted development process, **38-17**
 - comprehensive architectures, **38-11–38-12**
 - functionality improvements, **38-7–38-8**
 - human-machine interface, **38-8**
 - innovations in, **38-6**
 - maintenance systems, **38-9**
 - quality and reliability, **38-15–38-16**
 - DeeDS NG systems, **29-1–29-4**
 - distributed systems, **32-9–32-10**
 - embedded systems, **32-1–32-3**
 - interface composition, **32-19–32-20**
 - flexible time-triggered (FTT) communication paradigm, **19-5**
 - hardware/software codesign, **34-1–34-7**
 - interrupt systems, **16-3–16-4**
 - modeling tool architectures, **33-3–33-9**
 - sensor-net messaging architecture, **21-3–21-6**
 - unified link-layer abstraction, **21-17–21-18**
 - USB subsystems, **17-7–17-8**
 - wireless sensor networks, **20-8–20-9**
- ARTS kernel, temporal control, real-time systems, **10-11–10-13**
- ASCR process algebra, resource-sensitive systems operators, **31-10**
 - schedulability analysis, **31-13–31-19**
- aspect-oriented software development (AOSD)
 - application-tailored databases, **28-4–28-6**
 - COMET database platform, **28-7–28-9**
 - quality-of-service management, distributed real-time embedded systems, **36-9–36-10**
- assisted living facilities, wireless sensor networks, **20-7–20-9**
- associations, in metamodeling languages, **33-9–33-10**
- asynchronous events, real-time specification for Java, **12-7–12-8**
- asynchronous messaging system
 - embedded wireless networks, **22-2**
 - flexible time-triggered (FTT) architecture, **19-14–19-17**
- asynchronous requirements table (ART), flexible time-triggered (FTT) communication paradigm, **19-7**
 - schedulability analysis, **19-15–19-17**
- atomic state classes (ASC), Time Petri Nets, emergence of, **30-2**
- atomic state class graph (ASCG), **30-13–30-15**
- ATP process algebra, **31-8–31-9**
- audio/video streaming, Common Object Request Broker Architecture (CORBA), **25-10–25-11**
- automatons, synchronous programming, **14-12–14-14**
- automotive applications
 - embedded systems and software technology
 - architecture innovations, **38-6–38-9, 38-11–38-12**
 - coding systems, **38-9**
 - competency improvements, **38-7**
 - complex comprehensive data models, **38-8**
 - complexity reduction, **38-12**
 - cost issues, **38-6, 38-10–38-11**
 - crash prevention and safety, **38-5**
 - current practices, **38-2–38-3**
 - development and maintenance processes, **38-8–38-10**
 - domain profile, **38-3–38-4**
 - driver assistance, **38-5**
 - energy conservation, **38-5**
 - functionality innovations, **38-4, 38-7–38-8**
 - future applications, **38-4–38-6**
 - hardware/technical infrastructure, **38-10**
 - historical background, **38-2**
 - human-machine interface adaptations, **38-5, 38-8**
 - integrated tool support, **38-14–38-15**
 - interconnection car networking, **38-6**
 - maintenance systems, **38-9–38-10**
 - overview, **38-1–38-2**
 - personalization and individualization systems, **38-6**
 - practical challenges, **38-6–38-11**
 - process improvement, **38-13**
 - programmable car development, **38-5**
 - quality and reliability improvement, **38-15–38-16**
 - requirements engineering, **38-8**
 - research agenda, **38-16–38-18**
 - seamless model-driven development, **38-13–38-14**
 - software and systems integration, **38-9**

- real-time databases
 - adaptive cruise control, **39-5–39-7**
 - dual mode systems, **39-8–39-10**
 - mode-change experiments, **39-17**
 - on-demand task scheduling, **39-12**
 - overview, **39-1–39-2**
 - repository specifics, **39-10–39-11, 39-14–39-17**
 - research issues, **39-3–39-5**
 - robotic vehicle control experiment, **39-12–39-17**
 - task models, **39-11–39-12**
- automotive open system architecture (AUTSAR),
 - resource frugality and scalability, **18-4**
- average-case execution path (ACEP), path-based
 - intratask dynamic voltage scaling, **6-6**
- avionics mission computing
 - automotive embedded systems and, **38-15**
 - quality-of-service-enabled component
 - middleware, **15-9–15-10**

B

- backbone network, wireless sensor networks, **20-8**
- backend database, wireless sensor networks, **20-8**
- background/foreground systems, implicit temporal
 - control, **10-6–10-7**
- backlog analysis algorithm. *See also* steady-state
 - backlog analysis
 - priority-driven periodic real-time systems, **9-5–9-6**
 - complexity evaluation, **9-20–9-22**
 - computational complexity, **9-16**
- backlog dependency tree, priority-driven periodic
 - real-time systems, **9-7–9-9**
- backward compatibility, real-time specification for
 - Java, **12-2–12-3**
- Baker earliest deadline first (EDF) algorithm
 - Ada 2005 programming, preemption level
 - protocol, **13-15**
 - multiprocessor sporadic task systems, **3-9**
- bandwidth utilization, USB subsystems, **17-6–17-7**
 - workload reinsertion algorithm, **17-10–17-12**
- battle damage indications (BDIs), PCES capstone
 - (DARPA) case study, **36-19–36-22**
- Bayou storage system, DeeDS NG architecture,
 - 29-16–29-17**
- BC fixed-task priority scheduling algorithm,
 - multiprocessor sporadic task systems, **3-12**
- BCL fixed-task priority scheduling algorithm,
 - multiprocessor sporadic task systems, **3-13**
- beacon packets, sensor network messaging architecture,
 - 21-3–21-4**
- best-effort recovery, DeeDS NG architecture, **29-5**
- bicriteria scheduling issues, imprecise computation
 - model, **8-1–8-10**
- binary encoding algorithm
 - USB systems, period modification, **17-13**
 - worst-case execution time analysis, **35-14**

- bisimilarity, Time Petri Nets, branching properties
 - preservation, **30-13–30-15**
- bit error rate (BER), embedded wireless networks,
 - 22-2–22-3**
- black-box technology, automotive real-time systems,
 - 39-1–39-2**
- B-MAC protocol, wireless sensor networks, **20-2**
- body network architecture, wireless sensor networks,
 - 20-8**
- Boolean causality, synchronous programming,
 - 14-10–14-11**
- Boolean function feasible test, imprecise
 - computation model, 0/1-constraints,
 - 8-6–8-7**
- bounded-delay resource model, compositional
 - real-time framework, **5-7**
 - schedulability analysis, **5-9–5-11**
 - utilization bounds, **5-13–5-14**
- bounded-execution-time (BET) programming,
 - real-time systems, **11-7–11-10**
- boundedness, Time Petri Nets, **30-5**
 - decidability issues, **30-8**
- bounded recovery, DeeDS NG architecture, **29-5**
- branching properties, Time Petri Nets, state space
 - preservation, **30-13–30-15**
- branch prediction, embedded systems,
 - execution-time analysis, **35-7**
- broadcast and multicast services (BCMCS)
 - specification, multimedia mobile
 - computing, **37-12**
- broadcast communication model, point-to-point
 - real-time data distribution, **25-5–25-6**
- budget node, robust and implicit earliest deadline
 - first protocol, embedded wireless networks,
 - 22-4–22-5**
- buffering mechanisms
 - real-time databases, **24-11–24-12**
 - synchronous programming, preemptive
 - multitasking, **14-15–14-17**

C

- cache behavior, embedded systems, execution-time
 - analysis, **35-6–35-7**
- callgraphs, embedded systems, interrupt software
 - guidelines, **16-10**
- causal event ordering, stopwatch model of temporal
 - control, **10-5–10-6**
- causality
 - embedded systems, interface model, **32-8–32-9**
 - multifunctional system structuring, service
 - interface, **32-11–32-12**
 - synchronous programming, **14-4–14-5, 14-9–14-11**
- ccRM algorithm, intertask voltage scaling, **6-8**

- CCS-style synchronization, process algebras, **31-3**
 - discrete-time modeling, **31-7–31-8**
 - resource-sensitive systems, **31-10–31-13**
- central processing unit (CPU)
 - hardware/software codesign
 - customization, **34-5–34-6**
 - partitioning algorithms, **34-2–34-4**
 - worst-case execution time analysis, target
 - architecture for, **35-14**
- channel valuations
 - distributed systems model, **32-9–32-10**
 - embedded systems, interface model, **32-7–32-8**
- classes, in metamodeling languages, **33-9–33-10**
- clear channel assessment (CCA), wireless sensor networks, **20-2**
- client-server, point-to-point real-time data
 - distribution, **25-5**
- clock domains, Time Petri Nets, strong state class
 - graph construction, **30-9–30-13**
- clock synchronization, wireless sensor networks, **20-5**
- Close operator, process algebras, resource-sensitive systems, **31-12–31-13**
- clustering architectures, automotive embedded systems, **38-12**
- coarsening function, timescale modeling, **32-20–32-21**
- code generation, synchronous programming, **14-11–14-14**
- coding infrastructure, automotive embedded systems, **38-9**
- collision avoidance (automotive), real-time databases
 - for, **39-1–39-2**
- COMET database platform
 - application-tailored databases, **28-6–28-16**
 - automated configuration and analysis, **28-12–28-16**
- commit protocols, real-time databases, **24-9–24-10**
- Common Object Request Broker Architecture (CORBA)
 - distributed object computing applications, **15-3**
 - quality-of-service-enabled component middleware
 - distributed real-time embedded systems
 - aggregate management, **36-2**
 - PCES capstone program (DARPA) case study, **36-25–36-27**
 - real-time CORBA-lightweight CCM integration, **15-5–15-6**
 - quality-of-service-enabled DOC middleware, **15-3**
 - real-time data distribution, **25-7–25-11**
 - robot service communication architecture, **18-6–18-7**
- communication networks
 - flexible time-triggered (FTT) communication
 - paradigm, operational flexibility, **19-3–19-4**
 - real-time and embedded systems, **1-4–1-5**
 - communication requirements database (CRDB),
 - flexible time-triggered (FTT)
 - communication paradigm, operational flexibility, **19-4**
- compatibility, automotive embedded systems, **38-9–38-10**
- competency improvements, automotive embedded systems, **38-7**
- compilation tasks, SNAFU programming language, **23-6–23-7**
- compiler-directed intratask dynamic voltage scaling,
 - power-aware resource management, **6-6–6-7**
- complexity evaluation
 - automotive embedded systems, **38-12**
 - priority-driven periodic real-time systems,
 - stochastic analysis, **9-20–9-22**
- component abstraction, compositional real-time
 - framework, **5-3–5-5**
- component-based software development (CBSD),
 - application-tailored databases, real-time and embedded systems, **28-4–28-6**
- component composition
 - compositional real-time framework, **5-3–5-5**
 - embedded systems, syntactic interfaces, **32-4–32-5**
- component-integrated ACE ORB (CIAO)
 - avionics mission computing applications, **15-9–15-10**
 - inventory tracking applications, **15-11–15-12**
 - lessons learned from, **15-12–15-13**
 - quality-of-service-enabled component
 - middleware, **15-5–15-9**
 - shipboard computing applications, **15-10–15-11**
- component middleware technology, distributed
 - real-time and embedded system
 - applications, **15-1–15-2**
 - avionics mission computing, **15-9–15-10**
 - conventional component middleware, **15-3–15-4**
 - conventional DOC middleware, **15-3**
 - CoSMIC model-driven deployment and
 - configuration, **15-8–15-9**
 - DAnCE system, **15-6–15-8**
 - inventory tracking, **15-11–15-12**
 - lessons learned from CIAO, DAnCE and CoSMIC
 - systems, **15-12–15-13**
 - quality-of-service-enabled component
 - middleware, **15-4, 15-13–15-14**
 - quality-of-service-enabled DOC systems, **15-3, 15-13**
 - real-time CORBA-lightweight CCM CIAO
 - integration, **15-5–15-6**
 - shipboard computing, **15-10–15-11**
- component synthesis of model integrated computing (CoSMIC)
 - avionics mission computing, **15-9–15-10**

- inventory tracking, **15-11–15-12**
- lessons learned from, **15-12–15-13**
- quality-of-service component middleware,
 - model-driven deployment and configuration, **15-8–15-9**
- shipboard computing applications, **15-10–15-11**
- composed systems, distributed system model, **32-9–32-10**
- compositional real-time framework
 - bounded-execution-time programming, **11-8–11-10**
 - embedded systems, **32-18–32-20**
 - property refinement, **32-16**
 - schedulability analysis
 - bounded-delay resource model, **5-7, 5-9–5-11, 5-13–5-14**
 - extension, **5-16–5-18**
 - interacting task support, **5-17–5-18**
 - overview, **5-1–5-3, 5-5–5-6**
 - periodic resource model, **5-8, 5-11–5-12, 5-14–5-16**
 - resource models, **5-7–5-8**
 - schedulable workload utilization bounds, **5-12–5-16**
 - scheduling unit, component, and interface, **5-3–5-4**
 - sporadic task support, **5-17**
 - system models, **5-4**
 - workload models, **5-6–5-7**
- composition and composition patterns,
 - quality-of-service management software, distributed real-time embedded systems, **36-10**
- comprehensive system modeling theory, embedded systems, **32-3–32-10**
- computational complexity
 - priority-driven periodic real-time systems, stochastic analysis, **9-15–9-17**
 - Time Petri Nets, **30-15–30-16**
- concurrency control
 - embedded interrupt systems, **16-11**
 - real-time databases
 - irreconcilably conflicting transactions
 - concurrency control, **27-1–27-3**
 - high data contention, **27-11–27-12**
 - low resource contention, **27-12–27-13**
 - OCC-APR technique, **27-3–27-5**
 - serialization order, dynamic adjustment, **27-3**
 - simulation model, **27-6–27-11**
 - transaction processing, **24-6–24-9**
- conflict classification and detection, DeeDS NG architecture, **29-8**
- conflict resolution
 - DeeDS NG architecture, **29-8–29-9**
 - real-time databases, soft transactions, **24-7–24-9**
- congestion, wireless sensor networks, **20-4**
- congruence relations, process algebras, **31-4**
- constrained-deadline systems, multiprocessor sporadic task systems
 - dynamic priority scheduling, **3-5**
 - fixed-job priority scheduling, **3-6**
 - fixed-task priority scheduling, **3-11–3-13**
 - global scheduling, **3-8–3-10**
- constrained density, sporadic task systems, **3-2**
- constrained maximum weighted error (CMWE), imprecise computation model, **8-3–8-6**
- constrained total weighted error (CTWE), imprecise computation model, **8-2–8-3**
- constraint encoding, ASCR process algebra, schedulability analysis, **31-16–31-17**
- constructive causality, synchronous programming, **14-10–14-11**
- continuous transitions, Time Petri Nets, state spaces, **30-3**
- control engineering. *See also* temporal control, real-time systems
 - adaptive cruise control systems, **39-6**
 - automotive embedded systems, **38-7**
 - networked control systems, **11-14–11-17**
 - real-time programming
 - bounded-execution-time programming, **11-7–11-10**
 - computing complexity, **11-3–11-6**
 - deterministically networked real-time systems, **11-17–11-19**
 - embedded systems, **1-3–1-4**
 - evolution of, **11-1–11-3**
 - logical-execution-time programming, **11-12–11-14**
 - networked systems, **11-14–11-17**
 - nondeterministically networked real-time systems, **11-19–11-21**
 - physical-execution-time programming, **11-6**
 - zero-execution-time programming, **11-10–11-12**
 - synchronous programming, **14-2**
- control flow graph (CFG), interrupt systems, stack overflow management, **16-5–16-6**
- controller area network (CAN) interconnections
 - flexible time-triggered (FTT) communication paradigm
 - asynchronous messaging system, **19-14–19-17**
 - dual-phase elementary cycle, **19-5–19-6**
 - synchronous messaging system, **19-9–19-14**
 - networked control performance, **11-14–11-17**
- controller systems, robot service communication architecture, **18-5**

convolve-shrink procedure, priority-driven periodic real-time systems, backlog dependency tree, **9-7-9-9**

core framework, robot service communication architecture, **18-6-18-9**

correctness assessment, worst-case execution time analysis, **35-14**

cost issues, automotive embedded systems
 long-term investment, **38-11**
 reduction strategies, **38-6**
 software cost controls, **38-10**
 supplier independence, **38-10-38-11**

