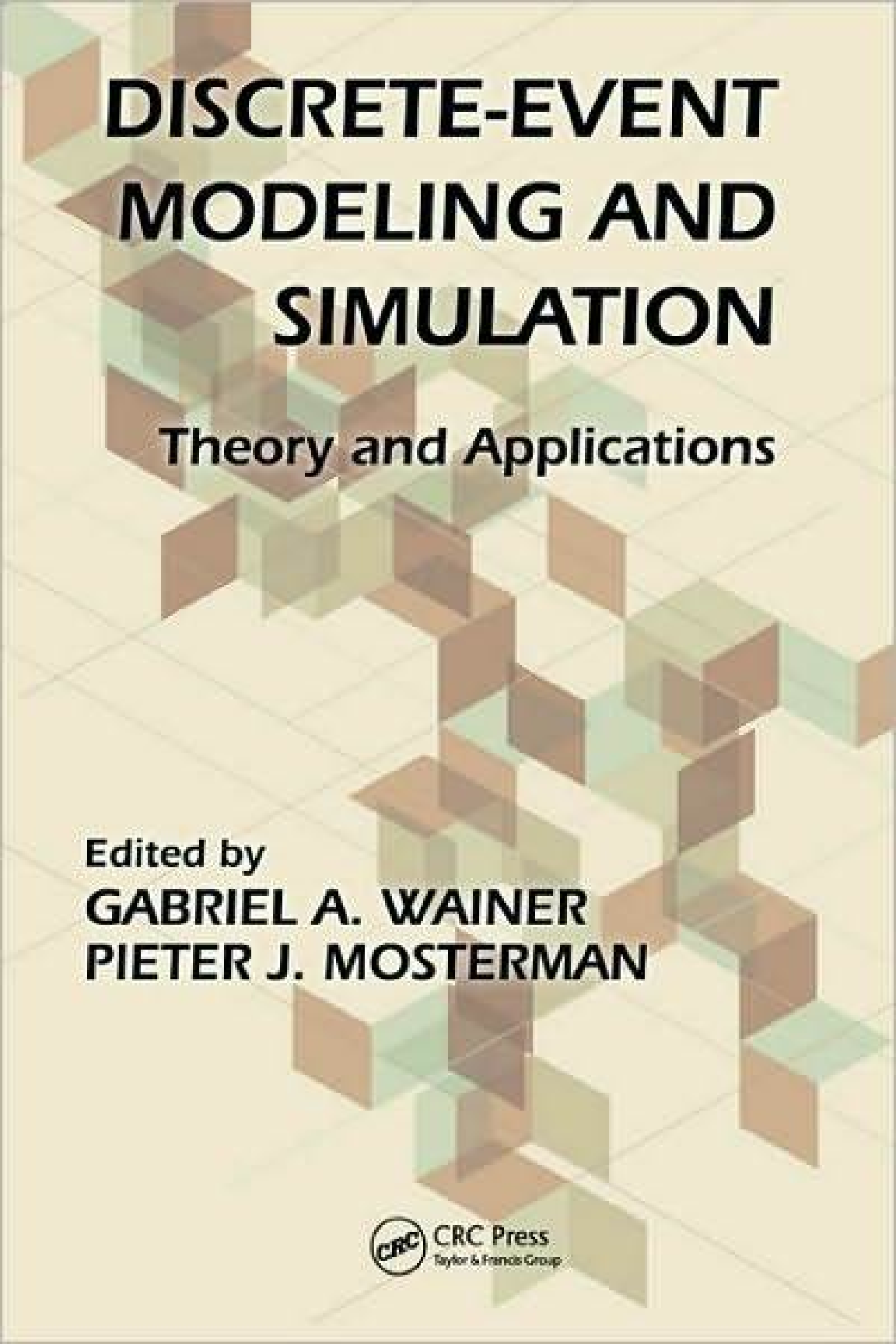




DISCRETE-EVENT MODELING AND SIMULATION

Theory and Applications

Edited by
GABRIEL A. WAINER
PIETER J. MOSTERMAN



CRC Press
Taylor & Francis Group

DISCRETE-EVENT MODELING AND SIMULATION

Theory and Applications

Computational Analysis, Synthesis, and Design of Dynamic Systems Series

Series Editor

Pieter J. Mosterman

MathWorks

Natick, Massachusetts

McGill University

Montréal, Québec

Discrete-Event Modeling and Simulation: A Practitioner's Approach,
Gabriel A. Wainer

Discrete-Event Modeling and Simulation: Theory and Applications,
edited by Gabriel A. Wainer and Pieter J. Mosterman

Model-Based Design for Embedded Systems,
edited by Gabriela Nicolescu and Pieter J. Mosterman

Model-Based Testing for Embedded Systems,
edited by Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman

Multi-Agent Systems: Simulation & Applications,
edited by Adeline M. Uhrmacher and Danny Weyns

Forthcoming Titles:

Real-time Simulation Technologies: Principles, Methodologies, and Applications,
edited by Katalin Popovici and Pieter J. Mosterman

DISCRETE-EVENT MODELING AND SIMULATION

Theory and Applications

Edited by

GABRIEL A. WAINER

PIETER J. MOSTERMAN



CRC Press

Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

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

© 2011 by Taylor and Francis Group, LLC
CRC Press 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: 978-1-4200-7233-4 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, 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/>) 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.

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

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

*In honor of Prof. Bernard P. Zeigler, the
pioneer of our field, and his 70th year.*

Contents

Foreword: The DEVS Core and Challenges to Its Continued Development xi
Bernard P. Zeigler

Contributors xxi

SECTION I Theory

Chapter 1 DEVS as a Semantic Domain for Programmed Graph Transformation 3
Eugene Syriani and Hans Vangheluwe

Chapter 2 System State Identification Using DEVS 29
Norbert Giambiasi, Diego Llarrull, and Maximiliano Cristiá

Chapter 3 A Formal Definition of Dynamic Structure Discrete-Flow Components..... 55
Fernando Barros

Chapter 4 Continuous System Simulation and Control 75
Ernesto Kofman, François E. Cellier, and Gustavo Migoni

SECTION II Methodology and Tools

Chapter 5 Heterogeneous Model Composability 111
Hessam S. Sarjoughian and Gary R. Mayer

Chapter 6 Effective and Efficient Modeling and Simulation with DEVS Variants 139
Adeline M. Uhrmacher, Jan Himmelspach, and Roland Ewald

Chapter 7 Object-Oriented Comodeling Methodology for Development of Domain Specific DEVS Models..... 177
Chang Ho Sung and Tag Gon Kim

Chapter 8	A Framework for Simulation-Based Structure and Parameter Optimization of Discrete-Event Systems	199
	<i>Olaf Hagendorf and Thorsten Pawletta</i>	
Chapter 9	Parallel Simulation of DEVS and Cell-DEVS Models in PCD++ ...	223
	<i>Gabriel A. Wainer, Qi Liu, and Shafagh Jafer</i>	
Chapter 10	From Virtual to Real—A Progressive Simulation-Based Design Framework.....	271
	<i>Xiaolin Hu</i>	

SECTION III Applications

Chapter 11	Formalization for Formal Verification of an Optical Network-on-Chip Using DEVS	293
	<i>Luiza Gheorghe Iugan, Gabriela Nicolescu, and Ian O'Connor</i>	
Chapter 12	Construction Simulation Using Cell-DEVS Modeling	309
	<i>Amin Hammad, Hong Pang, and Cheng Zhang</i>	
Chapter 13	Stochastic Modeling Strategies for the Simulation of Large (Spatial) Distributed Systems: Application to Fire Spread	331
	<i>Alexandre Muzy and David R.C. Hill</i>	
Chapter 14	Environmental Models in DEVS: Different Approaches for Different Applications	357
	<i>Jean-Baptiste Filippi, Teruhisa Komatsu, and David R.C. Hill</i>	

SECTION IV Standardization

Chapter 15	DEVS Standardization: Foundations and Trends.....	389
	<i>Gabriel A. Wainer, Khaldoun Al-Zoubi, Olivier Dalle, David R.C. Hill, Saurabh Mittal, José L. Risco Martín, Hessam Sarjoughian, Luc Touraille, Mamadou K. Traoré, and Bernard P. Zeigler</i>	

Chapter 16 An Introduction to DEVS Standardization 393
*Gabriel A. Wainer, Khaldoon Al-Zoubi, David R.C. Hill,
Saurabh Mittal, José L. Risco Martín, Hessam Sarjoughian,
Luc Touraille, Mamadou K. Traoré, and Bernard P. Zeigler*

Chapter 17 Standardizing DEVS Model Representation..... 427
*Gabriel A. Wainer, Khaldoon Al-Zoubi, David R.C. Hill,
Saurabh Mittal, José L. Risco Martín, Hessam Sarjoughian,
Luc Touraille, Mamadou K. Traoré, and Bernard P. Zeigler*

Chapter 18 Standardizing DEVS Simulation Middleware 459
*Gabriel A. Wainer, Khaldoon Al-Zoubi, Olivier Dalle, Saurabh
Mittal, José L. Risco Martín, Hessam Sarjoughian, and
Bernard P. Zeigler*

Index..... 495

Foreword: The DEVS Core and Challenges to Its Continued Development

Bernard P. Zeigler

Complex information-technology-based business, engineering, military systems, as well as the societal infrastructures they support, are at the root of this century's global challenges of economy, climate, and energy. We are accustomed to building such systems directly in the real world and letting subsequent use and Mother Nature tell us how good they are. Increasingly, however, it is becoming too dangerous, costly, unethical, or risky to do so. "Build and test within virtual reality" is increasingly the only workable alternative—whereby "virtual" we include a wide range of representations of the eventual fielded reality either wholly within a single computer or as networked distributed simulations, often enhanced with physically analogous and immersive environments. Modeling and simulation (M&S) supply the basis for such environments. Computer-based *modeling* refers to the construction of such environments while computer *simulation* connotes the experimentation using them to study alternative designs and architectures.