COSYMA partitioning system, hardware/software codesign, **34-3-34-4**

COSYN algorithm, hardware/software codesign, **34-5**

cosynthesis algorithms, hardware/software codesign, **34-4-34-5**

crash prevention and safety, automotive embedded systems, **38-5**

cross-layer dynamic power management, **6-12**

custom accelerator hardware, execution-time analysis, **35-8**

cycle-safe iteration, SNAFU programming language, **23-5**

D

DARPA adaptive and reflectivemiddleware systems (ARMS) program, quality-of-service-enabled component middleware, **15-10-15-11**

data aggregation, sensor networks, **24-17-24-18**

database services
 real-time systems, **1-5-1-6**
 data, transaction, and system characteristics, **24-2**
 data freshness and timing properties, **24-4-24-6**
 distribution, **24-3**
 dynamic web data dissemination, **24-20-24-21**
 mobile real-time databases, **24-18-24-20**
 overview, **24-1-24-2**
 quality of service
 data stream management, **24-15-24-17**
 distributed data services, **24-14-24-15**
 management, **24-12-24-14**
 metrics, **24-14**
 quality of data and, **24-3**
 related services, **24-3-24-4**
 scheduling and transaction processing, **24-2-24-3**
 sensor networks, **24-17-24-18**
 transaction processing, **24-6-24-12**
 distributed databases and commit protocols, **24-9-24-10**
 I/O and buffer management, **24-11-24-12**

recovery issues, **24-10-24-11**

scheduling and concurrency control, **24-6-24-9**

tailorability dimensions, **28-2-28-3**

data-centric publish-subscribe (DCPS) model, real-time data distribution service, **25-11-25-13**

data characteristics
 automotive real-time systems, **39-2-39-4**
 real-time databases, **24-2-24-6**
 real-time data distribution, **25-3-25-5**

data completeness metric, real-time databases, quality-of-service management, **24-16-24-17**

data contention, real-time databases, irreconcilably conflicting transactions, **27-11-27-12**

data distribution and processing, automotive real-time systems, **39-4**

data-driven adaptation, sensornet messaging architecture, learn-on-the-fly (LOF) protocol, **21-11-21-12**

data-driven link estimation and routing, sensornet messaging architecture, **21-6-21-17**
 ELD routing metric, **21-8-21-10**
 evaluation, **21-13-21-17**
 LOF data-driven protocol, **21-10-21-13**

dataflow coding, synchronous programming, **14-11-14-14**
 static scheduling, **14-18-14-19**

data models, embedded systems, **32-4**
 automotive applications, **38-8**

data repositories, real-time automotive applications, **39-10-39-11**
 robotic vehicle controls, **39-12-39-17**

data streams
 embedded systems, interface model, **32-6-32-7**
 point-to-point real-time data distribution, **25-6**
 real-time databases, quality-of-service management, **24-15-24-17**

data transfer modes, USB system, **17-5**

data types, in metamodeling languages, **33-9-33-10**

data usage, real-time data distribution, **25-4-25-5**

deadline miss percentage (DMP), real-time databases, performance evaluation, **27-8-27-12**

deadline monotonic algorithm
 fixed-priority scheduling, **2-6**
 multiprocessor sporadic task systems, fixed-task priority scheduling, **3-11**
 stochastic time demand analysis, **9-2**

deadline support, Ada 2005 programming, **13-14-13-15**

deadlocked processes, process algebras
 discrete-time modeling, **31-6-31-8**
 resource-sensitive systems, **31-10-31-13**

- DeeDS NG systems
 - database model and eventual consistency, **29-6–29-7**
 - design philosophy and capabilities, **29-1–29-2**
 - distributed real-time simulation, **29-5**
 - legacy DeeDS architecture, **29-3–29-4**
 - optimistic replication, **29-4, 29-7–29-9**
 - scalability, **29-4–29-5, 29-9–29-13**
 - simulation DeeDS, **29-13–29-16**
 - virtual full replication, **29-9–29-13**
 - wildfire scenario, **29-2–29-3**
- defect diagnosis and repair, automotive embedded systems, **38-9–38-10**
- deferrable scheduling algorithm (DS-FP), real-time databases, data timing and freshness, **24-5–24-6**
- deferrable server, Ada 2005 programming for, **13-8–13-10**
- demand bound function
 - compositional real-time framework, **5-4**
 - workload model, **5-5–5-7**
 - sporadic task systems, **3-3**
- density, sporadic task systems, **3-2**
- dependencies, DeeDS NG architecture, **29-11–29-12**
- deployment and configuration (DAnCE)
 - QoS-enabled component software
 - avionics mission computing, **15-9–15-10**
 - design, **15-6–15-8**
 - inventory tracking applications, **15-11–15-12**
 - lessons learned from, **15-12–15-13**
 - shipboard computing applications, **15-10–15-11**
 - static configuration optimization, **15-8**
- depth analysis, interrupt systems, stack overflow management, **16-5–16-6**
- DESERT metaprogrammable tool, design space exploration, **33-7–33-8**
- design environment
 - automotive embedded systems, logical architecture systems, **38-12**
 - quality-of-service management software, distributed real-time embedded systems, **36-10–36-12**
- design space exploration, DESERT metaprogrammable tool, **33-7–33-8**
- deterministically networked real-time systems (DNRTS), control performance, **11-14–11-19**
- deterministic systems, synchronous programming, **14-5**
- deterministic timing guarantee
 - applications, **9-1–9-2**
 - temporal constraints, **10-2–10-3**
- device configuration descriptor (DCD), robot service communication domain profile, **18-8–18-9**
- “Dhall phenomenon,” global EDF scheduling, multiprocessor sporadic task systems, **3-7**
- directed acyclic graph (DAG), SNBench platform, **23-2–23-7**
- directed diffusion solution, wireless sensor networks, **20-3–20-4**
- direct sequence spread spectrum (DSSS), embedded wireless networks, **22-2–22-3**
- direct sum of histories, embedded systems, interface model, **32-8**
- disabled interrupts, defined, **15-2**
- discrete-event systems, synchronous programming limitations, **14-3**
- discrete state graph (DSG), Time Petri Nets
 - definition, **30-4**
 - LTl property preservation, **30-7–30-8**
 - marking and trace preservation, **30-5–30-7**
 - strong state class graph properties, **30-12–30-13**
- discrete-time Markov decision processes (DTMDP), dynamic power management, **6-11**
- discrete-time modeling, process algebras, **31-5–31-8**
- discrete time stochastic policies, dynamic power management, **6-10–6-11**
- discrete transitions, Time Petri Nets, state spaces, **30-3**
- dispatching mechanism
 - Ada 2005 programming
 - EDF rules for, **13-16–13-18**
 - mixed systems, **13-19–13-20**
 - real-time specification for Java, **12-5**
- distance constrained scheduling, real-time databases, **24-10**
- distributed environment
 - real-time databases, **24-9–24-10**
 - quality-of-service management, **24-14–24-15**
 - robotics systems, device abstraction and component models, **18-3**
 - synchronous programming, **14-17–14-19**
- distributed multitasking sensor networks
 - applications, **23-16–23-19**
 - basic properties, **23-1–23-2**
 - research and development issues, **23-2–23-3**
 - sensorium execution environments, **23-12–23-16**
 - sensorium service dispatcher, **23-9–23-12**
 - sensorium task execution plan, **23-7–23-9**
 - SNAFU programming language, **23-4–23-7**
 - SN workBench platform, **23-3–23-4**
- distributed object computing (DOC), conventional middleware, **15-3**
- Distributed Programming System (DPS), temporal control, real-time systems, temporal scopes concept, **10-9–10-11**
- distributed QoS modeling environment (DQME), quality-of-service management software, **36-11–36-12**

distributed real-time database systems (DRTDBS).

See also DeeDS NG systems

commit protocols, **24-9–24-10**

distributed real-time embedded systems (DRE)

component middleware technology applications,
15-1–15-2

avionics mission compouting, **15-9–15-10**

conventional component middleware,
15-3–15-4

conventional DOC middleware, **15-3**

CoSMIC model-driven deployment and
configuration, **15-8–15-9**

DAnCE system, **15-6–15-8**

inventory tracking, **15-11–15-12**

lessons learned from CIAO, DAnCE and
CoSMIC systems, **15-12–15-13**

quality-of-service-enabled component
middleware, **15-4, 15-13–15-14**

quality-of-service-enabled DOC systems, **15-3,
15-13**

real-time CORBA-lightweight CCM CIAO
integration, **15-5–15-6**

shipboard computing, **15-10–15-11**

operational flexibility, flexible time-triggered
(FTT) communication paradigm, **19-3**

quality-of-service management

aggregate management, **36-4–36-5**

basic principles, **36-2–36-3**

case studies, **36-12–36-30**

dynamic management, **36-5**

end-to-end management, **36-3**

evolution of, **36-1–36-2**

HiPer-D navy testbed prototype, **36-13–36-16**

middleware for dynamic management,
36-5–36-9

multilayer management, **36-3–36-4**

PCES end-to-end multilayered capstone
demonstration, **36-19–36-30**

software engineering, **36-9–36-12**

Weapon System Open Architecture hybrid
control and reactive management case
study, **36-16–36-19**

research and development issues, **15-2**

distributed real-time simulation, DeeDS NG
architecture, **29-5**

distributed systems

embedded systems model, **32-9–32-10**

synchronous programming limitations, **14-3**

distribution process, real-time databases, **24-3**

Domain Model Designer (DMD), Microsoft Domain
Model Designer (MDMD)

metaprogramming language, **33-12–33-13**

domain profiles, robot service communication
architecture, **18-9**

domain-specific abstractions, robot service
communication software, **18-11–18-12**

domain-specific modeling languages (DSMLs),
33-2–33-5

DRA algorithm, intertask voltage scaling, **6-8**

driver architecture, USB subsystems, **17-7–17-8**

dual-phase elementary cycle, flexible time-triggered
(FTT) communication paradigm,
19-5–19-6

dynamic branch prediction, embedded systems,
execution-time analysis, **35-7**

dynamic power management

flexible time-triggered (FTT) architecture,
19-11–19-12

heuristic policies, **6-9–6-10**

low-power embedded systems, **6-8–6-12**

operating system and cross-layer policies, **6-12**

stochastic policies, **6-10–6-12**

dynamic priority (DP) scheduling

multiprocessor sporadic task systems, **3-4–3-5**

priority-driven periodic real-time systems,
stochastic analysis, **9-10–9-12**

dynamic quality-of-service (QoS) management,
distributed real-time embedded systems,
36-5–36-9

dynamic reconfiguration

networked service robots, **18-3–18-4**

real-time databases, serialization order, **27-3**

dynamic resource management (DRM),

quality-of-service-enabled component
middleware, shipboard computing,
15-10–15-11

dynamic schedule updates, robust and implicit

earliest deadline first protocol, embedded
wireless networks, **22-9**

dynamic systems, real-time data distribution,
25-2–25-5

automotive applications, **39-4**

dynamic TAO, DRE systems, **15-13**

dynamic voltage scaling (DVS), power-aware
resource management, **6-2–6-8**

intertask scaling, **6-7–6-8**

intratask scaling, **6-3–6-7**

dynamic web data, real-time database dissemination,
24-20–24-21

E

earliest deadline first (EDF) algorithm. *See also*

robust and implicit earliest deadline first

Ada 2005 programming

constraints on, **13-18–13-19**

deadline support, **13-14–13-15**

mixed dispatching systems, **13-19–13-20**

- preemption level protocol, 13-15
- scheduling support, 13-15–13-19
- aperiodic task scheduling, 2-8–2-9
- ASCR process algebra, schedulability analysis, 31-17–31-19
- bounded-execution-time programming, 11-8–11-10
- compositional real-time framework
 - bounded-delay resource model, 5-9–5-11
 - periodic resource model, 5-11–5-12
 - sporadic task systems, 5-17
 - utilization bounds, 5-13–5-16
 - workload model, 5-5–5-7
- fixed-priority scheduling, 2-6–2-7
- flexible scheduling systems, 13-2–13-3
- hierarchical scheduling framework, 5-2
- intertask voltage scaling, 6-7–6-8
- multimedia mobile computing, real-time packet scheduling, 37-11–37-12
- multiprocessor sporadic task systems
 - dynamic priority scheduling, 3-4–3-5
 - fixed-job priority scheduling, 3-6–3-10
- priority-driven periodic real-time systems,
 - dynamic- and fixed-priority systems, 9-11–9-12
- rate-based execution algorithm, rate-based resource allocation, 4-8–4-9
- stochastic time demand analysis, 9-2
- synchronous programming, preemptive multitasking, 14-14–14-17
- Earliest Due Date rule, imprecise computation model, total weighted error, 7-4–7-6
- earliest eligible virtual deadline first (EEVDF) algorithm, rate-based resource allocation
 - constant bandwidth server, 4-8–4-9
 - fluid-flow model, proportional share scheduling, 4-6–4-7
- hybrid rate-based scheduling, 4-12–4-13
- overview, 4-2
- workload performance evaluation, 4-11–4-13
- eclipse modeling framework (EMF)
 - Ecore metaprogramming language, 33-11–33-12
 - evolution of, 33-2
- Ecore metaprogramming language
 - characteristics of, 33-11–33-12
 - evolution of, 33-1–33-2
- Elastic Task Model, flexible time-triggered (FTT) architecture
 - mobile robot control system, 19-19
 - QoS management, 19-12–19-14
- electronic control units (ECUs)
 - automotive embedded systems, 38-2
 - COMET automated configuration and analysis, 28-13–28-16
 - database services, tailorability dimensions, 28-2–28-3
- electronic engineering, synchronous programming, 14-2
- elementary cycle (EC), flexible time-triggered (FTT) communication paradigm
 - dual-phase protocol, 19-5–19-6
 - temporal parameters, 19-7–19-8
- Embedded Machine code, logical execution time programming, 11-13–11-14
- embedded systems. *See also* distributed real-time and embedded systems (DRE)
 - application-tailored databases, 28-3–28-6
- automotive applications
 - architecture innovations, 38-6–38-9, 38-11–38-12
 - coding systems, 38-9
 - competency improvements, 38-7
 - complex comprehensive data models, 38-8
 - complexity reduction, 38-12
 - cost issues, 38-6, 38-10–38-11
 - crash prevention and safety, 38-5
 - current practices, 38-2–38-3
 - development and maintenance processes, 38-8–38-10
 - domain profile, 38-3–38-4
 - driver assistance, 38-5
 - energy conservation, 38-5
 - functionality innovations, 38-4, 38-7–38-8
 - future applications, 38-4–38-6
 - hardware/technical infrastructure, 38-10
 - historical background, 38-2
 - human-machine interface adaptations, 38-5, 38-8
 - integrated tool support, 38-14–38-15
 - interconnection car networking, 38-6
 - maintenance systems, 38-9–38-10
 - overview, 38-1–38-2
 - personalization and individualization systems, 38-6
 - practical challenges, 38-6–38-11
 - process improvement, 38-13
 - programmable car development, 38-5
 - quality and reliability improvement, 38-15–38-16
 - requirements engineering, 38-8
 - research agenda, 38-16–38-18
 - seamless model-driven development, 38-13–38-14
 - software and systems integration, 38-9
- execution-time analysis
 - applications of, 35-2
 - evolution of, 35-1–35-2
 - hardware timing, 35-5–35-8
 - hybrid techniques, 35-15

- embedded systems. *See also* distributed real-time and embedded systems (DRE) (*Contd.*)
 - measurement techniques, 35-8–35-10
 - software behavior, 35-4–35-5
 - static analysis, 35-10–35-14
 - WCET analysis, 35-3–35-4, 35-15–35-17
- hardware/software codesign
 - evolution of, 34-1–34-2
 - system integration and, 34-6–34-7
- interrupt software guidelines, 16-9–16-11
- modular hierarchies
 - applications, 32-1–32-3
 - architecture design, interface composition, 32-19–32-20
 - compositionality of property refinement, 32-16
 - composition and combinations, 32-18–32-20
 - comprehensive modeling theory, 32-3–32-10
 - data model algebra, 32-4
 - distributed system model, 32-9–32-10
 - functional enhancement, service combination, 32-18–32-19
 - future research issues, 32-23–32-24
 - granularity refinement, 32-16–32-18
 - interface model, 32-6–32-9
 - interface structuring, 32-10–32-15
 - multifunctional system structuring, 32-11–32-14
 - property refinement, 32-15–32-16
 - refinement process, 32-15–32-18
 - service relations, 32-14–32-15
 - state machines, 32-5–32-6
 - syntactic interfaces, 32-4–32-5
 - timescale modeling, 32-30–32-22
- multimedia mobile computing
 - application-layer approaches, 37-9–37-14
 - mobility requirements, 37-3–37-4
 - overview, 37-1–37-2
 - portability, 37-2–37-3
 - system-layer approaches, 37-4–37-9
 - wireless networks, 37-3
- overview, 1-1–1-8
- real-time applications, 1-6–1-8
- wireless networks, real-time communication
 - approximate total capacity, 22-13
 - entity-aware transport, 22-11
 - evolution of, 22-1–22-2
 - predictable communication concepts, 22-2–22-3
 - RAP protocol, 22-9–22-10
 - real-time capacity, 22-11–22-12
 - robust and implicit earliest deadline first, 22-3–22-9
 - dynamic schedule updates, 22-9
 - power-aware RI-EDF, 22-7–22-8
 - rules, 22-5–22-7
 - single-path feasible region, 22-12
 - SPEED protocol, 22-10–22-11
- emplaced sensors, wireless sensor networks, 20-8
- emulators, execution-time analysis, embedded systems, 35-10
- end-to-end constraint/delay, temporal control, real-time systems, 10-3–10-4
- end-to-end MAC latency, sensor network messaging architecture, 21-14–21-16
- end-to-end quality-of-service (QoS) management
 - distributed real-time embedded systems, 36-3
 - PCES capstone program (DARPA) case study, 36-22–36-25
- end-to-end real-time performance, embedded wireless networks, 22-1
- energy efficiency
 - automotive embedded systems, management innovations, 38-5
 - multimedia mobile computing, 37-10–37-11
 - sensor network messaging architecture, 21-16–21-17
- engineering notations, embedded systems
 - construction, 32-22–32-23
- Enhanced Host Controller Interface (EHCI), USB system, 17-5–17-6
 - periodic requests, QoS guarantees, 17-9–17-15
- entity-aware transport, embedded wireless networks, 22-11
- equational theory, process algebras, 31-4
- equivalence computation
 - process algebras, 31-4
 - Time Petri Nets, clock domains, 30-10–30-11
- error correction
 - automotive embedded systems, 38-16
 - multimedia mobile computing, 37-13–37-14
- Esterel programming language
 - embedded systems, 32-22–32-23
 - synchronous programming, 14-6–14-17
 - temporal control, real-time systems, 10-15–10-17
- Ethernets
 - networked control performance, 11-14–11-17
 - robot service communication architecture, 18-5
- event-condition-action (ECA) rule, DeeDS NG systems, 29-2
- event-driven intratask dynamic voltage scaling, power-aware resource management, 6-6–6-7
- event-driven stochastic policies, dynamic power management, 6-10–6-11
- event handlers, real-time specification for Java, 12-7–12-8
- event-triggered systems
 - flexible time-triggered (FTT) communication paradigm, operational flexibility, 19-4
 - process algebras, 31-2

eventual consistency, DeeDS NG architecture, 29-6–29-7

execution steps, process algebras, resource-sensitive systems, 31-10

execution-time analysis, embedded systems

- applications of, 35-2
- evolution of, 35-1–35-2
- hardware timing, 35-5–35-8
- hybrid techniques, 35-15
- measurement techniques, 35-8–35-10
- software behavior, 35-4–35-5
- static analysis, 35-10–35-14
- WCET analysis, 35-3–35-4, 35-15–35-17

execution-time clocks, Ada 2005 programming and, 13-4–13-5

execution-time servers (ETS)

- Ada 2005 programming for, 13-8–13-13
- deferrable servers, 13-8–13-10
- sporadic server, 13-10–13-13
- defined, 13-2–13-3

execution-time timers, Ada 2005 programming language, 13-6

expansion theorem, process algebras, 31-4

expected MAC latency-per-unit-distance to the destination (ELD), sensornet messaging architecture, data-driven routing, 21-8–21-10

expected transmission count (ETX), sensornet messaging architecture, 21-14

explicit automaton, synchronous programming, 14-12–14-13

exponentially weighted moving average (EWMA), sensornet messaging architecture, learn-on-the-fly (LOF) protocol, 21-11–21-13

extension issues, compositional real-time framework, 5-16–5-18

F

fault tolerant architecture, automotive real-time systems, 39-5

feasibility testing

- imprecise computation model, 0/1-constraints, 8-6–8-7
- multiprocessor sporadic task systems, scheduling algorithm, 3-3
- real-time resource management and scheduling, 2-3
- real-time specification for Java, 12-5

file system, multimedia mobile computing, middleware technology for, 37-8–37-9

filter algorithms, adaptive cruise control systems, 39-6

fine-granularity scalable (FGS) coding, multimedia mobile computing, 37-12

firing domains, Time Petri Nets

- state space preservation, 30-5–30-7
- state spaces, 30-2–30-4
- strong state class graph construction, 30-9–30-13

firm temporal constraints, defined, 10-2

first-class object (FCO) classes, MetaGME modeling language, 33-10–33-11

first-fit-decreasing utilization (FFDU) algorithm, multiprocessor sporadic task systems

- fixed-job priority scheduling, 3-6–3-7
- fixed-task priority scheduling, 3-10–3-11

first-fit-increasing utilization (FFIU) algorithm, multiprocessor sporadic task systems, fixed-job priority scheduling, 3-6–3-7

first-in-first-out (FIFO) channels

- deterministically networked real-time systems, 11-17–11-19
- nondeterministically networked real-time systems, 11-19–11-21

fixed-job priority (FJP) scheduling, multiprocessor sporadic task systems, 3-4–3-10

fixed-priority scheduling

- aperiodic task scheduling, 2-8–2-9
- periodic task handling, 2-4–2-6
- priority-driven periodic real-time systems, stochastic analysis, 9-10–9-12

fixed-task priority (FTP) scheduling, multiprocessor sporadic task systems

- algorithms, 3-10–3-13
- defined, 3-4

flexible time-triggered (FTT) communication paradigm

- quality-of-service (QoS) support and analysis
 - asynchronous messaging system, 19-14–19-17
 - traffic model, 19-15
 - traffic schedulability analysis, 19-15–19-17
- dual-phase elementary cycle, 19-5–19-6
- mobile robot control system, 19-17–19-19
- communication requirements, 19-17–19-18
- online adaptation, 19-18–19-19
- operational flexibility, 19-2–19-4
- overview, 19-1–19-2
- synchronous messaging system, 19-8–19-14
 - dynamic management addition, 19-11–19-12
 - elastic task model, 19-12–19-14
 - traffic schedulability analysis, 19-9–19-11
- system architecture, 19-5
- system requirements database, 19-6–19-7
- temporal parameters, elementary cycle, 19-7–19-8
- real-time and embedded systems, 1-4–1-5

flexible validity intervals, real-time databases, freshness management, 24-13–24-14

flow analysis, execution-time analysis, embedded systems, 35-11

flowtype runtime support, SNBench platform, 23-19

fluid-flow allocation, rate-based resource allocation, 4-5-4-7

forward recovery, DeeDS NG architecture, 29-5

frame-dependent (FD) coding, segment-based intratask dynamic voltage scaling, 6-3-6-4

frame-independent (FI) coding, segment-based intratask dynamic voltage scaling, 6-3-6-4

frontend architecture, wireless sensor networks, 20-8

full resource utilization, static priority scheduling, 4-4

functionality

- automotive embedded systems
 - architectural innovations, 38-7-38-8
 - innovations in, 38-4-38-6
 - user perspectives, 38-11
- embedded systems, service combination, 32-18-32-19
- multifunctional systems, 32-12-32-13
- service hierarchies, 32-12

fusion algorithms, adaptive cruise control systems, 39-5-39-7

G

generalized density, sporadic task systems, 3-2

generalized Liu and Layland style allocation, rate-based resource allocation, 4-5

- execution, 4-7-4-9
- workload performance, 4-11-4-12

generalized processor sharing (GPS), rate-based resource allocation, 4-5

generic modeling environment (GME), automated configuration and analysis, 28-12-28-16

Generic Modeling Environment (GME)

- metaprogrammable tool
 - MetaGME modeling language, 33-10-33-11
 - model-integrated computing, 33-5-33-6
 - MOF integration, 33-13-33-14

genetic algorithms, hardware/software codesign, 34-5

geographic distance, sensor network messaging architecture

- data-driven routing, 21-8
- per-hop measurements, 21-15-21-16

geographic forwarding algorithm, wireless sensor networks, 20-3-20-4

Giotto programming language

- logical execution time programming, 11-13-11-14
- temporal control, real-time systems, 10-16-10-18

global criticality/local phase (GCLP) algorithm, hardware/software codesign, 34-4-34-5

global load balancers (GLBs), distributed data services, quality-of-service management, 24-14-24-15

global scheduling, multiprocessor sporadic task systems

- dynamic priority scheduling, 3-3, 3-5
- fixed-job priority scheduling, 3-7-3-10

global time, real-time process algebras, 31-5

globally asynchronous and locally synchronous architecture, deterministically networked real-time systems, 11-18-11-19

GMeta programming language, 33-5-33-6

graceful/uniform degradation

- real-time databases, quality-of-service management, 24-13-24-14
- static priority scheduling, 4-4

granularity

- real-time data distribution, 25-4-25-5
- refinement of, 32-16-32-17

graph of state classes (SCG), Time Petri Nets algorithm, 30-6-30-7

- LTL property preservation, 30-7-30-8
- state and trace preservation, 30-8-30-13

graph partitioning/optimal, SNBench platform, 23-20

greatest degradation first (GDF), multimedia mobile computing, 37-11-37-12

GReAT metaprogrammable tool, model transformations, 33-8-33-9

grid infrastructure, multimedia mobile computing, 37-5-37-7

Grid Resource Allocation for Mobile Services (GRAMS), multimedia mobile computing, 37-7

ground jobs, priority-driven periodic real-time systems, backlog dependency tree, 9-7-9-9

group budgets, Ada 2005 programming language, 13-7-13-8

GUARD-link configuration, COMET database platform, 28-9

H

halt-point identification, synchronous programming, explicit automata, 14-12

hard temporal constraints

- defined, 10-2
- real-time databases, transaction scheduling and concurrency, 24-6-24-7
- real-time data distribution, 25-3-25-5

hardware platform

- automotive embedded systems
 - architectural requirements, 38-12
 - design criteria, 38-9
 - technical infrastructure, 38-10

embedded systems, execution-time analysis, 35-5-35-8
 bounded timing, 35-11-35-12
 robot service communication architecture, 18-5
 hardware/software codesign
 automotive embedded systems, 38-12
 cosynthesis algorithms, 34-4-34-5
 CPU customization, 34-5-34-6
 embedded systems integration of, 34-6-34-7
 evolution of, 34-1-34-2
 partitioning algorithms, 32-2-34-4
 hardware tracers, execution-time analysis, embedded systems, 35-9
 heterogeneous algebra, data models, 32-4
 heuristic policies, dynamic power management, 6-9-6-10
 hierarchical scheduling framework. *See also* modular hierarchies
 applications, 5-1-5-3
 high-resolution timers, execution-time analysis, embedded systems, 35-9
 high-volume data, robot service communication software, streaming support, 18-11
 HiPer-D quality-of-service management platform, U.S. Navy testbed case study, 36-13-36-16
 host controllers, USB system, 17-5
 HTL programming language, logical execution time programming, 11-13-11-14
 human-machine interface (HMI)
 automotive embedded systems, 38-5-38-6
 architectural innovations, 38-8
 wireless sensor networks, 20-8-20-9
 hybrid analysis techniques, worst-case execution time analysis, 35-15
 hybrid earliest deadline first (EDF-US) algorithm, multiprocessor sporadic task systems, 3-7-3-8
 hybrid intratask dynamic voltage scaling, power-aware resource management, 6-6-6-7
 hybrid rate-based scheduling, rate-based resource allocation, 4-12-4-13