Introduced in the last century as a rigorous systems-theory basis for discrete event modeling and simulation, the DEVS (Discrete Event System Specification) formalism has become an engine for advances in M&S technology and the support of "build and test in virtual reality." A variety of systems-theory and artificial-intelligence applications employing DEVS-based concepts were compiled by Sarjoughian and Cellier [1]. However, the wealth of research continues to expand, so an updated accounting of recent advances is in order. This book obliges admirably by presenting advances in DEVS theory, applications, and methodology.

However, you as today's web-savvy reader will question what a pricey collection of papers on DEVS will provide that an apparently free Google search of the Internet will not. The value proposition of this book lies in the one-stop shopping and metaorganizational features that it provides. As an M&S researcher, you will get not only a sampling of the best research in the area, but also a broader overview of this research that provides a picture of the DEVS landscape—what has been done and what this has opened up to be done in the future. As an M&S practitioner, you will see a variety of exciting new applications that were enabled by the DEVS approach, and that may inspire you to make similar application advances in your own area of interest.

This introductory chapter is aimed at providing the metaorganization just mentioned. We start with a review of core DEVS concepts and pointers to where you

can pursue this background in greater depth. We will then relate the research and application contributions of this book to this core to provide a sense of where it all fits. Finally, we draw some inferences and make some predictions of where DEVS work will evolve to support “build and test within virtual reality.”

DEVS CORE

The core of DEVS consists of basic concepts, the broader M&S framework, and the systems-theory basis.

BASIC CONCEPTS

Discrete Event Systems Specification (DEVS) is a modeling formalism. Just as arithmetic underlies addition, multiplication, and other calculations, so DEVS underlies simulation of discrete event models.

DEVS Simulation is performed by a DEVS simulator or simulation engine that correctly implements an Abstract DEVS Simulator description that is technology agnostic. This is much like a calculator program executing an algorithm that describes the program operation in terms that abstract away the implementation details.

System Entity Structure (SES) is a high-level, ontology framework targeted to modeling, simulation, systems design, and engineering. An SES is a formal structure governed by a small number of axioms that provide clarity and rigor to its models. The structure supports hierarchical and modular compositions allowing large complex structures to be built in stepwise fashion from smaller, simpler ones.

M&S FRAMEWORK AND SYSTEMS-THEORY BASIS

The Modeling and Simulation Framework provides a set of entities (viz., source; also called real or referent) system, model, simulator, and experimental frame (EF), as well as relations among the entities (viz., model validity, simulator correctness, among others). In effect, this framework presents an ontology of the modeling and simulation domain. DEVS models can be atomic, coupled, and hierarchical. The Simulation Relation specifies what constitutes a correct simulation of a model by a simulator. The Model Relation determines when a model can be said to be a valid representation of a source system within an experimental frame. The framework

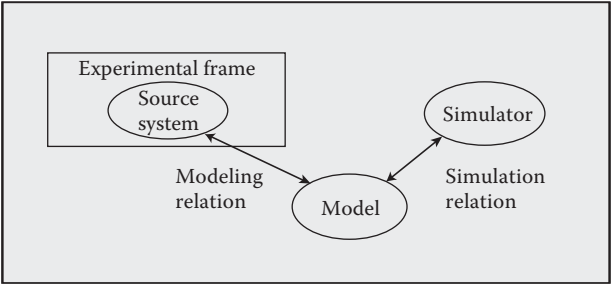


TABLE 1
Hierarchy of System Specifications

Level	Name	System Specification at This Level
4	Coupled systems	System built from component systems with a coupling recipe
3	I/O system structure	System with state and state transitions to generate the behavior
2	I/O function	Collection of input/output pairs constituting the allowed behavior partitioned according to initial state of the system
1	I/O behavior	Collection of input/output pairs constituting the allowed behavior of the system from an external black box view
0	I/O frame	Input and output variables and ports together with allowed values

entities are formulated in terms of the system specifications provided by systems theory (see below), and the framework relations are formulated in terms of the morphisms among system specifications. System specification morphisms are mappings between system specifications that preserve behaviors or structures of interest. They allow us to check consistency between models and frames, as well as between frames and models at different resolutions. This capability becomes especially important when components are composed together in distributed simulation federations. Computational support for this framework is based on the mathematical theory of systems and works hand-in-glove with object orientation and other computational paradigms.

Table 1 reviews the hierarchy of system specifications drawn from the *Theory of Modeling and Simulation* book [2]. Generally, transitioning from higher to lower levels in the specification hierarchy; that is, from structure to behavior, involves working out explicitly what is implicit in the given description. On the other hand going from lower to higher levels; that is, from behavior to structure, presents an underconstrained problem, requires adding new structure to what is given, and can be characterized as a search rather than a computation. The systems specification hierarchy provides a basis for understanding the types of problems that are encountered in applying modeling and simulation to systems development, engineering, management, and testing.

M&S LAYERED ARCHITECTURE

The M&S Layered Architecture is another organizing concept for placing the work in this book into understandable relationships. A brief overview of this architecture follows.

- Network Layer* includes networks, sensors, and computing devices that are organized in grids, clouds, and service-oriented architectures, and that do the work of supporting all aspects of the M&S life cycle.
- Simulation Layer* is the software that executes the models to generate and analyze their behaviors in both virtual and real time. Included in this layer

is the DEVS simulation protocol that provides the basis for correct local and distributed execution.

Data and Modeling Layer supports the development of data ontologies, experimental/pragmatic frames, and models in the DEVS formalisms that are separated from, but compliant with, the simulation layer protocol but agnostic with respect to its implementation.

Negotiation and Search Layer supports users in characterizing their data needs and negotiates with data sources to discover data to meet these specifications; supports the user in formulating experimental frames and in the discovery of model components meeting frame specifications; and supports searches through space of model compositions for candidates meeting frame-specified criteria.

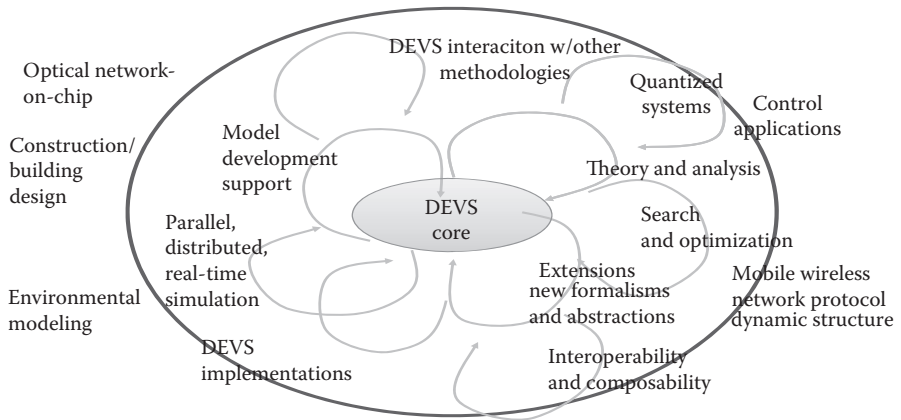
Domain Decision Layer enables the decision maker to interact with, and apply the results of, lower layer processing to make decisions and solve systems problems within the particular application domains that are being addressed.

Collaboration Layer enables participants to bring to bear individual perspectives and contributions to make decisions and solve problems based upon the functionalities provided by the underlying layers.

ADVANCES IN DEVS RADIATING OUT FROM THE CORE

This book provides chapters divided into theory, methodology, and applications that all relate to the DEVS core.

As illustrated in the figure, radiating out from the core are areas of research in theory and methodology, each with applications, as well as an outer ring of trend setting applications. Let’s consider how these areas relate to the M&S Layered Architecture. First we consider areas that relate to working with the architecture itself.



**Methodology for Working with the
M&S Architecture**

Areas Covered in This Book/Subtopics

- DEVS Model Development Support
 - Progressive model continuity
 - Plug-ins for software platform frameworks
 - Graph transformation model generation
 - DEVS Interaction with Other Methodologies
 - System design and development
-

Now consider how the areas are distributed among the layers in the table below. Notice that both the top and bottom layers are not directly covered in this book. Notwithstanding coverage in the open literature, this suggests some major open areas for future DEVS research—we will return to this idea below.

M&S Architectural Layer

Areas Covered in This Book/Subtopics

Network Layer

Simulation Layer

- Parallel, Distributed, Real-Time Simulation
 - Parallel simulation
 - Graph Transformation in Real Time
 - Exploiting DEVS variants
- DEVS Implementations (DEVSTJava, DEVS Suite, DevsSim++, James, CD++, DEVSPython, PowerDEVS, Web-Based Implementations)
- Interoperability and Composability
 - Multiformalism composability
 - DEVS Standardization

Data and Modeling Layer

- Extensions, New Formalisms, and Abstractions
 - Other system specifications
 - DEVS variants
 - DEVS subclasses
 - Activity abstraction concepts
- DEVS Theory and Analysis
 - System identification
 - Quantized systems

Negotiation and Search Layer

- Search and Optimization
 - SES-based search/model generation

Domain Decision Layer

- DEVS Applications
 - Optical network-on-chip
 - Construction/building design
 - Process control
 - Workflow systems
 - Environmental models

Collaboration Layer

Finally, we relate the areas and subtopics in the table to chapters in the book. The subtopics are shown in italics, and are followed by chapters that address them. A chapter may be referenced more than once. Each area and its subtopics are placed under the M&S Architecture layer to which they apply.

Methodology for Working with the M&S Architecture—Areas and subtopics here relate to the support of DEVS model development and the interaction with other methodologies.

DEVS Model Development Support

- *Progressive model continuity* (Hu, From Virtual to Real: A Progressive Simulation-Based Design Framework)
- *Plug-ins for software platform frameworks* (Uhrmacher/Himmelspach/Ewald, Effective and Efficient Modeling and Simulation of DEVS Variants)
- *Graph transformation model generation* (Syriani/Vangheluwe, DEVS as a Semantic Domain for Programmed Graph Transformation)

DEVS Interaction with Other Methodologies

- *System design and development* (Sung/Kim, Object-Oriented Comodeling Methodology for Development of Domain Specific DEVS Models)

Simulation Layer—Areas and subtopics here relate to different forms of simulation supported by DEVS, the use of real-time DEVS simulation in graphs transformation, and the influence of DEVS variants on simulation performance. For convenience, the important topic of interoperability is included in this layer, although it also relates to the modeling layer.

Parallel, Distributed, Real-Time Simulation

- *Parallel simulation* (Wainer/Liu/Jafer, Parallel Simulation of DEVS and Cell-DEVS Models in PCD++)
- *Graph transformation in real time* (Syriani/Vangheluwe, DEVS as a Semantic Domain for Programmed Graph Transformation)
- *Exploiting DEVS variants* (Uhrmacher/Himmelspach/Ewald, Effective and Efficient Modeling and Simulation of DEVS Variants)

DEVS Implementations

- *DEVSTJava, DEVS Suite* (Sarjoughian/Mayer, Heterogeneous Model Composability)
- *DevsSim++* (Sung/Kim, Object-Oriented Comodeling Methodology for Development of Domain Specific DEVS Models)
- *James* (Uhrmacher/Himmelspach/Ewald, Effective and Efficient Modeling and Simulation of DEVS Variants)
- *CD++* (Wainer/Liu/Jafer, Parallel Simulation of DEVS and Cell-DEVS Models in PCD++)
- *DEVSPython* (Syriani/Vangheluwe, DEVS as a Semantic Domain for Programmed Graph Transformation)

- *PowerDEVS* (Kofman/Migoni/Cellier, Continuous System Simulation and Control)
- *Web-based Implementations*, (Various Authors, DEVS Standardization: Ideas, Trends, and the Future)

Interoperability and Composability

- *Multi-formalism composability* (Sarjoughian/Mayer, Heterogeneous Model Composability)
- *DEVS standardization* (Various Authors, DEVS Standardization: Ideas, Trends, and the Future)

Data and Modeling Layer—Areas and subtopics here relate to enhancements to DEVS model expression (extensions, new formalisms, and abstractions) and conditions in which they apply. Also included are theory and analysis for real-world system identification and control that are facilitated by the close connection of DEVS to systems theory.

Extensions, New Formalisms, and Abstractions

- *Other system specifications* (Barros, A Formal Definition of Dynamic Structure Discrete-Flow Components)
- *DEVS variants* (Uhrmacher/Himmelspace/Ewald, Effective and Efficient Modeling and Simulation of DEVS Variants)
- *DEVS subclasses* (Filippi/Santucci/Komatsu/Hill, Environmental Models in DEVS: Different Approaches for Different Applications)
- *Activity abstraction concepts* (Muzy/Hill, Stochastic Modeling Strategies for the Simulation of Large (Spatial) Distributed: Application to Fire Spread)

DEVS Theory and Analysis

- *System identification* (Giambiasi/Llarull/Cristia, System State Identification using DEVS)
- *Quantized systems* (Kofman/Migoni/Cellier, Continuous System Simulation and Control)

Negotiation and Search Layer—A framework is developed to support generation and search for models that satisfy desired performance criteria. The framework is based on the system entity structure and its transformation to DEVS simulation models.

Search and Optimization

- *SES-based search/model generation* (Hagendorf/Pawletta, A Framework for Simulation Based Structure and Parameter Optimization of Discrete Event Systems)

Domain Decision Layer—A number of interesting examples are described that illustrate the potential use of DEVS to build successful applications following the methodology and layers of the M&S layered architecture.

DEVS Applications

- *Optical network-on-chip* (Gheorghe/Nicolescu/O'Connor, Formalization and Validation of an Optical Network-on-Chip Using DEVS and Timed Automata)
- *Construction/building design* (Hammad/Pang/Zhang, Construction Simulation Using Cell-DEVS Modeling)
- *Process control* (Kofman/Migoni/Cellier, Continuous System Simulation and Control)
- *Workflow systems* (Hagendorf/Pawletta, A Framework for Simulation-Based Structure and Parameter Optimization of Discrete Event Systems)
- *Environmental models* (Filippi/Santucci/Komatsu/Hill, Environmental Models in DEVS: Different Approaches for Different Applications)

CHALLENGES IN CONTINUED DEVS-BASED DEVELOPMENT

DEVS is finding increasing acceptance in the model-based simulation research community as one of the preferred paradigms to conduct modeling and simulation enquiries. Starting from a core set of formally stated concepts, the paradigm has elaborated into deeper theory, new variants, extensions and abstractions, and groundbreaking applications. Indeed, DEVS has become a base for the development of specialized modeling formalisms for specific domains that may be useful for a plethora of applications. Moreover, the strict separation of models and simulators has facilitated the development of alternative algorithms to dramatically speed up the simulation of DEVS models. Several challenges present themselves here: One is to maintain the interoperability of the variants among themselves and legacy models by strictly maintaining compliance to the DEVS core formalism. A second is to continue to develop powerful DEVS-based, domain-specific environments with associated methodologies and tools—while retaining compatibility with other such developments. An intriguing direction is to continue to employ DEVS as a semantic domain for other formalisms with the possibility of precise and powerful semantics as well as efficient and robust implementation that includes time (virtual and real) in a progressive development framework. Once again achievement of composability and interoperability of the heterogeneous model collections that emerge will be the distinguishing mark of the DEVS-based approach.

The increasing popularity of DEVS, in applications as well as in research, suggests that it exists at the right level of definition. It is both abstract enough to allow the development of new concepts and structures, as well as precise enough to avoid ambiguities and to support rigorous M&S applications. A challenge in continuing in this effort will be to find the right level of abstraction to suit new domains and applications. The real systems problems to be tackled (global warming, financial meltdown, etc.) and the hardware and software platforms available (Web 2.0, virtual reality environments, etc.) are growing in complexity at “exponential” rates.

To manage this accelerated growth in complexity, new refinements of the M&S Layered Architecture must be developed and automatic mappings between higher and lower layers must be designed to support easy to use and robust M&S services.

As mentioned earlier, the collaboration layer represents a placeholder in this book that should spur the development of DEVS-based theory, concepts, and tools to support the collaborative work needed to tackle systems complexity and to “build and test within virtual reality.” Moreover, an abstraction discussed in the book, the *activity concept*, relates to the absence of discussion of the network layer. This abstraction promises to provide an important refinement between the network layer and the simulation layer. The challenge is to exploit this refinement to facilitate better linkage between DEVS simulators and the underlying use of computational and energetic resources and thereby provide a quantum leap in the autonomic efficiency of DEVS simulations.

REFERENCES

1. Sarjoughian, H. S., and Cellier, F. E., eds. *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*. New York: Springer, 2001.
2. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, 2nd ed. Orlando, FL: Academic Press, 2000.