I

identical processors, imprecise computation model, total weighted error, 7-6-7-7
 implementation-defined schedulers, real-time specification for Java, 12-9
 implicit automaton, synchronous programming, 14-12-14-14
 implicit-deadline systems, multiprocessor sporadic task systems
 dynamic priority scheduling, 3-5

 fixed-job priority scheduling, 3-6
 fixed-task priority scheduling, 3-10-3-13
 global scheduling, 3-7-3-8, 3-11-3-13
 implicit path enumeration technique (IPET), worst-case execution time, embedded systems, 35-12-35-14
 implicit temporal control, real-time systems, 10-6-10-7
 imprecise computation model
 applications, 7-1-7-2
 bicriteria problems, 8-1-8-2
 maximum weighted error, 7-9-7-13
 maximum *w-Weighted* error constraints, 8-3-8-6
 real-time databases, quality-of-service management, 24-13-24-14
 total weighted error, 7-3-7-9
 parallel and identical processors, 7-6-7-7
 single processor, 7-3-7-6
 uniform processors, 7-7-7-9
 total *w-Weighted* error constraints, 8-2-8-3
 0/1-constraints, 8-6-8-10
 individualized cars, automotive embedded systems, 38-6
 infinitely fast machine paradigm, synchronous programming, 14-4
 infrastructure development, automotive embedded systems, 38-10, 38-16
 in-network database, wireless sensor networks, 20-8
 input/output (I/O) behavior
 embedded systems, interface model, 32-8-32-9
 granularity refinement, 32-17-32-18
 logical execution time programming, 11-12-11-14
 real-time databases, 24-11-24-12
 Universal Serial Bus (USB) systems, 17-2
 zero-execution-time programming, 11-10-11-12
 instantaneous events, process algebras, resource-sensitive systems, 31-10
 integrated tool support, automotive embedded systems, 38-14-38-15
 interacting task systems, compositional real-time framework, 5-17-5-18
 interconnection car networking, automotive embedded systems, 38-6
 interface definition language (IDL), application-tailored databases, 28-4-28-6
 interfaces
 automotive embedded systems, 38-15
 compositional real-time framework, 5-3
 embedded systems, 32-6-32-9
 architectural composition, 32-19-32-20
 refinements, 32-15-32-18
 structuring of, 32-10-32-15
 robot service communication architecture, core framework, 18-8-18-9

interference analysis algorithm, priority-driven
 periodic real-time systems, 9-4-9-7
 complexity evaluation, 9-20-9-22
 computational complexity, 9-16

internet routing, wireless sensor networks, 20-2-20-4

interpreted architectures, distributed systems, 32-10

interruptions, execution-time analysis, embedded
 systems, 35-9

interrupt systems
 definitions, 16-2-16-3
 embedded software guidelines, 16-9-16-11
 overload problems, 16-6-16-8
 process algebras, discrete-time modeling,
 31-7-31-8
 real-time analysis, 16-8-16-9
 real-time and embedded software, evolution of,
 16-1-16-2
 semantics, 16-3-16-4
 stack overflow problems, 16-4-16-6

intertask voltage scaling, power-aware resource
 management, 6-7-6-8

intratask voltage scaling, power-aware resource
 management
 miscellaneous techniques, 6-6-6-7
 path-based intraDVS, 6-4-6-6
 segment-based intraDVS, 6-3-6-4

inventory tracking systems (ITS),
 quality-of-service-enabled component
 middleware, 15-11-15-12

investment strategies, automotive embedded systems,
 38-11

IppsEDF/IppsRM algorithms, intertask voltage
 scaling, 6-8

irreconcilably conflicting transactions (ICTs),
 real-time databases
 concurrency control, 27-1-27-3
 high data contention, 27-11-27-12
 low resource contention, 27-12-27-13
 OCC-APR technique, 27-3-27-5
 serialization order, dynamic adjustment, 27-3
 simulation model, 27-6-27-11

iterative approximation, priority-driven periodic
 real-time systems, steady-state backlog
 analysis, 9-15

J

Java programming language. *See also* real-time
 specification for Java
 Ada programming language comparisons,
 13-20-13-21

Java RMI system, distributed object computing
 applications, 15-3

jitter accumulation
 networked control performance, 11-16-11-17

nondeterministically networked real-time systems,
 11-19-11-21
 stopwatch model of temporal control, 10-5-10-6

Just-In-Time scheduling (JITS) algorithm, sensor
 networks, real-time data services, 24-18

K

Kansei sensornet testbed, sensornet messaging
 architecture, 21-6-21-7

Kokyu scheduling and dispatching service, Weapon
 System Open Architecture (WSOA) case
 study, 36-17-36-19

L

labeled transition system (LTS), process algebras,
 31-1

lag minimization, rate-based resource allocation,
 fluid-flow model, proportional share
 scheduling, 4-6-4-7

language differentiators, metaprogramming
 languages, 33-13

last-in first out (LIFO), nondeterministically
 networked real-time systems, 11-19-11-21

latency, interrupt systems, 15-2-15-3

learn-on-the-fly (LOF) protocol, sensornet
 messaging architecture, data-driven
 routing, 21-10-21-13

legacy DeeDS NG architecture, 29-3-29-4

let assignments, SNAFU programming language,
 23-6

level-i busy window, flexible time-triggered (FTT)
 communication paradigm, asynchronous
 traffic schedulability analysis, 19-15-19-17

life-cycle management, automotive software
 engineering, 38-3

lightweight CORBA Component Model (CCM),
 quality-of-service-enabled component
 middlewear, real-time CORBA integration,
 15-5-15-6

linear bounded arrival process (LBAP), rate-based
 execution algorithm, rate-based resource
 allocation, 4-8-4-9

Linux system
 real-time data distribution service, 25-14-25-17
 USB subsystem on, 17-3-17-5
 periodic requests, QoS guarantees, 17-9-17-15

Liu/Layland bound (LL-bound)
 flexible time-triggered (FTT) architecture,
 synchronous messaging system, 19-11
 multiprocessor sporadic task systems, 3-6
 periodic task scheduling, 2-5-2-6

Liu/Layland periodic task model
 compositional real-time framework, 5-4-5-7
 hierarchical scheduling framework, 5-2

liveness, Time Petri Nets, **30-5**
 local time, real-time process algebras, **31-5**
 logical execution time (LET) programming
 compositional real-time framework, **5-3**
 real-time systems, **11-12-11-14**
 synchronous programming, **14-4**
 logic analyzers, execution-time analysis, embedded systems, **35-9**
 long timing effects, embedded systems,
 execution-time analysis, **35-5-35-6**
 low-level analysis, embedded systems, execution-time analysis, **35-11-35-12**
 low-power embedded systems, power-aware resource management
 dynamic power management, **6-8-6-12**
 dynamic voltage scaling, **6-2-6-8**
 intertask voltage scaling, **6-7-6-8**
 intratask voltage scaling, **6-3-6-7**
 overview, **6-1-6-2**
 low resource contention, real-time databases,
 irreconcilably conflicting transactions,
 27-12-27-13
 Lustre programming language
 embedded systems, **32-22-32-23**
 synchronous programming, **14-7-14-9**
 LYCOS algorithm, hardware/software codesign, **34-5**

M

maintenance systems, automotive embedded applications, **38-8-38-10**
 managing isolation in replicated real-time object repositories (MIRROR) protocol, real-time databases, **24-9-24-10**
 MANET routing, wireless sensor networks, **20-2-20-4**
 marking domains, Time Petri Nets, state space preservation, **30-5-30-8**
 Markov chain, priority-driven periodic real-time systems, steady-state backlog analysis, **9-13-9-14**
 Markov matrix truncation method, priority-driven periodic real-time systems
 approximated solutions, **9-14-9-15**
 computational complexity, **9-17**
 master interrupt enable bit, defined, **15-2**
 maximal progress
 process algebras, resource-sensitive systems, **31-9-31-19**
 real-time process algebras, **31-5**
 maximum weighted error (MWE), imprecise computation model, **7-9-7-13**
 bicriteria scheduling issues, **8-3-8-6**
 measurement techniques, execution-time analysis, embedded systems, **35-8-35-10**
 medium access control (MAC) protocol
 embedded wireless networks, **22-1-22-2**
 predictable wireless communication, **22-2-22-3**
 sensornet messaging architecture, **21-4-21-6**
 data-driven routing, **21-7-21-8**
 wireless sensor networks, **20-1-20-2**
 memory access times, embedded systems,
 execution-time analysis, **35-5**
 memory-aware intratask dynamic voltage scaling,
 power-aware resource management,
 6-6-6-7
 memory management
 COMET automated configuration and analysis,
 28-13-28-16
 multimedia mobile computing, **37-4-37-5**
 real-time specification for Java, **12-3-12-5**,
 12-13-12-16
 message description list (MEDL), deterministically networked real-time systems, **11-18-11-19**
 MetaGME modeling language
 characteristics of, **33-10-33-11**
 evolution of, **33-2**
 GME metaprogrammable tool, **33-6**
 metamodeling languages
 comparisons, **33-9-33-14**
 defined, **33-3-33-4**
 differentiators, **33-14**
 Ecore, **33-11-33-12**
 MetaGME, **33-10-33-11**
 metaprogrammable tools interface with,
 33-14-33-16
 Microsoft Domain Model Designer, **33-12-33-14**
 overview of, **33-1-33-2**
 metaprogrammable tools
 API-UDM model manipulation management,
 33-6-33-7
 DESERT design space exploration tool, **33-7-33-8**
 GME modeling tool, **33-5-33-6**
 GReAT model transformation tool, **33-8-33-9**
 metamodeling language interface with,
 33-14-33-16
 overview of, **33-3-33-5**
 metaprogramming language, defined, **33-3**
 Microsoft Domain Model Designer (MDMD)
 metaprogramming language,
 characteristics of, **33-12-33-13**
 middleware technology. *See also* component
 middleware technology
 conventional component middleware, **15-3-15-4**
 conventional DOC systems, **15-3**
 distributed real-time embedded systems,
 quality-of-service management, **36-5-36-9**
 qoskets and qosket components, **36-6-36-9**
 quality objects, **36-5-36-6**
 multimedia mobile computing

- middleware technology. *See also* component
 - middleware technology (*Contd.*)
 - file systems and, 37-8–37-9
 - grid infrastructure, 37-5–37-7
- networked service robots, reference middleware
 - architecture, 18-1–18-12
- sensor networks, real-time data services,
 - 24-17–24-18
- wireless sensor networks, 20-6–20-7
- Milner's Calculus of Communicating Systems,
 - synchronous programming, 14-3–14-5
- minimum-cost-maximum-flow technique, imprecise
 - computation model
 - total weighted error, 7-3–7-6
 - uniform processors, 7-7–7-9
- minimum interval time, interrupt systems, 16-9
- "misbehaved" task management, static priority
 - scheduling, 4-3–4-4
- miss ratios
 - COMET database platform, 28-11
 - real-time databases, quality-of-service
 - management, 24-16–24-17
- mixed dispatching systems
 - Ada 2005 programming, 13-19–13-20
 - synchronous programming limitations, 14-3
- mobile real-time databases, basic properties,
 - 24-18–4-20
- mobile robot control system, flexible time-triggered
 - (FTT) architecture, 19-17–19-19
- mobility
 - multimedia mobile computing, 37-37-4
 - wireless sensor networks, 20-4
- model-driven engineering (MDE)
 - automotive embedded systems, 38-13–38-14
 - evolution of, 33-1–33-2
- model-integrated computing (MIC)
 - API-UDM metaprogrammable tool, 33-6–33-7
 - evolution of, 33-1–33-2
 - GME metaprogrammable tool, 33-5–33-6
 - GReAT metaprogrammable tool, 33-7–33-8
 - metamodeling languages and metaprogrammable
 - tools
 - architectures and metaprogrammability,
 - 33-3–33-5
 - overview, 33-1–33-2
- mode-specific data and tasks, automotive real-time
 - systems, 39-4, 39-7
 - robotic vehicle controls, 39-17
- modular hierarchies, embedded systems
 - applications, 32-1–32-3
 - architecture design, interface composition,
 - 32-19–32-20
 - compositionality of property refinement,
 - 32-16
 - composition and combinations, 32-18–32-20
 - comprehensive modeling theory, 32-3–32-10
 - data model algebra, 32-4
 - distributed system model, 32-9–32-10
 - functional enhancement, service combination,
 - 32-18–32-19
 - future research issues, 32-23–32-24
 - granularity refinement, 32-16–32-18
 - interface model, 32-6–32-9
 - multifunctional system structuring, 32-11–32-14
 - property refinement, 32-15–32-16
 - refinement process, 32-15–32-18
 - service relations, 32-14–32-15
 - state machines, 32-5–32-6
 - structuring interfaces, 32-10–32-15
 - syntactic interfaces, 32-4–32-5
 - timescale modeling, 32-30–32-22
- MOF metaprogramming language
 - evolution of, 33-2–33-4
 - GME integration, 33-14–33-16
- MonitorControl hierarchy, real-time specification for
 - Java, resource sharing and synchronization,
 - 12-9–12-11
- More-Less approach, real-time databases, data timing
 - and freshness, 24-5–24-6
- MPEG coding, multimedia mobile computing
 - energy efficiency, 37-10–37-11
 - real-time packet scheduling, 37-12
- m*-processor schedule, scheduling algorithm, 3-3
- multicast communication model, point-to-point
 - real-time data distribution, 25-5–25-6
- multicore systems, execution-time analysis,
 - 35-7–35-8
- multienabledness, Time Petri Nets, 30-8
- multifunctional systems, interface structuring,
 - 32-11–32-14
- multilayer quality-of-service (QoS) management,
 - distributed real-time embedded systems,
 - 36-3–36-4
- multimedia systems, embedded mobile computing
 - application-layer approaches, 37-9–37-14
 - mobility requirements, 37-3–37-4
 - overview, 37-1–37-2
 - portability, 37-2–37-3
 - system-layer approaches, 37-4–37-9
 - wireless networks, 37-3
- multiaperiodic systems, synchronous programming
 - limitations, 14-3
- multiprocessor sporadic task systems
 - real-time data distribution, 25-15–25-16
 - schedulability analysis
 - definitions and models, 3-2–3-4
 - dynamic processor scheduling, 3-4–3-5
 - fixed job-priority scheduling, 3-5–3-10
 - fixed task-priority scheduling, 3-10–3-13
 - sporadic model relaxation, 3-13–3-15

synchronous programming, dataflow coding,
14-18–14-19
multiprocessor systems, execution-time analysis,
35-7–35-8
multisensor data fusion research, sensor networks,
real-time data services, 24-17–24-18
mutual consistency, real-time databases, data timing
and freshness, 24-5–24-6