Contributors

Khaldoon Al-Zoubi

Department of Systems and Computer
Engineering
Carleton University Centre on
Advanced Visualization and
Simulation (V-Sim)
Carleton University
Ottawa, Canada

Fernando Barros

Departamento de Engenharia
Informática
Universidade de Coimbra
Coimbra, Portugal

François E. Cellier

Modeling and Simulation Research
Group
Department of Computer Science
ETH Zurich, Switzerland

Maximiliano Cristiá

Flowgate Consulting and CIFASIS
Rosario, Argentina

Olivier Dalle

Département Informatique UFR
Sciences, Université de Nice Sophia
Antipolis
Laboratoire I3S-UMR CNRS 6070
INRIA Sophia Antipolis
France

Roland Ewald

Institute of Computer Science
University of Rostock
Rostock, Germany

Jean-Baptiste Filippi

UMR CNRS 6134
Systèmes physiques pour
l'environnement
Università di Corsica -Pasquale Paoli
Corte, France

Luiza Gheorghe Iugan

Department of Computer and Software
Engineering
Ecole Polytechnique de Montréal
Montréal, Canada

Norbert Giambiasi

LSIS, Université Paul Cézanne
Marseille, France

Olaf Hagendorf

Research Group Computational
Engineering and Automation
Hochschule Wismar - University of
Applied Sciences
Wismar, Germany

Amin Hammad

Concordia Institute for Information
Systems Engineering
Concordia University
Montréal, Canada

David R.C. Hill

ISIMA/LIMOS/UMR CNRS 6158
Computer Science & Modeling
Laboratory
Université Blaise Pascal
Aubiere, France

Jan Himmelspace

Institute of Computer Science
University of Rostock
Rostock, Germany

Xiaolin Hu

Computer Science Department
Georgia State University
Atlanta, Georgia

Shafagh Jafer

Department of Systems and Computer
Engineering
Carleton University Centre on
Advanced Visualization and
Simulation (V-Sim)
Carleton University
Ottawa, Canada

Tag Gon Kim

Department of Electrical
Engineering
KAIST
Daejeon, Korea

Ernesto Kofman

Laboratorio de Sistemas Dinámicos
FCEIA, Universidad Nacional
de Rosario, CIFASIS,
CONICET
Rosario, Argentina

Teruhisa Komatsu

Ocean Research Institute
The University of Tokyo
Tokyo, Japan

Qi Liu

Department of Systems and Computer
Engineering
Carleton University Centre on
Advanced Visualization and
Simulation (V-Sim)
Carleton University
Ottawa, Canada

Diego Llarrull

LSIS, Université Paul Cézanne
Marseille, France

Gary R. Mayer

Department of Computer Science
Southern Illinois University
Edwardsville
Edwardsville, Illinois

Gustavo Migoni

Laboratorio de Sistemas
Dinámicos
FCEIA, Universidad Nacional
de Rosario, CIFASIS,
CONICET
Rosario, Argentina

Saurabh Mittal

Dunip Technologies
Tempe, Arizona

and

L-3 Communications
US Air Force Research Laboratory
711th Human Performance Wing
Mesa, Arizona

Alexandre Muzy

Laboratory LISA UMR 6240
Università di Corsica–Pasquale Paoli
Campus Mariani
Corti, France

Gabriela Nicolescu

Department of Computer and Software
Engineering
Ecole Polytechnique de Montréal
Montréal, Canada

Ian O'Connor

Lyon Institute of Nanotechnology
Ecole Centrale de Lyon
University of Lyon
Ecully, France

Hong Pang

Concordia Institute for Information
Systems Engineering
Concordia University
Montréal, Canada

Thorsten Pawletta

Research Group
Computational Engineering
and Automation
Hochschule Wismar - University of
Applied Sciences
Wismar, Germany

José L. Risco Martín

Department of Computer Architecture
and Automation
Universidad Complutense de Madrid
Madrid, Spain

Hessam S. Sarjoughian

Computer Science and Engineering
Department
Arizona State University
Tempe, Arizona

Chang Ho Sung

Department of Electrical
Engineering
KAIST
Daejeon, Korea

Eugene Syriani

School of Computer Science
McGill University
Montréal, Canada

Luc Touraille

ISIMA/LIMOS/UMR CNRS 6158
Computer Science & Modeling
Laboratory
Université Blaise Pascal
Aubiere, France

Mamadou K. Traoré

ISIMA/LIMOS/UMR CNRS 6158
Computer Science & Modeling
Laboratory
Université Blaise Pascal
Aubiere, France

Adeline M. Uhrmacher

Institute of Computer Science
University of Rostock
Rostock, Germany

Hans Vangheluwe

School of Computer Science
McGill University
Montréal, Canada

and

Department of Mathematics and
Computer Science
University of Antwerp
Antwerp, Belgium

Gabriel A. Wainer

Department of Systems and Computer
Engineering
Carleton University
Centre on Advanced
Visualization and
Simulation (V-Sim)
Carleton University
Ottawa, Canada

Cheng Zhang

Building, Civil, and Environmental
Engineering Department
Concordia University
Montréal, Canada

Bernard P. Zeigler

Department of Electrical & Computer
Engineering
University of Arizona
Tucson, Arizona

Section I

Theory

1 DEVS as a Semantic Domain for Programmed Graph Transformation

Eugene Syriani and Hans Vangheluwe

CONTENTS

1.1	Introduction	3
1.1.1	Meta-Modeling and Model Transformation	3
1.1.2	The Discrete Event System Specification	4
1.2	The AntWorld Simulation Case Study	7
1.2.1	The AntWorld Language (Abstract and Concrete Syntax)	8
1.2.2	The AntWorld Semantics (Graph Transformation)	9
1.3	A Meta-Model for DEVS-Based Graph Transformation	10
1.3.1	The <i>ARule</i>	11
1.3.2	The <i>CRule</i>	13
1.3.3	The <i>Selector</i>	13
1.3.4	The Synchronizer	14
1.4	Using MoTif for the AntWorld Simulator Case Study	15
1.4.1	The Round Block	16
1.4.2	The HumanStep Block	20
1.4.3	The Controller Block	20
1.4.4	The User Block	21
1.4.5	Simulation Results	21
1.5	Timed Graph Transformation	22
1.5.1	Introduction of Time	22
1.5.2	Real-Time Model Transformation and Deployment	24
1.6	Related Graph Transformation Tools	24
1.7	Conclusions	26
	References	27

1.1 INTRODUCTION

1.1.1 META-MODELING AND MODEL TRANSFORMATION

Model-driven approaches are becoming increasingly important in the area of software engineering. In model-driven development, models are constructed to conform to meta-models. A meta-model defines the (possibly infinite) set of all well-formed

model instances. As such, a meta-model specifies the syntax and static semantics of models. Meta-models are often described as the Unified Modeling Language (UML) Class Diagrams. In model-driven engineering, meta-modeling goes hand-in-hand with model transformation.

In almost all modeling and simulation endeavors, some form of model transformation is used. Models are, for example, transformed for optimization purposes, to address new requirements, to synthesize real-time embedded code, and so on. Transformations are also commonly used to describe the semantics of a modeling formalism. In the case of operational semantics, the transformation iteratively updates the state of a model. In the case of denotational semantics, the transformation maps a model in one formalism onto a model in a known formalism, thereby defining the meaning of the original model. Model transformations can be described in many ways. Rule-based descriptions are elegant and easy to understand. Such descriptions have declarative (specifying “what” to change, not “how” to) model rewriting rules as their primitive building blocks. A rule consists of a left-hand side (LHS) pattern that is matched against a host model. If a match is found, this pattern is updated, in the host model, based on what is specified in the right-hand side (RHS) of the rule. Additionally, Negative Application Condition (NAC) patterns may be used, specifying which patterns should not be found in the host model. Because at some level of abstraction, all models can be represented as (typed, attributed) graphs, and thanks to its rigorous formal underpinning, our rule-based specification is based on the theory of graph rewriting.

Though elegant, the declarative, rule-based specifications of model transformations do not scale well. When the number of rules grows, it becomes difficult for a modeler to clearly understand what the behavior of the transformation will be. Also, the complexity of matching grows with the number of rules that need to be tried. Programmed (or controlled) graph rewriting mitigates these problems. It combines rewriting rules with a control structure. In this chapter we show how the Discrete Event system Specification (DEVS) can be used as a semantic domain for the control structures in a model/graph transformation system.

In Section 1.2 we introduce our running example, an extended version of a recent benchmark for graph transformation [1]. Section 1.3 shows how extending the meta-model of DEVS allows for the introduction of programmed (or controlled) model/graph transformation. Then, Section 1.4 illustrates a solution to the case study problem using that transformation language. Using the notion of time inherent in DEVS, we show how the notion of time can elegantly be added to a transformation ultimately allowing real-time deployment in Section 1.5. Section 1.6 compares our DEVS-based approach to other graph transformation approaches. Finally, Section 1.7 highlights some of the advantages of this approach, summarizes, and concludes. We start with a brief overview of the DEVS formalism.

1.1.2 THE DISCRETE EVENT SYSTEM SPECIFICATION

This section introduces the DEVS formalism. In the rest of the paper, it will be shown how the modularity and expressiveness of DEVS allow for elegant encapsulation of model transformation (i.e., graph rewriting) building blocks.

The DEVS formalism was introduced in the late 1970s by Bernard Zeigler to develop a rigorous basis for the compositional modeling and simulation of discrete-event systems [2]. It has been successfully applied to the design, performance analysis, and implementation of a plethora of complex systems. Figure 1.1 shows the meta-model of a model transformation language based on DEVS. The dashed-line elements in Figure 1.1 (for now ignore the full-lined elements) show a simplified meta-model of DEVS in the UML Class Diagram notation. A DEVS model (the abstract class *Block*) is either an *AtomicBlock* or a *CoupledBlock*. An atomic model describes the behavior of a timed, reactive system. A coupled model is the composition of

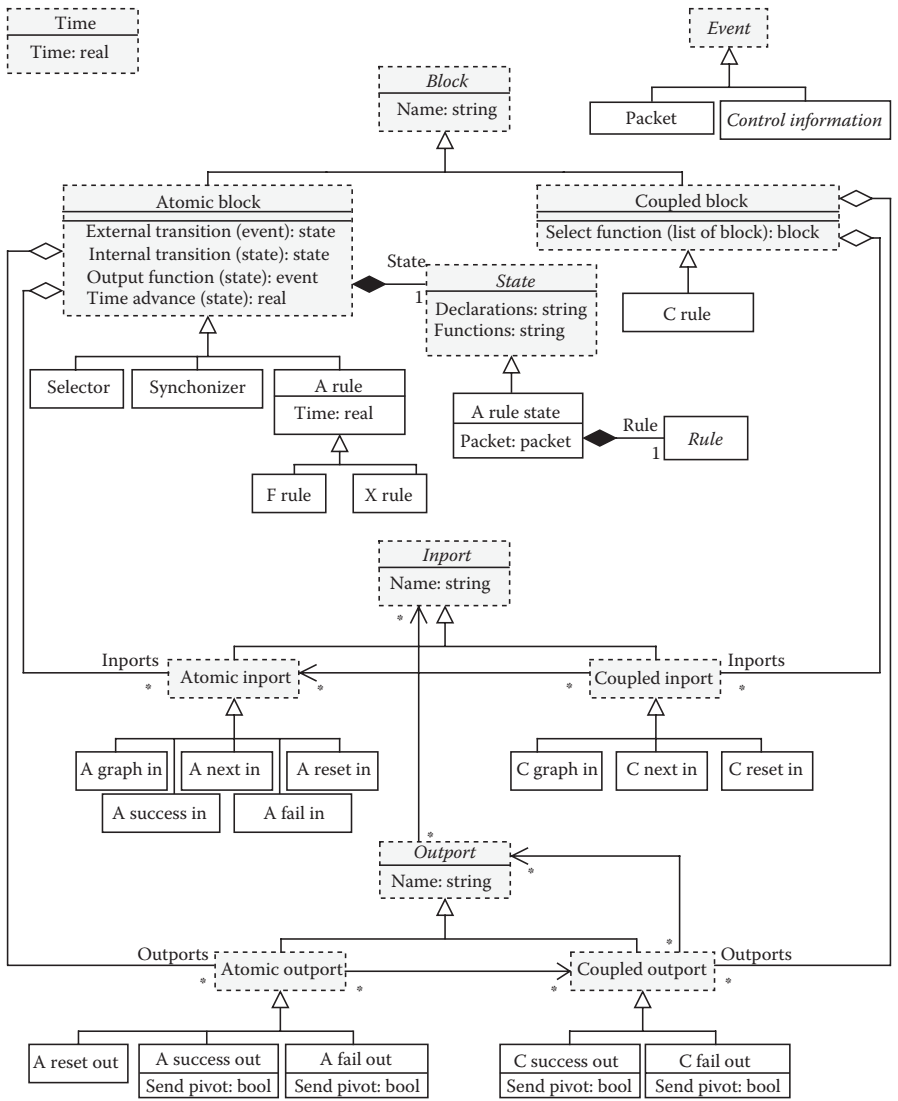


FIGURE 1.1 The MoTif meta-model, based on the DEVS meta-model.

several DEVS submodels that can be either atomic or coupled. Submodels have *ports* that are connected by channels (represented by the associations between the different ports). Ports are either *Input* or *Output*. The abstract classes (*In/Out*)*port* can be instantiated in an *Atomic(In/Out)port* or a *Coupled(In/Out)port*, respectively. Ports and channels allow a model to receive and send events (any subclass of event) from and to other models. A channel must go from an output port of some model to an input port of a different model, from an input port of a coupled model to an input port of one of its submodels, or from an output port of a submodel to an output port of its parent model, as depicted by the associations of Figure 1.1. Note that the dynamic semantics of DEVS cannot be expressed by the meta-model and will be informally outlined hereafter.

An *atomic* DEVS model is a structure $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$ where S is a set of sequential *states*, one of which is the *initial* state. X is a set of allowed *input events*. Y is a set of allowed *output events*. There are two types of transitions between states: $\delta_{int} : S \rightarrow S$ is the *internal transition function* and $\delta_{ext} : Q \times X \rightarrow S$ is the *external transition function*. Associated with each state are $\tau : S \rightarrow \mathbb{R}_0^+$, the *time-advance function*, and $\lambda : S \rightarrow Y$, the *output function*. In this definition, $Q = \{(s, e) | s \in S, 0 \leq e \leq \tau(s)\}$ is called the *total state space*. For each $(s, e) \in Q$, e is called the *elapsed time*. The $\mathbb{R}_0^+$ denotes the positive reals with zero included.

Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state. It will remain in any given state for as long as the time-advance of that state specifies or until input is received on an input port. If no input is received, after the time-advance of the state expires, the model first (before changing state) sends output as specified by the *outputFunction*, and then instantaneously jumps to a new state specified by the *internalTransition*. If, however, input is received before the time for the next internal transition, then the *externalTransition* is applied. The external transition depends on the current state, the time elapsed since the last transition, and the inputs from the input ports.

A *coupled* DEVS model named D is a structure $\langle X, Y, N, M, I, Z, select \rangle$ where X is a set of allowed *input events* and Y is a set of allowed *output events*. N is a set of *component names* (or labels) such that $D \notin N$. $M = \{M_n | n \in N, M_n \text{ is a DEVS model (atomic or coupled) with input set } X_n \text{ and output set } Y_n\}$ is a set of DEVS *submodels*. The $I = \{I_n | n \in N, I_n \subseteq N \cup \{D\}\}$ is a set of *influencer* sets for each component named n . I encodes the connection topology of submodels. $Z = \{Z_{i,n} | \forall n \in N, i \in I_n, Z_{i,n} : Y_i \rightarrow X_n \text{ or } Z_{D,n} : X \rightarrow X_n \text{ or } Z_{i,D} Y_i \rightarrow Y\}$ is a set of *transfer functions* from each component i to some component n . The $select : 2^N \rightarrow N$ is the *select* or tie-breaking function. The 2^N denotes the powerset of N (the set of all subsets of N).

The connection topology of submodels is expressed by the influencer set of each component. Note that for a given model n , this set includes not only the external models that provide inputs to n , but also its own internal submodels that produce its output (if n is a coupled model). Transfer functions represent output-to-input translations between components, and can be thought of as channels that make the appropriate type translations. For example, a “departure” event output of one submodel is translated into an “arrival” event on a connected submodel’s input. The *select* function takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the submodels. A priori, each submodel in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modeled by the output function of a sender, and the external transition function of a receiver. There is, however, a *serialization* whenever there are multiple submodels that have an internal transition scheduled to be performed at the same time. The modeler controls which of the conflicting submodels undergoes its transition first by means of the *select* function.

We have developed our own DEVS simulator called **pythonDEVS** [3], grafted onto the object-oriented scripting language Python.

1.2 THE ANT WORLD SIMULATION CASE STUDY

The case study used in this chapter is based on case no. 2 (the AntWorld Simulation case study) of the GraBaTs 2008 tool contest [1]. This is a benchmark for the comparison of graph transformation tools that stresses local rule application. A solution using a DEVS-based graph transformation language MoTif (Modular Timed graph transformation) [4] was presented at the 2008 tool contest.

The complete description of the behavior can be found online [1] and is as follows. The AntWorld simulation map is discretized into concentric circles of nodes (representing a large area) centered at a hill (the ant home). Ants are moving around searching for food. When an ant finds food, it brings it back to the ant hill in order to grow new ants. On its way home, the ant drops pheromones marking the path to the food reservoir. If an ant without food leaves the hill or if a searching ant hits a pheromone mark, it follows the pheromone path leading to the food. This behavior already results in the well-known ant trails.