N

nested interrupts, 16-3
networked control systems (NCS), programming
languages, 11-14–11-17
network flow approach, imprecise computation
model, total weighted error, 7-6–7-7
networking systems, automotive embedded systems,
38-6
NIL inactive process, process algebras
discrete-time modeling, 31-6–31-8
resource-sensitive systems, 31-10–31-13
node localization, wireless sensor networks,
20-4–20-5
nonblocking write (NBW) protocol,
bounded-execution-time programming,
11-9–11-10
noncritical mode (NC), automotive adaptive cruise
control, 39-8–39-10
nondeterministically networked real-time systems
(NNRTS)
programming languages, 11-14, 11-19–11-21
synchronous programming, Boolean causality,
14-10–14-11
nonground jobs, priority-driven periodic real-time
systems, backlog dependency tree, 9-7–9-9
nonmaskable interrupt (NMI), defined, 16-3
nonpreemptability, sporadic task model relaxation,
3-13–3-14
non-real-time requirements table (NRT), flexible
time-triggered (FTT) communication
paradigm, 19-7
schedulability analysis, 19-15–19-17
nontardy units (NTU), imprecise computation
model, total weighted error, 7-3–7-6
normalization, Time Petri Nets, clock domains, 30-11
normalized advance (NADV), sensornet messaging
architecture, 21-18
null execution time paradigm, synchronous
programming, 14-4

O

Object Management Group (OMG), real-time data
distribution service, 25-11–25-13
object request broker (ORB) command line

deployment and configuration
quality-of-service-enabled component
middleware, 15-7–15-8
open ORB project, 15-13
Object STore (OBST) manager, legacy DeeDS NG
architecture, 29-3–29-4
on-demand task scheduling, real-time automotive
applications, 39-12
One-One approach, real-time databases, data timing
and freshness, 24-5–24-6
Open CCM project, 15-14
operating systems
dynamic power management, 6-12
execution-time analysis, embedded systems, 35-10
flexible time-triggered (FTT) communication
paradigm, flexibility, QoS support for,
19-2–19-4
multimedia mobile computing, 37-4–37-5
rate-based resource allocation, scheduling
protocols, 4-9–4-11
real-time and embedded systems, 1-4–1-5
robot service communication architecture,
18-5–18-7
optimistic concurrency control-adaptive priority
(OCC-APR), real-time databases
performance evaluation, 27-6–27-12
serialization order, 27-3–27-5
optimistic concurrency control-early discard
(OCC-ED), real-time databases
irreconcilably conflicting transactions, 27-5–27-6
performance evaluation, 27-6–27-12
optimistic concurrency control (OCC), real-time
databases, 27-1–27-3
optimistic divergence control, COMET database
platform, 28-9
optimistic replication, DeeDS NG architecture, 29-4,
29-7–29-9
ORB service configuration options, deployment and
configuration quality-of-service-enabled
component middleware, 15-7–15-8
ordered binary decision diagrams (OBDDs),
DESERT metaprogrammable tool, 33-8
oscilloscope measurements, execution-time analysis,
embedded systems, 35-9
overload management
interrupt systems, 16-6–16-8
real-time scheduling, 2-12–2-14

P

packet-by-packet generalized processor sharing
(PGPS), rate-based resource allocation, 4-5
packet reception rate and distance (PRD), sensornet
messaging architecture, 21-14

- paradoxical interface behavior, embedded systems, 32-9
- parallel composition operator, process algebras, 31-3
- parallel processors, imprecise computation model, total weighted error, 7-6-7-7
- partitioned scheduling, multiprocessor sporadic task systems
 - dynamic priority scheduling, 3-3-3-5
 - fixed-job priority scheduling, 3-5-3-6
 - fixed-task priority scheduling, 3-10-3-11
- partitioning algorithms
 - hardware/software codesign, 34-2-34-4
- Time Petri Nets
 - branching properties preservation, 30-13
 - strong classes, 30-14
- path-based estimates, worst-case execution time, embedded systems, 35-12-35-13
- path-based intratask dynamic voltage scaling, power-aware resource management, 6-4-6-6
- patient events, process algebras, discrete-time modeling, 31-6-31-8
- PCES capstone program (DARPA), quality-of-service management case study, 36-19-36-30
 - distributed real-time embedded system construction, 36-22-36-25
 - separate subsystem development, 36-25-36-27
 - system requirements, 36-19-36-22
- PEARL programming language
 - bounded-execution-time programming, 11-8-11-10
 - temporal control, real-time systems, 10-8-10-9
- pending interrupts, defined, 15-2
- performance counters, execution-time analysis, embedded systems, 35-9
- performance evaluation
 - PCES capstone program (DARPA)
 - quality-of-service management case study, 36-27-36-30
 - priority-driven periodic real-time systems, 9-20-9-22
 - real-time databases, quality-of-service management, 24-14-24-15
 - robot service communication software, 18-10
 - optimization metrics, 18-11
 - SNBench platform, 23-19-23-20
 - workload performance, 4-11-4-13
- performance requirements, static priority scheduling, 4-3
- periodic adaptation, overload management, 2-13-2-14
- periodic query (PQuery) model, real-time databases, quality-of-service management, 24-16-24-17
- periodic recurrence, temporal control, real-time systems, 10-4
- periodic requests
 - real-time databases
 - data timing and freshness, 24-4-24-6
 - hard scheduling constraints, 24-6-24-7
 - real-time data distribution, 25-3-25-6
 - USB subsystem architecture, QoS guarantees, 17-8-17-15
- periodic resource model, compositional real-time framework, 5-8
 - schedulability analysis, 5-11-5-12
 - utilization bounds, 5-14-5-16
- periodic task handling
 - compositional real-time framework, workload model, 5-5-5-7
 - real-time scheduling and resource management, 2-3-2-7
- period modification policy, USB systems
 - binary encoding algorithm, 17-13
 - QoS guarantees, 17-9-17-15
- period multiple relationship, compositional real-time framework, utilization bounds, 5-14-5-16
- personalized cars, automotive embedded systems, 38-6
- pfair scheduling, multiprocessor sporadic task systems, implicit deadline systems, 3-5
- physical-execution-time (PET) programming, real-time systems, 11-6
- pipelining architecture, embedded systems, interface composition, 32-19-32-20
- 2PL locking protocol, real-time databases, 24-10
- pluggable schedulers, real-time specification for Java, 12-9
- point-to-point real-time data distribution, 25-5-25-6
- polling tree components, USB 1.x subsystem, 17-13-17-15
- portability issues, multimedia mobile computing, 37-2-37-3
- Portable Executive for Reliable Control (PERC), 12-1
- power-aware resource management, low-power embedded systems
 - dynamic power management, 6-8-6-12
 - dynamic voltage scaling, 6-2-6-8
 - intertask voltage scaling, 6-7-6-8
 - intratask voltage scaling, 6-3-6-7
- power-aware robust and implicit earliest deadline first protocol, embedded wireless networks, 22-7-22-8
- power management hints (PMH), segment-based intratask dynamic voltage scaling, 6-3-6-4
- power management (PM)
 - dynamic power management, 6-8-6-12
 - wireless sensor networks, 20-5-20-6

- power management point (PMP), segment-based intratask dynamic voltage scaling, 6-3–6-4
- power state machine model, dynamic power management, 6-8–6-12
- precedence constraints, ASCR process algebra, schedulability analysis, 31-14–31-15
- precision parameters, real-time data distribution, 25-4–25-5
- predictability, real-time databases, transaction scheduling and concurrency, 24-6–24-9
- predictable scheduling algorithms, multiprocessor sporadic task systems, 3-4
- predictive policies, dynamic power management, 6-10
- preemption relations, process algebras, resource-sensitive systems, 31-13
- preemptive multitasking, synchronous programming, 14-14–14-17
- prefix operator, process algebras, 31-2
- prioritized transitions, process algebras, resource-sensitive systems, 31-13
- priority adaptation query resource scheduling (PAQRS), real-time databases, quality-of-service (QoS) support and analysis, 24-12–24-14
- priority assignments, ASCR process algebra, schedulability analysis, 31-17–31-19
- priority-based slack stealing, intertask voltage scaling, 6-7–6-8
- Priority Ceiling Protocol (PCP), shared-resource management, 2-11
- priority cognizant concurrency control algorithm, real-time databases, irreconcilably conflicting transactions, 27-5–27-6
- priority-driven periodic real-time systems
 - Ada 2005 programming, EDF constraints, 13-18–13-19
 - bounded-execution-time programming, 11-8–11-10
 - real-time databases, hard scheduling constraints, 24-6–24-7
 - stochastic analysis
 - backlog analysis algorithm, 9-5–9-6
 - backlog dependency tree, 9-7–9-10
 - backlog/interference analysis evaluation, 9-20–9-22
 - computational complexity, 9-15–9-17
 - dynamic-priority and fixed-priority systems, 9-10–9-12
 - framework properties, 9-4–9-12
 - future research issues, 9-22
 - interference analysis algorithm, 9-6–9-7
 - overview, 9-1–9-3
 - solution comparisons, 9-18–9-20
 - steady-state backlog analysis, 9-12–9-15
 - system model, 9-3–9-4
- priority inheritance
 - bounded-execution-time programming, 11-9–11-10
 - real-time databases, soft transactions, 24-8–24-9
- Priority Inheritance Protocol (PIP), shared-resource management, 2-10–2-11
- priority scheduler, real-time specification for Java, 12-5
- proactive distributed programming model, DRE systems, 15-13
- probabilistic guarantees
 - applications, 9-1–9-2
 - QoS analysis, USB subsystems, sporadic transfers, 17-15–17-18
- probabilistic neighbor switching, sensornet messaging architecture, learn-on-the-fly (LOF) protocol, 21-13, 21-18
- probabilistic time demand analysis (PTDA), priority-driven periodic real-time systems, applications, 9-2–9-3
- probability mass function (PMF), priority-driven periodic real-time systems
 - backlog analysis algorithm, 9-5–9-6
 - steady-state backlog analysis, 9-13–9-14
 - stochastic analysis, 9-3–9-4
- probe effect, execution-time analysis, embedded systems, 35-8
- process algebras
 - ATP example, 31-8–31-9
 - basic principles, 31-1
 - conventional formalisms, 31-1–31-4
 - discrete-time modeling, 31-5–31-8
 - resource-sensitive systems, 31-9–31-19
 - schedulability analysis, 31-13–31-19
 - syntax and semantics, 31-10–31-13
 - time-sensitive system modeling, 31-4–31-5
 - TPL example, 31-8–31-9
- process improvements, automotive embedded systems, 38-13
- processing group parameters, real-time specification for Java, 12-3–12-5
- profilers, execution-time analysis, embedded systems, 35-9–35-10
- programmable car development, automotive embedded systems, 38-5
- programming languages. *See also* specific programming languages, e.g. Ada
 - GMeta, 33-5–33-6
 - real-time programming
 - bounded-execution-time programming, 11-7–11-10
 - control engineering computing, 11-3–11-6

programming languages. *See also* specific programming languages, e.g. Ada (*Contd.*)
 deterministically networked real-time systems, 11-17–11-19
 embedded systems, 1-3–1-4
 evolution of, 11-1–11-3
 logical-execution-time programming, 11-12–11-14
 networked systems, 11-14–11-17
 nondeterministically networked real-time systems, 11-19–11-21
 physical-execution-time programming, 11-6
 zero-execution-time programming, 11-10–11-12
 SNAFU programming language, SNBench platform, 23-3–23-7
 synchronous programming, 14-5–14-14
 causality checking, 14-9–14-11
 code generation, 14-11–14-14
 Esterel, 14-6–14-7
 Lustre, 14-7–14-9
 temporal control, real-time systems
 basic principles, 10-7–10-8
 system comparisons, 10-19–10-20
 PROMPT protocol, real-time databases, 24-10
 property combinations, DeeDS NG architecture, 29-11–29-12
 property refinement, 32-15–32-16
 proportional fair queuing (PFQ), multimedia mobile computing, 37-12
 proportional share (PS) scheduling algorithm, rate-based resource allocation
 fluid-flow model, 4-5–4-7
 overview, 4-2
 workload performance, 4-11
 proxy caches, web data dissemination, 24-21

Q

QoS-enabled distributed objects (Qedo) project, 15-13–15-14
 qoskets and quosket components
 distributed real-time embedded systems, quality-of-service management, 36-6–36-9
 PCES capstone program (DARPA) case study, 36-22–36-25
 QoS management architecture (QMF), real-time databases, 24-13–24-14
 quality assurance
 automotive embedded systems, 38-9
 automotive embedded systems and, reliability improvements, 38-15–38-16
 quality objects (QuO) framework
 distributed real-time embedded systems, quality-of-service management, 36-5–36-6

DRE systems, 15-13
 HiPer-D quality-of-service management platform, 36-13–36-16
 Weapon System Open Architecture (WSOA) case study, 36-17–36-19
 quality-of-data (QoD), real-time databases, 24-3
 freshness management, 24-13–24-14
 quality-of-service (QoS) distributed object computing (DOC) middleware
 current models, 15-13
 evolution of, 15-3
 quality-of-service (QoS)-enabled component middleware
 avionics mission computing, 15-9–15-10
 current models, 15-13–15-14
 distributed real-time and embedded system applications, 15-4–15-6
 inventory tracking applications, 15-11–15-12
 shipboard computing applications, 15-10–15-11
 quality-of-service (QoS) management
 COMET database platform, 28-10–28-12
 distributed real-time embedded systems
 aggregate management, 36-4–36-5
 basic principles, 36-2–36-3
 case studies, 36-12–36-30
 dynamic management, 36-5
 end-to-end management, 36-3
 evolution of, 36-1–36-2
 HiPer-D navy testbed prototype, 36-13–36-16
 middleware for dynamic management, 36-5–36-9
 multilayer management, 36-3–36-4
 PCES end-to-end multilayered capstone demonstration, 36-19–36-30
 software engineering, 36-9–36-12
 Weapon System Open Architecture hybrid control and reactive management case study, 36-16–36-19
 flexible time-triggered communication
 asynchronous messaging system, 19-14–19-17
 dual-phase elementary cycle, 19-5–19-6
 mobile robot control system, 19-17–19-19
 operational flexibility, 19-2–19-4
 overview, 19-1–19-2
 synchronous messaging system, 19-8–19-14
 dynamic management addition, 19-11–19-12
 elastic task model, 19-12–19-14
 traffic schedulability analysis, 19-9–19-11
 system architecture, 19-5
 system requirements database, 19-6–19-7
 temporal parameters, elementary cycle, 19-7–19-8
 multimedia mobile computing, 37-7–37-8
 application-layer approaches, 37-11–37-14