The AntWorld simulation works in rounds (similar to time-slices). Within each round, each ant makes one move. If an ant is not in carry mode and is on a node with food parts, it takes one piece of food and enters carrying mode. Note that it may still move within the current round. On the other hand, if an ant carries some food, it follows the links toward the inner circle one node per round. During its way home (toward the unique hill at the center of all nodes), on each visited node (including the node that it picked food from) the ant drops 1024 parts of pheromones in order to guide other ants to the food place. However, if a carrying ant is on the hill, it drops the food and enters the search mode. It may leave the hill within the same round. Any ant without food is in search mode. While in this mode, the ant checks the neighboring node(s) of the next outer circle for pheromones. If some hold more than nine parts of pheromones, the ant chooses one of these nodes randomly. Otherwise, the ant moves to any of its neighbor nodes based on a fair random choice (but never enter the ant hill). Whenever during one round an ant is on a node on the outmost circle, a new circle of nodes shall be created. For each outmost grid node, a new grid node is created; but three nodes are created in the case of a main axis node. During the creation of this next circle, every 10th node shall carry 100 food parts. If a circle has, for example, 28 nodes, node 10 and node 20 of that circle shall have food. Thus, this circle would need just two more nodes to create a third food place. Therefore,

these eight nodes are kept in mind and during the creation of the next circle (in our example with 36 nodes) we add another food place when two more nodes have been added. Thus, across circles, every 10th node becomes a food place. After each round, all pheromones shall evaporate: reducing by 5%. Also, the hill shall consume the food brought to it by creating one new ant per delivered food part.

To emphasize the advantages of the use of DEVS for controlled graph rewriting, the AntWorld case study has been extended as follows. The ants running around seeking food are not protected from external factors. In fact, a human could step on an area discovered by the ants. Hence at random points in time, ants will be killed. This happens on a grid node chosen at random as well as its neighboring nodes. Irrespective of whether ants are present or not on these four nodes, the nodes will lose food parts by a factor of 2 and pheromones by a factor of 10. Section 1.4 will show how this can be modeled by simulating the user decision of when a human step will occur and Section 1.5 will show how this can be done in real time, allowing an actual user to intervene.

1.2.1 THE ANTWORLD LANGUAGE (ABSTRACT AND CONCRETE SYNTAX)

As shown in Figure 1.2, the AntWorld formalism consists of Ants and GridNodes. An *Ant* element can go on a *GridNode* that can also be a *Hill*. GridNodes can hold pheromones and food parts. Ants can be in “carry mode.” The grid nodes are connected in circles centered at the hill in a very specific way. This is why the meta-model differentiates between connections in the same circle and to the next circle for neighboring grid nodes. Furthermore, different strategies for the generation of the grid are used depending on whether it is a node along the two main axes. This is why we distinguish between main axis nodes and the other grid nodes. A node counter is also needed to decide which generated node will hold food parts. Using AToM³ [5] as a modeling environment enables one to associate a concrete syntax for each meta-model element. This is depicted in Figure 1.2 by the pictures attached to the elements by red-dashed lines.

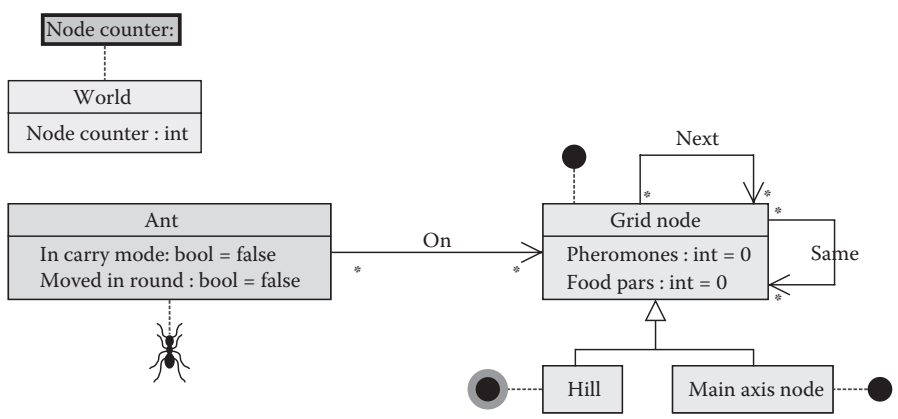


FIGURE 1.2 AntWorld meta-model.

1.2.2 THE ANTWorld SEMANTICS (GRAPH TRANSFORMATION)

As we have seen at the beginning of this section, the semantics of the AntWorld formalism is described in terms of simulation rules. In our case, these rules are graph rewriting rules taking as input a host graph (model) and producing as output the transformed graph. This encodes the state changes (i.e., dynamics) of the system. In MoTif, a rule consists of a LHS, a RHS, and optionally a NAC. The LHS represents a precondition pattern to be found in the host graph along with conditions on attributes. The RHS represents the postcondition after the rule has been applied on the matched subgraph by the LHS. The NAC represents what pattern condition in the host graph shall not be found, inhibiting the application of the rule if it is. Additionally, hints can be provided through pivot information. A rule using a pivot on a node of the LHS pattern binds the matching process to a previously specified matched node (this is useful for local search).

To illustrate how a graph rewriting rule is applied, we will consider the *ConnectNodesInSameCircle* rule from Figure 1.3. This rule states that whenever two neighboring nodes* N and M are on the same circle and are each linked to a *GridNode* (labeled, respectively, 1 and 2) on the next circle, a connection *toSameCircle* must be drawn between these latter nodes in the same direction N and M are connected. However, 1 may not be the source of a link to a node on the same circle nor 2 be the target of such a link.

Also, N must be bound to the pivot that this rule receives and M will be the pivot of the next rule. This internal dependency between rules is the essence of how local rule application is achieved in MoTif. For this particular case, the next rule that will be executed is again *ConnectNodesInSameCircle* as the control structure will show, in Section 1.3. This makes the connecting step between generated nodes ordered in a clockwise direction.

One of the advantages of using graph transformation for transforming models is that they can easily be described in a visual language, which makes the rules human readable. The remaining rules described in the appendix also use concrete syntax.

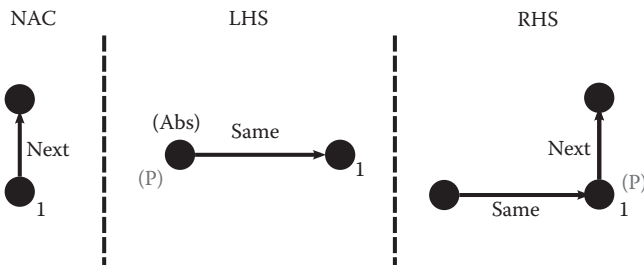


FIGURE 1.3 The *ConnectNodesInSameCircle* rule.

* We use the term nodes in the general sense (i.e., including *GridNode*, *MainAxisNode*, and *Hill*) because subtype matching can be used. This allows a pattern element to be matched to any element from the same class or from a subclass of it. In MoTif, such a pattern element is flagged with the {Abs} label, short for abstract.

1.3 A META-MODEL FOR DEVS-BASED GRAPH TRANSFORMATION

MoTif is a controlled graph transformation language. It offers a clean separation of the transformation entities (i.e., the rewriting rules) from the structure and flow of execution of the transformation. While Section 1.2.2 outlined the graph transformation rules, we focus here on the structural and control flow aspect of MoTif. Revisiting Figure 1.1 where up to now only the DEVS meta-model was mentioned, we will now see how a MoTif model is a DEVS model specialized for graph transformation.

The central elements of this DEVS-based graph transformation meta-model are the rule blocks. The graphs are encoded in the events that flow through the ports from block to block. The atomic block **ARule** (for “Atomic Rule”) is the smallest transformation entity and the coupled block **CRule** (for “Coupled Rule”) is meant for composition of rule blocks. A rewriting rule is part of the state of an *ARule* as a reference to the compiled rule. The rule application is performed in two phases: (1) the matching (where all the possible matches are found) and (2) the transformation on one or more matches. The *ARuleState* also keeps track of the graph and pivot received. The time advance of an *ARule* can be specified at modeling time to set its execution time (both match and transform). Otherwise the time advance is $+\infty$.

ARule blocks receive packets (a graph, with a potential pivot node) from the *AGraphIn* port. In case of success (i.e., when at least one match has been found), the packet containing the transformed graph is output through the *ASuccessOut* port. In case of failure, the original graph is sent through the *AFailOut* port. Furthermore, it is possible to enable pivot passing for these two outputs. For the success output, either the new pivot specified by the rewriting rule or the original received pivot is passed on to the next block. In the case of multiple matches found in the received graph, a host graph ever received by the *ANextIn* port will only apply the transformation on the next match without running the matching phase one more time. This feature is very useful as we will see in the AntWorld example for the flow logic and performance. On reception of an event through the *AResetIn* port, the rule application is cancelled and the state of the *ARule* is reset. Similar ports are available for a *CRule* block that serve as interface from its incident blocks to its submodels.

To increase the expressiveness of the language MoTif, additional rule blocks have been added. Among them is the *FRule* that will be used in our example. It is an *ARule* that applies its transformation phase to *all* the matches found (in arbitrary order) before sending the new graph. The matches are assumed to be parallel independent.

As in a general purpose DEVS model, atomic and coupled rule blocks are connected through their ports. There could be one-to-many or many-to-one connections between them. The semantics of an *(A/C)SuccessOut* output* connected to many *(A/C)GraphIn* inports is the parallel execution of the rules encoded in the receiving blocks. Since the classical DEVS is used here, the parallel execution of the external transition of these rule blocks is serialized as specified by the select function. In our case, one block is chosen at random, first among the matching rules and then among those that failed. Many-to-one connections between rule blocks ports

* The *(A/C)FailOut* has an analogous semantics.

are not encouraged, since different graphs will be received by a single *ARule* at the same time.

In graph grammars, it is sometimes wished to have many rules that match but let only one execute. That is why MoTif introduces the **Selector** block. Such a pattern can be found in Figure 1.6 involving, for example, a **Selector** and the **Generate** and **CreateFood ARules** connected to it. Its purpose is to receive, through its *ASuccessIn* inport, the transformed graph sent from an *ARule* that has been chosen by the **select** function. Instantaneously it outputs an event via its *AResetOut* outport, forcing all remaining rules to reset their state. Then, with a time advance of 0, the *Selector* passes the packet it received to the next block(s) via its *ASuccessOut* port. In the case of failure of all *ARules*, the rule selected by the **select** function sends its original packet to the *AFailIn* inport of the *Selector*. In return, the *Selector* forces the reset of all these rules and outputs the packet received through its *AFailOut* port. Since the semantics of DEVS is the compositional parallelization of atomic blocks, MoTif allows rules to conceptually be applied in parallel. This leads to what we call “threads” of rule applications, for example, the *HumanStep CRule* has four such threads in Figure 1.7c (which will be described later in the context of the case-study). Therefore a *Synchronizer* is needed to merge and synchronize the concurrent threads. Our approach uses *in-place* transformation of models, which means that the events sent and received are references to the host model, in contrast with *out-place* transformation where rules work on copies of the host model. This avoids the undecidable problem of merging transformed models. In that sense, the *Synchronizer* waits until all the threads have sent their packets through its *ASuccessIn* and/or *AFailIn* inports. Only then will it send the transformed graph through its *ASuccessOut* port if at least one thread has succeeded or else it will send the unmodified graph through its *AFailOut* port. To formalize these concepts, we define each of these elements in terms of a DEVS structure.

1.3.1 THE ARULE

The *ARule* is an *atomicDEVS*, parameterized by a rule $\mathbf{r}$ and by σ_1 to determine if a pivot is sent on a successful matching and σ_2 on failure.

$$ARule_{\mathbf{r}, \sigma_1, \sigma_2} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle,$$

where

$$T = \mathbb{R}_0^+$$

$$S = \{s = (\gamma, \rho, \in, \sigma_1, \sigma_2, \Sigma(\mathbf{r})) \mid \gamma \in G, \rho \in V_G \cup \{\phi\}, \in, \sigma_1, \sigma_2, \in \{\mathbf{true}, \mathbf{false}\}\}$$

$$\tau(s) = \begin{cases} t \in [0, \infty) & \text{if } \in = \mathbf{true} \\ \infty & \text{otherwise} \end{cases}, \forall s \in S$$

$$X = X_{A\text{GraphIn}} \times X_{A\text{ResetIn}} \times X_{A\text{NextIn}}$$

$$X_{A\text{GraphIn}} = \{\langle \gamma, \rho \rangle\} \cup \{\phi\}$$

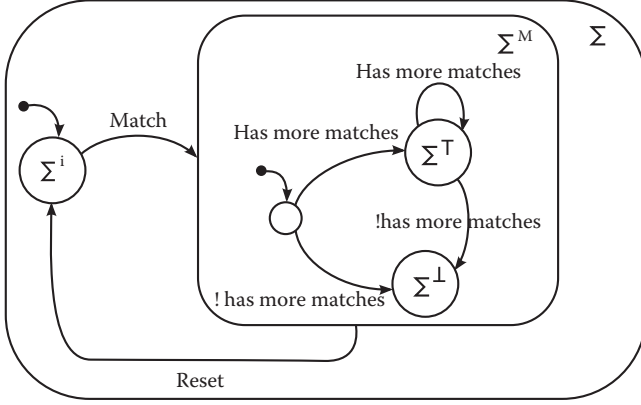


FIGURE 1.4 The state automaton of a rule r .

$$\begin{aligned}
 X_{\text{AResetIn}} &= \{\mathbf{false}\} \cup \{\phi\} \\
 X_{\text{ANextIn}} &= \{\langle \gamma, \rho \rangle_n\} \cup \{\phi\} \\
 Y &= Y_{\text{ASuccessOut}} \times Y_{\text{AFailOut}} \\
 Y_{\text{ASuccessOut}} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\
 Y_{\text{AFailOut}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
 \omega: T &\rightarrow X \\
 \delta_{\text{int}} &= (\gamma, \rho, \epsilon, \sigma_1, \sigma_2, \Sigma(r)) = (\gamma, \rho, \mathbf{false}, \sigma_1, \sigma_2, \Sigma(r))
 \end{aligned}$$

$$\delta_{\text{ext}}((s, e), x) = \begin{cases} (\gamma, \rho, \mathbf{true}, \sigma_1, \sigma_2, \Sigma^M(r)) & \text{if } x = \langle \gamma, \rho \rangle \\ (\gamma_n, \rho, \mathbf{true}, \sigma_1, \sigma_2, \Sigma^M(r)) & \text{if } x = \langle \gamma, \rho \rangle_n \\ (\gamma, \rho, \mathbf{false}, \sigma_1, \sigma_2, \Sigma^i(r)) & \text{if } x = \mathbf{false} \end{cases}$$

$$\lambda(s) = \begin{cases} \langle \gamma', \rho' \rangle & \text{if } \sigma_1 = \mathbf{true} \wedge \Sigma(r). \text{INSTATE}(\Sigma^T(r)) \\ \langle \gamma', \phi \rangle & \text{if } \sigma_1 = \mathbf{false} \wedge \Sigma(r). \text{INSTATE}(\Sigma^T(r)) \\ \langle \gamma', \rho' \rangle & \text{if } \sigma_2 = \mathbf{true} \wedge \Sigma(r). \text{INSTATE}(\Sigma^\perp(r)) \\ \langle \gamma', \phi \rangle & \text{if } \sigma_2 = \mathbf{false} \wedge \Sigma(r). \text{INSTATE}(\Sigma^\perp(r)). \end{cases}$$

In this notation, γ is a graph from the set of all possible graphs G . ρ is a node of this graph representing the pivot. $\langle \gamma', \rho' \rangle$ is the resulting packet after the transformation phase of the application of r . Note how it is possible to have $\rho' = \rho$. The ϵ is used to determine if the *ARule* is active. The function Σ gives the state of the rule r . It is illustrated by the automaton in Figure 1.4. The Σ^M gives the state of r after the matching phase, Σ^T gives the state of r when it has at least one unprocessed match left, and $\Sigma^\perp$ gives the state of r when there are none.

1.3.2 THE CRULE

The *CRule* is defined exactly like a *coupledDEVS*

$$CRule = \langle X, Y, N, \{M_i | i \in N\}, \{I_i\}, \{Z_{i,j}\}, select \rangle,$$

where

$$\begin{aligned} X &= X_{CGraphIn} \times X_{CResetIn} \times X_{CNextIn} \\ X_{CGraphIn} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\ X_{CResetIn} &= \{\mathbf{false}\} \cup \{\phi\} \\ X_{CNextIn} &= \{\langle \gamma, \rho \rangle_n\} \cup \{\phi\} \\ Y &= Y_{CSuccessOut} \times Y_{CFailOut} \\ Y_{CSuccessOut} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\ Y_{CFailOut} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \end{aligned}$$

The *select* function is described by these prioritized algorithmic steps:

1. If the *Selector* is in the *imminent list*, choose the *Selector*.
2. Among all the rules that still have a match, choose a corresponding *ARule* from the *imminent list* at random, no matter what depth it is at inside this *CRule*.
3. At this point no rule has any unprocessed match left, then choose any of the *ARule* in the *imminent list*.
4. Finally, the *imminent list* contains either custom atomic blocks or *Synchronizers*. Proceed with a first-in-first-out selection.