- networked service robots, real-time capabilities, **18-4**
 - real-time databases
 - data stream management, **24-15–24-17**
 - distributed data services, **24-14–24-15**
 - management, **24-12–24-14**
 - metrics, **24-14**
 - quality of data and, **24-3**
 - USB systems
 - binary encoding algorithm, multiple QH approach, **17-13**
 - overview, **17-1–17-2**
 - periodic requests, **17-8–17-15**
 - period modification policy, **17-8–17-10**
 - sporadic transfers, probabilistic analysis, **17-15–17-18**
 - subsystem guarantees, **17-6–17-18**
 - workload reinsertion algorithm, **17-10–17-12**
 - quantization parameter (QP), multimedia mobile computing, energy efficiency, **37-10–37-11**
- ## R
-
- radar algorithms, adaptive cruise control systems, **39-6**
 - RAP protocol, embedded wireless networks, **22-9–22-10**
 - rate-based execution (RBE) algorithm, rate-based resource allocation, **4-2, 4-7–4-9**
 - rate-based resource allocation
 - fluid-flow model, proportional share scheduling, **4-5–4-7**
 - hybrid scheduling, **4-12–4-13**
 - Liu-Layland extensions, **4-7–4-8**
 - workload performance, **4-11–4-12**
 - overview, **4-1–4-2**
 - proportional share allocation workload
 - performance, **4-11**
 - sample workload, **4-9–4-11**
 - server-based allocation, constant bandwidth server, **4-8–4-9**
 - server-based allocation, workload performance, **4-12**
 - taxonomy, **4-4**
 - traditional static priority scheduling, **4-3–4-4**
 - rate-monotonic algorithm
 - bounded-execution-time programming, **11-8–11-10**
 - compositional real-time framework
 - utilization bounds, **5-13–5-16**
 - workload model, **5-7**
 - fixed-priority scheduling, **2-4–2-6**
 - hierarchical scheduling framework, **5-2**
 - interrupt systems, **16-9**
 - intertask voltage scaling, **6-7–6-8**
 - multiprocessor sporadic task systems, fixed-task priority scheduling, **3-10–3-13**
 - stochastic time demand analysis, **9-2**
 - reachability problems, Time Petri Nets, **30-5**
 - real-time adaptive resource manager (RT-ARM), Weapon System Open Architecture (WSOA) case study, **36-17–36-19**
 - real-time analysis
 - interrupt systems, **16-8–16-9**
 - networked service robots, **18-4**
 - wireless sensor networks, **20-4**
 - real-time capacity, embedded wireless networks, **22-11–22-13**
 - real-time component model (RTCOM), COMET database platform, **28-6–28-7**
 - real-time databases (RTDB), **1-5–1-6**
 - application-tailored databases
 - COMET approach, **28-6–28-16**
 - existing systems, **28-3–28-6**
 - overview, **28-1–28-2**
 - tailorability dimensions, **28-2–28-3**
 - automotive applications
 - adaptive cruise control, **39-5–39-7**
 - dual mode systems, **39-8–39-10**
 - mode-change experiments, **39-17**
 - on-demand task scheduling, **39-12**
 - overview, **39-1–39-2**
 - repository specifics, **39-10–39-11, 39-14–39-17**
 - research issues, **39-3–39-5**
 - robotic vehicle control experiment, **39-12–39-17**
 - task models, **39-11–39-12**
 - data, transaction, and system characteristics, **24-2**
 - data freshness and timing properties, **24-4–24-6**
 - distribution, **24-3**
 - dynamic web data dissemination, **24-20–24-21**
 - irreconcilable transactions
 - concurrency control, **27-1–27-3**
 - high data contention, **27-11–27-12**
 - low resource contention, **27-12–27-13**
 - OCC-APR technique, **27-3–27-5**
 - serialization order, dynamic adjustment, **27-3**
 - simulation model, **27-6–27-11**
 - mobile real-time databases, **24-18–24-20**
 - overview, **24-1–24-2**
 - priority cognizant CC algorithm, **27-5–27-6**
 - quality of service
 - data stream management, **24-15–24-17**
 - distributed data services, **24-14–24-15**
 - management, **24-12–24-14**
 - metrics, **24-14**
 - quality of data and, **24-3**
 - real-time data distribution and, **25-6–25-7**
 - related services, **24-3–24-4**
 - scheduling and transaction processing, **24-2–24-3**
 - sensor networks, **24-17–24-18**

- real-time databases (RTDB) (*Contd.*)
 - transaction processing, 24-6–24-12
 - distributed databases and commit protocols, 24-9–24-10
 - I/O and buffer management, 24-11–24-12
 - recovery issues, 24-10–24-11
 - scheduling and concurrency control, 24-6–24-9
- real-time data distribution (RTDD)
 - Common Object Request Broker Architecture, 25-7–25-11
 - current research, 25-13–25-16
 - defined, 25-1
 - OMG data distribution service, 25-11–25-13
 - point-to-point RTDD, 25-5–25-6
 - real-time databases, 25-6–25-7
 - system characteristics, 25-1–25-5
- real-time distance-aware scheduling, RAP protocol,
 - embedded wireless networks, 22-9–22-10
- real-time event service, Common Object Request Broker Architecture (CORBA), 25-7–25-09
- real-time execution, rate-based resource allocation
 - fluid-flow model, proportional share scheduling, 4-7
 - workload sample, 4-9
- real-time interface, compositional real-time framework, 5-4
- real-time notification service, Common Object Request Broker Architecture (CORBA), 25-9–25-11
- real-time operating system (RTOS)
 - adaptive cruise control, 39-6
 - robot service communication architecture, 18-6–18-7
 - synchronous programming, preemptive multitasking, 14-14–14-17
- real-time packet scheduling, multimedia mobile computing, 37-11–37-12
- real-time process algebras, time-sensitive systems, 31-4–31-5
- real-time programming, evolution of
 - bounded-execution-time programming, 11-7–11-10
 - control engineering computing, 11-3–11-6
 - deterministically networked real-time systems, 11-17–11-19
 - embedded systems, 1-3–1-4
 - evolution of, 11-1–11-3
 - logical-execution-time programming, 11-12–11-14
 - networked systems, 11-14–11-17
 - nondeterministically networked real-time systems, 11-19–11-21
 - physical-execution-time programming, 11-6
 - zero-execution-time programming, 11-10–11-12
- real-time specification for Java (RTSJ)
 - Ada programming language comparisons, 13-20–13-21
 - bounded-execution-time programming, 11-8–11-10
 - evolution of, 12-1–12-3
 - memory management, 12-13–12-17
 - resource sharing and synchronization, 12-9–12-11
 - schedulable objects, 12-3–12-9
 - asynchronous event handling, 12-7–12-8
 - future research issues, 12-8–12-9
 - real-time threads, 12-6–12-7
 - temporal control, 10-13–10-15
 - time values and clocks, 12-11–12-13
- Real-Time Static Scheduling Service (RTSS), current research, 25-15–25-17
- real-time systems
 - ASCR process algebra, schedulability analysis, 31-13–31-19
 - overview, 1-1–1-8
 - schedulability analysis
 - aperiodic task handling, 2-7–2-9
 - future research issues, 2-14–2-15
 - models and terminology, 2-2–2-3
 - overload management, 2-12–2-14
 - overview, 2-1–2-2
 - period adaptation, 2-13–2-14
 - periodic task handling, 2-3–2-7
 - priority ceiling protocol, 2-10
 - priority inheritance protocol, 2-9–2-10
 - resource reservation, 2-12–2-13
 - schedulability analysis, 2-10–2-12
 - shared resources, 2-9–2-12
 - temporal control
 - ARTS kernel, 10-11–10-13
 - Esterel, 10-15–10-17
 - future research issues, 10-18–10-20
 - Giotto programming, 10-17–10-18
 - implicit control systems, 10-6–10-7
 - Java specifications, 10-13–10-15
 - model parameters, 10-3–10-4
 - overview, 10-1–10-3
 - PEARL programming model, 10-7–10-8
 - programming applications, 10-7–10-18
 - stopwatch model, 10-4–10-6
 - ARTS kernel, 10-13
 - Esterel, 10-15–10-17
 - Giotto programming, 10-17–10-18
 - Java specifications, 10-14–10-15
 - PEARL programming, 10-9
 - temporal scopes principle, 10-9–10-11
 - USB subsystem architecture, 17-7–17-8
- Realtime Thread, real-time specification for Java, 12-6–12-7

- real-time workshop (RTW), compositional real-time framework, 5-3
- reconcilably conflicting transactions (RCTs), real-time databases, 27-3
- recovery issues
 - DeeDS NG architecture, 29-5
 - real-time databases, 24-10–24-11
- recovery node, robust and implicit earliest deadline first protocol, embedded wireless networks, 22-4–22-5
- Reed-Solomon (RS) coding, multimedia mobile computing, 37-12
- reference middleware architecture, networked service robots
 - basic requirements, 18-2–18-4
 - core frame work, 18-7–18-9
 - device abstraction and component models, 18-2–18-3
 - domain-specific abstractions, 18-11–18-12
 - dynamic reconfigurability, 18-3–18-4
 - future research issues, 18-10–18-12
 - hardware platform, 18-5
 - high-volume data streaming support, 18-11
 - performance optimization metrics, 18-11
 - real-time and QoS capabilities, 18-4
 - resource frugality, 18-4
 - software evaluation, 18-10
 - software motivations, 18-5
 - software structure, 18-5–18-7
- reference path, path-based intratask dyanamic voltage scaling, 6-5–6-6
- refinement function, timescale modeling, 32-21–32-22
- regression model configuration, COMET database platform, 28-11
- relational coarse partitioning, Time Petri Nets, branching properties preservation, 30-13–30-15
- relative temporal constraints, defined, 10-2
- release parameters, real-time specification for Java, schedulable objects, 12-3–12-5
- reliability
 - automotive embedded systems and, 38-15–38-16
 - wireless sensor networks, 20-3–20-4
- remaining predicted execution cycles (RPEC), path-based intratask dyanamic voltage scaling, 6-5–6-6
- replicated database, DeeDS NG architecture, 29-6–29-7
- reply intervals, QoS analysis, USB subsystems, sporadic transfers, 17-15–17-18
- requantization technique, multimedia mobile computing, energy conservation with, 37-10–37-11
- requirements engineering, automotive embedded systems, 38-8–38-10
 - adapted development process, 38-17
- resolution, execution-time analysis, embedded systems, 35-9
- resource allocation, rate-based methods
 - fluid-flow model, proportional share scheduling, 4-5–4-7
 - hybrid scheduling, 4-12–4-13
 - Liu-Layland extensions, 4-7–4-8
 - workload performance, 4-11–4-12
- overview, 4-1–4-2
- proportional share allocation workload performance, 4-11
- sample workload, 4-9–4-11
- server-based allocation, constant bandwidth server, 4-8–4-9
- server-based allocation, workload performance, 4-12
- taxonomy, 4-4
- traditional static priority scheduling, 4-3–4-4
- resource demand, compositional real-time framework, 5-4
- resource frugality, robotics systems, 18-4
- resource hiding operator, process algebras, resource-sensitive systems, 31-12–31-13
- resource management
 - real-time systems, 1-1–1-3
 - aperiodic task handling, 2-7–2-9
 - future research issues, 2-14–2-15
 - models and terminology, 2-2–2-3
 - overload management, 2-12–2-14
 - overview, 2-1–2-2
 - period adaptation, 2-13–2-14
 - periodic task handling, 2-3–2-7
 - priority ceiling protocol, 2-10
 - priority inheritance protocol, 2-9–2-10
 - resource reservation, 2-12–2-13
 - schedulability analysis, 2-10–2-12
 - shared resources, 2-9–2-12
 - Sensorium Service Dispatcher, 23-9–23-10
- resource models, compositional real-time framework, 5-7–5-8
- resource reservation, overload management, 2-12–2-13
- resource-sensitive systems, process algebras, 31-9–31-19
 - schedulability analysis, 31-13–31-19
 - syntax and semantics, 31-10–31-13
- resource sharing, real-time specification for Java, 12-9–12-11
- resource supply, compositional real-time framework, 5-4, 5-11–5-12
- restart time estimation, real-time databases, validating transactions, 27-4

reuse strategies, automotive embedded systems, **38-11**

robotic vehicle control, real-time databases and, **39-12-39-17**

robot software communications architecture (RSCA), networked service robots

- basic requirements, **18-2-18-4**
- core framework, **18-7-18-9**
- device abstraction and component models, **18-2-18-3**
- domain-specific abstractions, **18-11-18-12**
- dynamic reconfigurability, **18-3-18-4**
- evaluation, **18-10**
- future research issues, **18-10-18-12**
- hardware platform, **18-5**
- high-volume data streaming support, **18-11**
- motivations, **18-5**
- performance optimization metrics, **18-11**
- real-time and QoS capabilities, **18-4**
- resource frugality, **18-4**
- structure, **18-5-18-7**

robust and implicit earliest deadline first (RI-EDF)

- protocol, embedded wireless networks, **22-3-22-9**
- dynamic schedule updates, **22-9**
- power-aware RI-EDF, **22-7-22-8**
- rules, **22-5-22-7**

rollback recovery, DeeDS NG architecture, **29-5**

routing service

- sensornet messaging architecture, **21-18**
- wireless sensor networks, **20-2-20-4**

runtime types

- SNAFU programming language, **23-6**
- SNBench platform, **23-19**

S

safety critical mode, automotive adaptive cruise control, **39-8-39-10**

safety properties

- automotive embedded systems, crash prevention and safety, **38-5**
- Time Petri Nets, **30-16**

sample sensing applications, SNBench platform, **23-16-23-18**

sampld-data control

- sensornet messaging architecture, learn-on-the-fly (LOF) protocol, **21-10-21-11**
- synchronous programming, **14-2-14-3**

scalability

- DeeDS NG architecture, **29-4-29-5, 29-9-29-13**
- networked service robots, resource frugality and, **18-4**

SCAN algorithm, real-time databases, **24-11-24-12**

schedulability analysis

ASCR process algebra, **31-13-31-19**

compositional real-time framework

- bounded-delay resource model, **5-7, 5-9-5-11, 5-13-5-14**
- extension, **5-16-5-18**
- interacting task support, **5-17-5-18**
- overview, **5-1-5-3, 5-5-5-6**
- periodic resource model, **5-8, 5-11-5-12, 5-14-5-16**
- resource models, **5-7-5-8**
- schedulable workload utilization bounds, **5-12-5-16**
- scheduling unit, component, and interface, **5-3-5-4**
- sporadic task support, **5-17**
- system models, **5-4**
- workload models, **5-6-5-7**

flexible time-triggered (FTT) architecture

- asynchronous traffic, **19-15-19-17**
- synchronous traffic, **19-9-19-11**

interrupt systems, **16-8-16-9**

multiprocessor sporadic task systems

- definitions and models, **3-2-3-4**
- dynamic processor scheduling, **3-4-3-5**
- fixed job-priority scheduling, **3-5-3-10**
- fixed task-priority scheduling, **3-10-3-13**
- sporadic model relaxation, **3-13-3-15**

real-time systems, **1-1-1-3**

- aperiodic task handling, **2-7-2-9**
- future research issues, **2-14-2-15**
- models and terminology, **2-2-2-3**
- overload management, **2-12-2-14**
- overview, **2-1-2-2**
- period adaptation, **2-13-2-14**
- periodic task handling, **2-3-2-7**
- priority ceiling protocol, **2-10**
- priority inheritance protocol, **2-9-2-10**
- resource management, **2-3**
- resource reservation, **2-12-2-13**
- schedulability analysis, **2-10-2-12**
- shared resources, **2-9-2-12**

robust and implicit earliest deadline first protocol, embedded wireless networks, **22-7**

shared-resource management, **2-11-2-12**

USB systems, QoS guarantees, sporadic transfers, **17-15-17-18**

schedulability test, multiprocessor sporadic task systems, scheduling algorithm, **3-3**

schedulable objects, real-time specification for Java, **12-3-12-9**

- asynchronous event handling, **12-7-12-8**
- future research issues, **12-8-12-9**
- real-time threads, **12-6-12-7**

schedule-carrying code (SCC), logical execution time programming, **11-13-11-14**

- schedule drift, embedded wireless networks, robust and implicit earliest deadline first protocol, 22-4–22-5
- scheduler process, deterministically networked real-time systems, 11-19
- scheduling algorithms
 - Ada 2005 programming, earliest deadline first support, 13-15–13-19
 - embedded systems, interrupt software guidelines, 16-10
 - flexible time-triggered (FTT) architecture, synchronous messaging system, 19-9
 - multiprocessor sporadic task systems, 3-3–3-4
 - real-time specification for Java, applications, 12-2–12-9
- scheduling anomalies, execution-time analysis, embedded systems, 35-7
- scheduling overhead, sporadic task model relaxation, 3-14
- scheduling parameters
 - real-time databases, 24-2–24-3
 - soft transactions, 24-7–24-9
 - transaction processing, 24-6–24-9
 - real-time specification for Java, 12-4–12-5
- scheduling policy, real-time specification for Java, 12-5
- scheduling unit, compositional real-time framework, 5-3–5-4
- seamless model-driven development, automotive embedded systems, 38-13–38-14
- security
 - SNBench platform, 23-20
 - wireless sensor networks, 20-4
- segment-based intratask dynamic voltage scaling, power-aware resource management, 6-3–6-4
- self-tuning regulator (STR), COMET database platform, 28-11
- semantics
 - deterministically networked real-time systems, 11-17–11-19
 - embedded systems, architectural composition, 32-19–32-20
 - process algebras, 31-2
 - resource-sensitive systems, 31-10–31-13
 - stopwatch model of temporal control, 10-6
 - wireless sensor networks, 20-3–20-4
- semiautonomous database evolution system (SADES), application-tailored databases, 28-5–28-6
- Sensorium Execution Environment (SXE)
 - future research and development, 23-18–23-19
 - SNBench platform, 23-12–23-16
- Sensorium Service Dispatcher (SSD)
 - sample sensing applications, 23-16–23-18
- SNBench platform, 23-2–23-7
 - resource management, 23-9–23-10
 - STEP scheduling and dispatch, 23-10–23-12
- Sensorium Task Execution Plan (STEP)
 - sample sensing applications, 23-16–23-18
- Sensorium Service Dispatcher scheduling and dispatch, 23-10–23-12
- SNBench platform, 23-2–23-3, 23-7–23-9
 - admission and interpretation, 23-14–23-15
 - node evaluation, 23-15–23-16
 - program/node removal, 23-16
 - SXE implementation, 23-12–23-16
- sensornet messaging architecture (SMA)
 - application-adaptive scheduling, 21-5–21-6
 - application-adaptive structuring, 21-4–21-5
 - components, 21-2–21-3
 - data-driven adaptation, 21-11–21-13
 - data-driven link estimation and routing, 21-6–21-17
 - ELD routing metric, 21-8–21-10
 - experimental design, 21-13–21-14
 - experimental results, 21-14–21-17
 - location and neighborhood identification, 21-10 sampling, 21-10–21-11
 - LOF data-driven protocol, 21-10–21-13
 - overview, 21-1–21-2
 - probabilistic neighbor switching, 21-13
 - traffic-adaptive link estimation and routing, 21-3–21-4
- sensor networks. *See* distributed multitasking sensor networks; SNBench platform; wireless sensor networks
 - automotive real-time systems
 - adaptive cruise control systems, 39-5–39-7
 - data fusion, 39-4
 - data services, 24-17–24-18
 - real-time and embedded systems, 1-4–1-5
- sensor transactions, real-time databases, 24-4–24-6
- sequential coding, synchronous programming, 14-11–14-14
- sequential composition operator, process algebras, 31-2
- serialization order, real-time databases, dynamic adjustment, 27-3
- server-based allocation, rate-based resource allocation, 4-5
 - constant bandwidth server, 4-8–4-9
 - workload performance evaluation, 4-12
- service combination, embedded systems, 32-18–32-19
- service hierarchies
 - multifunctional system structuring, 32-12
 - system structuring in, 32-13–32-14
- service interface, multifunctional system structuring, 32-11–32-12

- service relationships, multifunctional systems, 32-14–32-15
- service time, compositional real-time framework, periodic resource model, 5-8
- Shannon-Nyquist sampling theorem, synchronous programming, 14-2–14-3
- shared interrupts, defined, 15-2
- shared resources, real-time scheduling and resource management, 2-9–2-12
- shipboard computing, quality-of-service-enabled component middleware, 15-10–15-11
- signal processing, synchronous programming, 14-2–14-3
- signal-to-noise ratio (SNR), embedded wireless networks, 22-2–22-3
- Signal programming language, embedded systems, 32-22–32-23
- similarity stack protocols (SSP), real-time databases, soft transactions, 24-9
- simulated recursion, SNAFU programming language, 23-5–23-6
- simulation DeeDS, real-time applications, 29-13–29-14
- simulators, execution-time analysis, embedded systems, 35-10
- single-path feasible region, embedded wireless network real-time capacity, 22-12
- single processor
 - imprecise computation model, total weighted error, 7-3–7-6
 - synchronous programming, preemptive multitasking applications, 14-14–14-17
- slack estimation and distribution, intertask voltage scaling, 6-7–6-8
- sleeping, in sporadic task systems, 3-13
- smallest-optional-execution-time-first algorithm, imprecise computation model, 0/1-constraints, 8-8–8-10
- SNAFU programming language, SNBench platform, 23-3–23-7
 - compilation tasks, 23-6–23-7
 - cycle-safe iteration, 23-5
 - let assignments, 23-6
 - runtime types, 23-6
 - sensorium task execution plan, 23-7–23-9
 - simulated recursion, 23-5–23-6
- SNBench platform
 - applications, 23-16–23-19
 - basic properties, 23-1–23-2
 - future research and development, 23-20–23-21
 - research and development issues, 23-2–23-3
 - sensorium execution environments, 23-12–23-16
 - sensorium service dispatcher, 23-9–23-12
 - sensorium task execution plan, 23-7–23-9
 - SNAFU programming language, 23-4–23-7
 - SN workBench platform, 23-3–23-4
- soft temporal constraints
 - defined, 10-2
 - real-time databases, transaction scheduling and concurrency, 24-7–24-8
 - real-time data distribution, 25-3–25-5
- software assembly descriptor (SAD), robot service communication domain profile, 18-8–18-9
- software behavior, embedded systems, execution-time analysis, 35-4–35-5
- flow analysis, 35-11
- software engineering
 - automotive embedded systems, 38-2–38-3
 - architectural requirements, 38-12
 - competency improvements, 38-7
 - design criteria, 38-9
 - system infrastructure and, 38-16
 - systems integration, 38-9
 - quality-of-service management, distributed real-time embedded systems, 36-9–36-12
- software package descriptors (SPDs), robot service communication domain profile, 18-8–18-9
- software radio domain (SRD), robot service communication architecture, 18-5
- source code, worst-case execution time analysis, 35-14
- specification-based temporal constraints
 - programming sources, 10-7–10-8
 - real-time Java specification, 10-13–10-15
- SpecSyn algorithm, hardware/software codesign, 34-5
- speedometer example, synchronous programming, 14-6–14-9
- SPEED protocol
 - embedded wireless networks, 22-10–22-11
 - sensor net messaging architecture, 21-18
 - real-time data services, 24-18
- speed update ratio, path-based intratask dynamic voltage scaling, 6-5–6-6
- split-convolve-merge procedure, priority-driven periodic real-time systems, interference analysis algorithm, 9-6–9-7
- split-transaction-based approach, USB systems, QoS guarantees, periodic requests, 17-10–17-15
- sporadic requests
 - real-time data distribution, 25-3–25-5
 - USB subsystem architecture, QoS guarantees, 17-8, 17-15–17-18
- sporadic server systems, Ada 2005 programming for, 13-10–13-13
- sporadic task systems
 - compositional real-time framework, 5-17
 - real-time specification for Java, 12-8–12-9
 - relaxations, 3-13–3-15
 - schedulability analysis, 3-2–3-3

- stack model, embedded interrupt systems, 16-11
- stack overflow, interrupt systems, 16-4–16-6
- Stanford real-time information processor (STRIP),
 - real-time databases, quality-of-service (QoS) support and analysis, 24-13–24-14
- state classes, Time Petri Nets
 - graphing of, 30-2
 - marking and trace preservation, 30-5–30-7
- state graph (SG), Time Petri Net behavior, 30-4
- state machine model, embedded systems, 32-5–32-6
 - future research issues, 32-22–32-23
- state space abstractions
 - embedded systems, 32-5–32-6
 - Time Petri Nets (TPNs)
 - atomic state class graph construction, 30-14–30-15
 - branching properties preservation, 30-13
 - clock domains, 30-9–30-11
 - computing experiments, 30-15–30-16
 - development of, 30-1–30-2
 - extensions, 30-17
 - firing schedules, 30-2–30-4
 - future research issues, 30-16–30-17
 - general theorems, 30-5
 - marking and tracing preservation, 30-5–30-7
 - SCG property preservation, 30-7–30-8
 - SSCG construction, 30-9–30-13
 - state and trace preservation, 30-8–30-12
 - strong class partitioning, 30-14
 - variations, 30-8
- state transitions, embedded systems, 32-5–32-6
- static configuration
 - DAnCe optimization of, 15-8
 - DeeDS NG architecture, 29-10–29-11
 - sensornet messaging architecture, data-driven routing, 21-8
- static priority scheduling
 - rate-based resource allocation
 - full resource utilization, 4-4
 - graceful/uniform degradation, 4-4
 - “misbehaved” task management, 4-3–4-4
 - overview, 4-1–4-2
 - performance mapping, 4-3
 - synchronous programming
 - distributed implementation, 14-18–14-19
 - preemptive multitasking, 14-14–14-17
- static timing analysis, execution-time analysis,
 - embedded systems, 35-10–35-11
- stationary backlog distribution, priority-driven
 - periodic real-time systems, 9-12
- statistical rate monotonic scheduling (SRMS),
 - stochastic analysis, 9-3
- steady-state backlog analysis, priority-driven periodic
 - real-time systems, 9-12–9-15
 - approximated solutions, 9-14–9-15
 - computational complexity, 9-17
 - exact solution, 9-12–9-14
 - solution comparisons, 9-18–9-20
 - stationary distribution, 9-12
 - truncation of exact solution, 9-15
- stochastic analysis
 - dynamic power management, 6-10–6-12
 - priority-driven periodic real-time systems
 - backlog analysis algorithm, 9-5–9-6
 - backlog dependency tree, 9-7–9-10
 - backlog/interference analysis evaluation, 9-20–9-22
 - computational complexity, 9-15–9-17
 - dynamic-priority and fixed-priority systems, 9-10–9-12
 - framework properties, 9-4–9-12
 - future research issues, 9-22
 - interference analysis algorithm, 9-6–9-7
 - overview, 9-1–9-3
 - solution comparisons, 9-18–9-20
 - steady-state backlog analysis, 9-12–9-15
 - system model, 9-3–9-4
- stochastic intratask dynamic voltage scaling,
 - power-aware resource management, 6-6–6-7
- stochastic time demand analysis (STDA),
 - priority-driven periodic real-time systems
 - applications, 9-2–9-3
 - solution comparisons, 9-18–9-20
- stopwatch model, temporal control, real-time systems
 - ARTS kernel, 10-13
 - basic principles, 10-4–10-6
 - Esterel programming, 10-16–10-17
 - Giotto programming, 10-17–10-18
 - PEARL programming, 10-9
 - real-time specification for Java, 10-14–10-15
 - temporal scope model, 10-9–10-11
- streaming support, robot service communication
 - software, high-volume data management, 18-11
- stretching-t-NTA, intertask voltage scaling, 6-7–6-8
- strong bisimulation, process algebras, 31-4
- strong state classes (SSCs), Time Petri Nets
 - emergence of, 30-2
 - partitioning of, 30-14–30-15
 - state space preservation, 30-8–30-13
- strong state class graph (SSCG), Time Petri Nets,
 - clock domains for, 30-8–30-13
- subcoders, multimedia mobile computing, energy
 - efficiency, 37-11
- subhistory ordering, embedded systems, interface
 - model, 32-8
- subtype-relation, embedded systems, 32-5
- supervisory control module, automotive real-time
 - systems, 39-4–39-5

supply bound function, compositional real-time framework, 5-4

bounded-delay resource model, 5-7

periodic resource model, 5-8

surveillance applications

PCES capstone program (DARPA) case study, 36-20–36-22

wireless sensor networks, 20-6–20-7

switching modes, automotive adaptive cruise control, 39-10

Synchronous Calculus of Communicating Systems (SCCS), synchronous programming, 14-3–14-5

synchronous language, embedded systems, 32-22–32-23

synchronous messaging system

embedded wireless networks, 22-2

flexible time-triggered (FTT) architecture, 19-8–19-14

dynamic management addition, 19-11–19-12

elastic task model, 19-12–19-14

traffic schedulability analysis, 19-9–19-11

synchronous programming

control and electronic engineering applications, 14-2–14-3

distributed implementations, 14-17–14-19

languages and compilers, 14-5–14-14

causality checking, 14-9–14-11

code generation, 14-11–14-14

Esterel, 14-6–14-7

Lustre, 14-7–14-9

Milner's calculus, communicating systems, 14-3–14-5

overview, 14-1–14-2

real-time specification for Java, 12-9–12-11

single-processor, preemptive multitasking, 14-14–14-17

zero-execution-time programming, 11-10–11-12

synchronous requirements table (SRT), flexible time-triggered (FTT) architecture

synchronous messaging system, 19-8–19-14

system requirements database, 19-6–19-7

syntactic extension, process algebras, discrete-time modeling, 31-5–31-8

syntactic interfaces, embedded systems, 32-4–32-5

architectural composition, 32-19–32-20

syntactic reject, synchronous programming, causality checking, 14-10–14-11

syntax, process algebras, 31-2

resource-sensitive systems, 31-10–31-13

synthetic utilization, embedded wireless network

real-time capacity, 22-12–22-13

system characteristics

automotive embedded systems, software/system infrastructure, 38-16

real-time databases, 24-2

real-time data distribution, 25-2–25-5

system-layer technology, multimedia mobile computing, 37-4–37-9

embedded operating systems and memory management, 37-4–37-5

grid infrastructure, 37-5–37-7

middleware and file system requirements, 37-8–37-9

quality-of-service management, 37-7–37-8

system requirements database (SRDB), flexible time-triggered (FTT) communication paradigm, 19-5–19-7

systems engineering, automotive embedded systems, 38-7–38-9

system utilization, priority-driven periodic real-time systems

stationary backlog distribution, 9-12

stochastic analysis, 9-3–9-4

T

table-driven schedulers, real-time databases, hard scheduling constraints, 24-6–24-7

tailorability dimensions, real-time databases, 28-2–28-3

target tracking and engagement operations, quality-of-service management case study, 36-19–36-22

task execution coding, ASCR process algebra, schedulability analysis, 31-15–31-16

task interdependency, sporadic task model relaxation, 3-13–3-14

task life cycle, temporal control, real-time systems, 10-3–10-4

task models, real-time automotive applications, 39-11–39-12

task scheduling

automotive adaptive cruise control, 39-10–39-12

automotive real-time systems, 39-3–39-4

imprecise computation model, 0/1-constraints, 8-8–8-10

task-switching overhead, sporadic task model relaxation, 3-14

task system changes, sporadic task model relaxation, 3-14

TDL programming language, logical execution time programming, 11-13–11-14

temporal control, real-time systems

ARTS kernel, 10-11–10-13

Esterel, 10-15–10-17

flexible time-triggered (FTT) communication paradigm, elementary cycle parameters, 19-7–19-8

future research issues, 10-18–10-20

- Giotto programming, **10-17–10-18**
- implicit control systems, **10-6–10-7**
- Java specifications, **10-13–10-15**
- model parameters, **10-3–10-4**
- overview, **10-1–10-3**
- PEARL programming model, **10-7–10-8**
- programming applications, **10-7–10-18**
- real-time databases, **27-1–27-3**
- real-time data distribution, **25-3–25-5**
- stopwatch model, **10-4–10-6**
 - ARTS kernel, **10-13**
 - Esterel, **10-15–10-17**
 - Giotto programming, **10-17–10-18**
 - Java specifications, **10-14–10-15**
 - PEARL programming, **10-9**
 - temporal scopes principle, **10-9–10-11**
- synchronous programming, Lustre programming language, **14-7–14-9**
- temporal scope model, temporal control, real-time systems, **10-9–10-11**
- testing-based memory allocation, interrupt systems, stack overflow management, **16-4–16-5**
- testing systems, automotive embedded systems, **38-17**
- time-cognizant extensions, real-time databases, soft transactions, **24-8–24-9**
- time determinism
 - embedded systems, interrupt software guidelines, **16-10–16-11**
 - process algebras
 - discrete-time modeling, **31-6–31-8**
 - resource-sensitive systems, **31-13**
 - real-time systems, **10-2–10-3**
- time frames, USB subsystems, **17-6–17-7**
- time-indexed semi-Markov decision process (TISM DP) model, dynamic power management, **6-11–6-12**
- Time petri Net Analyzer (TINA), **30-15–30-16**
- Time Petri Nets (TPNs), state space abstractions
 - atomic state class graph construction, **30-14–30-15**
 - branching properties preservation, **30-13**
 - clock domains, **30-9–30-11**
 - computing experiments, **30-15–30-16**
 - development of, **30-1–30-2**
 - extensions, **30-17**
 - firing schedules, **30-2–30-4**
 - future research issues, **30-16–30-17**
 - general theorems, **30-5**
 - marking and tracing preservation, **30-5–30-7**
 - SCG property preservation, **30-7–30-8**
 - SSCG construction, **30-9–30-13**
 - state and trace preservation, **30-8–30-12**
 - strong class partitioning, **30-14**
 - variations, **30-8**
- time-prefix operator, process algebras, discrete-time modeling, **31-5–31-8**
- time progress rules, process algebras, discrete-time modeling, **31-5–31-8**
- time-safe implementation, logical execution time programming, **11-12–11-14**
- timescale modeling, embedded systems, **32-20–32-22**
- time-sensitive systems, process algebra modeling, **31-4–31-5**
- time streams, embedded systems, interface model, **32-6–32-7**
- Time-To-Refresh (TTR) value, web data dissemination, **24-21**
- time-triggered architecture (TTA)
 - deterministically networked real-time systems, **11-18–11-19**
 - networked control performance, **11-14–11-17**
 - synchronous programming, distributed implementation, **14-17–14-18**
- time values and clocks
 - Ada 2005 programming language, **13-4–13-6**
 - real-time specification for Java, **12-11–12-13**
- Time Warp protocol, simulation DeeDS, **29-13–29-14**
- timed actions, process algebras, resource-sensitive systems, **31-10**
- Timed Petri nets, real-time and embedded systems, **1-6–1-7**
- timeline scheduling, periodic task handling, **2-3–2-4**
- timeout operator, process algebras, discrete-time modeling, **31-7–31-8**
- timeout policies, dynamic power management, **6-9–6-10**
- timing analysis
 - execution-time analysis, embedded systems
 - applications of, **35-2**
 - evolution of, **35-1–35-2**
 - hardware timing, **35-5–35-8**
 - hybrid techniques, **35-15**
 - measurement techniques, **35-8–35-10**
 - software behavior, **35-4–35-5**
 - static analysis, **35-10–35-14**
 - WCET analysis, **35-3–35-4, 35-15–35-17**
 - synchronous programming, **14-2**
- timing anomalies, execution-time analysis, embedded systems, **35-7**
- timing event, Ada 2005 programming language, **13-4**
- TinyOS protocol, power-aware robust and implicit earliest deadline first protocol *vs.*, **22-7–22-8**
- tool support integration, automotive embedded systems, **38-14–38-15**
 - adapted development, **38-18**
- total bandwidth server (TBS), rate-based resource allocation, **4-8–4-9**
- total capacity approximation, embedded wireless network real-time capacity, **22-13**

total error minimization algorithm, imprecise computation model, 0/1-constraints, **8-7-8-8**

total weighted error (TWE), imprecise computation model, **7-3-7-9**

 bicriteria scheduling issues, **8-1-8-3**

 parallel and identical processors, **7-6-7-7**

 single processor, **7-3-7-6**

 uniform processors, **7-7-7-9**

TPL process algebra, example of, **31-9**

trace analysis, execution-time analysis, embedded systems, **35-9**

tracking systems

 automotive applications, **39-7**

 wireless sensor networks, **20-6-20-7**

traffic-adaptive link estimation and routing (TLR), sensor network messaging architecture (SMA), **21-2-21-6**

transaction

 attempt, QoS analysis, USB subsystems, sporadic transfers, **17-15-17-18**

 DeeDS NG architecture, conflicts and compensation, **29-9**

 real-time databases

 characteristics, **24-2**

 input/output and buffer management, **24-11-24-12**

 processing, **24-2-24-3, 24-6-24-10**

 recovery issues, **24-10-24-11**

translation lookaside buffer (TLB), multimedia mobile computing, **37-5**

tree-based estimates, worst-case execution time, embedded systems, **35-12-35-13**

trigger message (TM), flexible time-triggered (FTT) communication paradigm, dual-phase elementary cycle, **19-6**

truncated solution, priority-driven periodic real-time systems, **9-15-9-17**

two-phase execution, real-time process algebras, **31-4-31-5**

typed channels, embedded systems, **32-4-32-5**

U

Unified Data Model (UDM)

 metaprogrammable manipulation, **33-6-33-7**

 metaprogrammable tools/metamodeling language integration, **33-13**

uniform processors, imprecise computation model, **7-7-7-9**

uninterpreted architectures, distributed systems model, **32-9-32-10**

UNION-FIND algorithm, imprecise computation model, total weighted error, **7-3-7-6**

unit-delay operator, process algebras, discrete-time modeling, **31-7-31-8**

Universal Serial Bus (USB) systems

m-frame/ μ -frame time frame, **17-7-17-8**

 quality of service (QoS) support and analysis

 binary encoding algorithm, multiple QH approach, **17-13**

 overview, **17-1-17-2**

 periodic requests, **17-8-17-15**

 period modification policy, **17-8-17-10**

 sporadic transfers, probabilistic analysis, **17-15-17-18**

 subsystem guarantees, **17-6-17-18**

 workload reinsertion algorithm, **17-10-17-12**

real-time driver architecture, **17-7-17-8**

subsystem properties, **17-3-17-6**

 USB 1.x subsystem, QoS guarantees, **17-13-17-15**

 system review, **17-2-17-3**

 topology example, **17-6-17-7**

unmanned aerial vehicles (UAVs)

 HiPer-D quality-of-service management case study of, **36-13-36-16**

 PCES capstone program (DARPA), quality-of-service management case study, **36-19-36-30**

update scheduling configuration, COMET database platform, **28-11**

urgency inversion, embedded wireless network real-time capacity, **22-12**

urgent events, process algebras, discrete-time modeling, **31-6-31-8**

URI naming protocol, SNBench platform, **23-20**

USB request block (URB), **17-5-17-6**

 periodic requests, QoS guarantees, **17-8-17-15**

USB schedule, components of, **17-4-17-5**

user transactions, real-time databases, **24-4-24-6**

utilization bounds

 compositional real-time framework, schedulable workload, **5-12-5-13**

 sporadic task systems, **3-2**

utilization updating, intertask voltage scaling, **6-7-6-8**

V

validating transactions, real-time databases, **27-3-27-5**

 performance evaluation, **27-6-27-12**

VDM semantics, embedded systems construction, **32-22-32-23**

velocity-monotonic scheduling (VMS), RAP protocol, embedded wireless networks, **22-10**

velocity response, real-time automotive applications,
 robotic vehicle controls, **39-14–39-17**
 verification process, automotive embedded systems,
 38-18
 VigilNet system, **20-6–20-7**
 virtual full replication, DeeDS NG architecture,
 29-4–29-5, 29-9–29-13
 virtual time scheduling, compositional real-time
 framework, bounded-delay resource
 model, **5-9–5-11**
 visibility, execution-time analysis, embedded systems,
 35-9
 voids, wireless sensor networks, **20-4**
 voltage scaling points (VSPs), path-based intratask
 dynamic voltage scaling, **6-5–6-6**
 Vulcan partitioning, hardware/software codesign,
 34-4

W

wait-free communication, real-time specification for
 Java, **12-11**
 wake/sleep schedule integration, wireless sensor
 networks, **20-3–20-4**
 Weapon System Open Architecture (WSOA)
 program, quality-of-service management
 case study, **36-16–36-19**
 web data, real-time database dissemination,
 24-20–24-21
 weighted fair queuing (WFQ)
 multimedia mobile computing, **37-12**
 rate-based resource allocation, **4-5**
 wildfire scenario, DeeDS NG systems, **29-2–29-3**
 Wired Token Network, control performance,
 11-15–11-17
 wireless sensor networks (WSNs)
 assisted living facilities, **20-7–20-9**
 basic properties, **20-1–20-2**
 clock synchronization, **20-5**
 embedded systems, real-time communication
 approximate total capacity, **22-13**
 entity-aware transport, **22-11**
 evolution of, **22-1–22-2**
 predictable communication concepts,
 22-2–22-3
 RAP protocol, **22-9–22-10**
 real-time capacity, **22-11–22-12**
 robust and implicit earliest deadline first,
 22-3–22-9
 dynamic schedule updates, **22-9**
 power-aware RI-EDF, **22-7–22-8**
 rules, **22-5–22-7**
 single-path feasible region, **22-12**
 SPEED protocol, **22-10–22-11**

MAC protocol, **20-2**
 messaging applications
 data-driven adaptation, **21-11–21-13**
 data-driven link estimation and routing,
 21-6–21-17
 ELD routing metric, **21-8–21-10**
 experimental design, **21-13–21-14**
 experimental results, **21-14–21-17**
 location and neighborhood identification, **21-10**
 sampling, **21-10–21-11**
 LOF data-driven protocol, **21-10–21-13**
 overview, **21-1–21-2**
 probabilistic neighbor switching, **21-13**
 multimedia mobile computing, **37-3**
 quality-of-service management, **37-7–37-8**
 node localization, **20-4–20-5**
 power management, **20-5–20-6**
 real-time and embedded systems, **1-4–1-5**
 routing, **20-2–20-4**
 sensornet messaging architecture, **21-2–21-6**
 surveillance and tracking applications, **20-6–20-7**
 workload interference, priority-driven periodic
 real-time systems, stochastic analysis,
 9-4–9-5
 workload model
 compositional real-time framework, **5-5–5-7**
 rate-based resource allocation, **4-2, 4-11**
 USB system, periodic requests, QoS guarantees,
 17-9–17-15
 workload reinsertion algorithm, USB systems,
 bandwidth utilization optimization,
 17-10–17-12
 worst-case execution path (WCEP), path-based
 intratask dynamic voltage scaling, **6-5–6-6**
 worst-case execution time (WCET)
 Ada 2005 programming language, **13-4–13-6**
 application-tailored databases, **28-6**
 bounded-execution-time programming,
 11-7–11-10
 COMET automated configuration and analysis,
 28-13–28-16
 embedded systems
 analysis tools, **35-15**
 estimate calculation, **35-12–35-13**
 evolution of, **35-1–35-4**
 hardware timing, **35-5–35-8**
 hybrid techniques, **35-15**
 industrial applications, **35-15–35-17**
 measurement techniques, **35-8–35-10**
 software behavior, **35-4–35-5**
 static analysis, **35-10–35-14**
 interrupt systems, **16-9, 16-11**
 intertask voltage scaling, **6-7–6-8**
 logical execution time programming,
 11-12–11-14

worst-case execution time (WCET) (*Contd.*)
 real-time resource management and scheduling,
 2-2-2-3
 synchronous programming, 14-2
 preemptive multitasking,
 14-14-14-17
worst-case interrupt latency, 16-3
worst-case job sequence, multiprocessor sporadic
 task systems, 3-4

“Write Once, Run Anywhere” principle, real-time
 specification for Java, 12-2-12-3

Z

0/1-constraints, imprecise computation model,
 8-6-8-10
zero-execution-time (ZET) programming, real-time
 systems, 11-10-11-12

Handbook of Real-Time and Embedded Systems

Real-time and embedded systems are essential to our lives, from controlling car engines and regulating traffic lights to monitoring plane takeoffs and landings to providing up-to-the-minute stock quotes. Bringing together researchers from both academia and industry, the **Handbook of Real-Time and Embedded Systems** provides comprehensive coverage of the most advanced and timely topics in the field.

The book focuses on several major areas of real-time and embedded systems. It examines real-time scheduling and resource management issues and explores the programming languages, paradigms, operating systems, and middleware for these systems. The handbook also presents challenges encountered in wireless sensor networks and offers ways to solve these problems. It addresses key matters associated with real-time data services and reviews the formalisms, methods, and tools used in real-time and embedded systems. In addition, the book considers how these systems are applied in various fields, including adaptive cruise control in the automobile industry.

With its essential material and integration of theory and practice, the **Handbook of Real-Time and Embedded Systems** facilitates advancements in this area so that the services we rely on can continue to operate successfully.

Features

- Offers real-time scheduling and resource management solutions, including rate-based resource allocation methods, a hierarchical scheduling framework, and management techniques for low-power embedded systems
- Addresses the critical need for reliable, efficient, and scalable real-time wireless communication, providing an overview of a first-generation implementation of snBench
- Presents methods and tools to optimize real-time databases and data services, including optimistic concurrency control early discard (OCC-ED) and DeeDS NG
- Illustrates different approaches and models for designing real-time systems