1.3.3 THE SELECTOR

The *Selector* is also an atomic *DEVS*

$$Selector = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle,$$

where

$$\begin{aligned} T &= \mathbb{R}_0^+ \\ S &= \{s = (t, f, \langle \gamma, \rho \rangle) | t, f \in \{\mathbf{true}, \mathbf{false}\}, \gamma \in G \cup \{\phi\}, \rho \in V_G \cup \{\phi\}\} \\ \tau(s) &= \begin{cases} 0 & \text{if } t = \mathbf{true} \vee f = \mathbf{true} \\ \infty & \text{otherwise} \end{cases}, \forall s \in S \\ X &= X_{ASuccessIn} \times X_{AFailIn} \\ X_{ASuccessIn} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\ X_{AFailIn} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\ Y &= Y_{ASuccessOut} \times Y_{AFailOut} \times Y_{AResetOut} \\ Y_{ASuccessOut} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\ Y_{AFailOut} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\ Y_{AResetOut} &= \{\mathbf{false}\} \cup \{\phi\} \end{aligned}$$

$$\begin{aligned}
\omega: T &\rightarrow X \\
\delta_{int}(s) &= (\mathbf{false}, \mathbf{false}, \langle \phi, \phi \rangle) \\
\delta_{ext}((s, e), x) &= \begin{cases} (\mathbf{true}, \mathbf{false}, \langle \gamma', \rho' \rangle) & \text{if } x = \langle \gamma', \rho' \rangle \\ (\mathbf{false}, \mathbf{true}, \langle \gamma, \rho \rangle) & \text{if } x = \langle \gamma, \rho \rangle \end{cases} \\
\lambda(s) &= \begin{cases} \{\langle \gamma', \rho' \rangle\} \cup \{\mathbf{false}\} & \text{if } s = (\mathbf{true}, \mathbf{false}) \\ \{\langle \gamma, \rho \rangle\} \cup \{\mathbf{false}\} & \text{if } s = (\mathbf{false}, \mathbf{true}). \end{cases}
\end{aligned}$$

1.3.4 THE SYNCHRONIZER

The *Synchronizer* is also an *atomicDEVS*, parametrized by the number of threads θ to synchronize.

$$Synchronizer_{\theta} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle,$$

where

$$\begin{aligned}
T &= \mathbb{R}_0^+ \\
S &= \{s = (t, f, \theta, \langle \gamma, \rho \rangle) \mid t, f, \theta, \in \mathbb{N}, \gamma \in G \cup \{\phi\}, \rho \in V_G \cup \{\phi\}\} \\
\tau(s) &= \begin{cases} 0 & \text{if } t + f = \theta \\ \infty & \text{otherwise} \end{cases}, \forall s \in S \\
X &= X_{ASuccessIn} \times X_{AFailIn} \\
X_{ASuccessIn} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\
X_{AFailIn} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
Y &= Y_{ASuccessOut} \times Y_{AFailOut} \times Y_{AResetOut} \\
Y_{ASuccessOut} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\
Y_{AFailOut} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
\omega: T &\rightarrow X \\
\delta_{int}(s) &= (0, 0, \theta, \langle \phi, \phi \rangle) \\
\delta_{ext}((s, e), x) &= \begin{cases} (t+1, f, \theta, \langle \gamma', \rho' \rangle) & \text{if } x = \langle \gamma', \rho' \rangle \\ (t, f+1, \theta, \langle \gamma, \rho \rangle_{-1}) & \text{if } x = \langle \gamma, \rho \rangle \end{cases} \\
\lambda(s) &= \begin{cases} \{\langle \gamma', \rho' \rangle\} & \text{if } t \geq 1 \\ \{\langle \gamma, \rho \rangle\} & \text{otherwise.} \end{cases}
\end{aligned}$$

On top of these constructs, pure atomic and coupled DEVS models are also allowed to be present in MoTif models. This allows the modeler to add customized behavior to the transformation model. De facto, the MoTif code generator makes use of it, when compiling the model down to an executable model transformation environment, by modeling the user of the transformation as well as an interface between the rule model and the user.

1.4 USING MOTIF FOR THE ANTWORLD SIMULATOR CASE STUDY

Having described the DEVS-based transformation language, we will now explore how MoTif models can be used. MoTif is a meta-modeled language and is provided with graphical visual concrete syntax. Figure 1.5 shows the overall structure of the DEVS model for the AntWorld graph transformation. Each block is shown with its ports along with the connections. The execution of the transformation is triggered by some user control. User intervention (such as a possible interrupt of a running simulation) is modeled in the *User* block, since the DEVS formalism allows one to specify preemptive external interrupts through the external transition function. The *Controller* block acts as the interface of the transformation system to the user: it receives user inputs and informs the user of the status of the execution. It also models the management of the transformation steps. The two rule blocks, *Round* and *HumanStep*, can both receive the host graph from the *Controller* and return a graph, transformed or not. This approach was also used to model a PacMan game [6].

Figures 1.6 and 1.7 show the core of the transformation model. The top left triangle on each rule block represents the *GraphIn* port. On the top right, the triangle with a line through it is the *ResetIn* port. The two small filled triangles on the left represent the *NextIn* port. At the bottom left, the double-lined triangle is the *SuccessOut* port and at the bottom right, the filled double-lined port is *FailOut*. Pivot passing is enabled when there is a round at the summit of one of the triangular outputs. For the

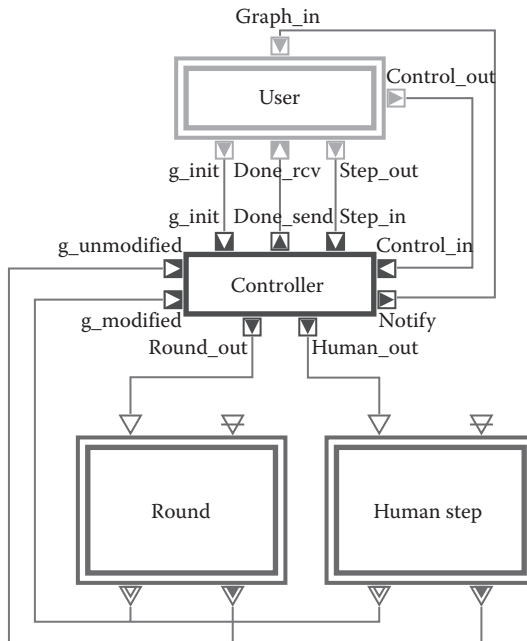


FIGURE 1.5 The overall transformation model in MoTif notation.

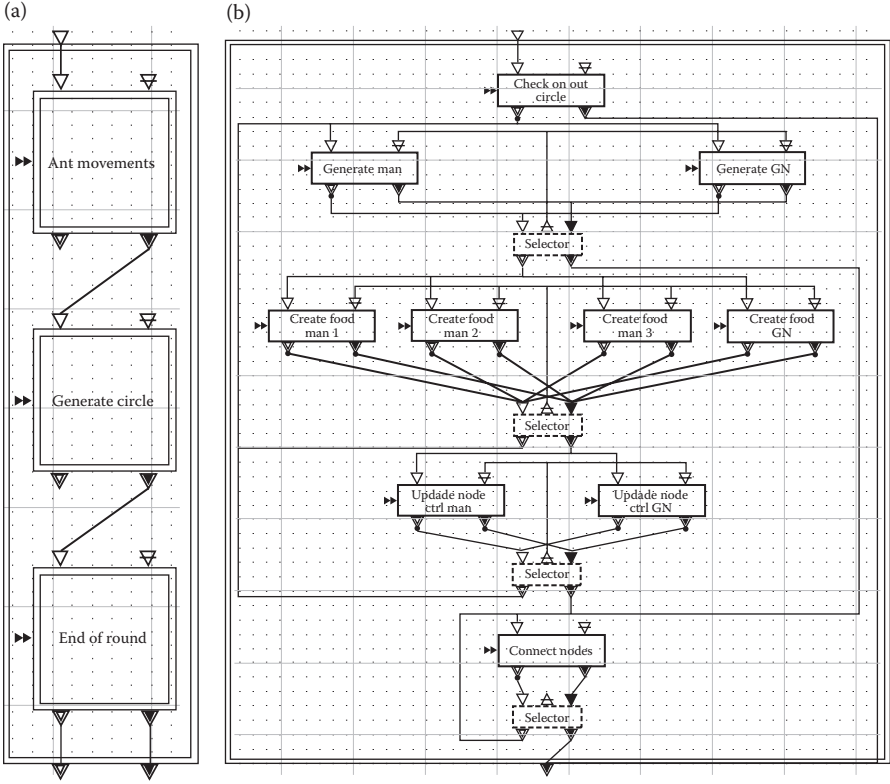


FIGURE 1.6 Submodels of the transformation model: the *Round CRule* in (a) and the *GenerateCircle* in (b) *CRule*.

Selector and the *Synchronizer*, the thick-lined triangle is the *SuccessIn* port and the filled triangle is the *FailIn* port. On the top of the *Selector*, the triangle with a line through it pointing up is the *ResetOut* port.

For completeness, Table 1.1 lists all the rules with a brief description used for a solution to the extended AntWorld case study using MoTif.

1.4.1 THE ROUND BLOCK

The AntWorld simulation is run in rounds. Figure 1.6a illustrates how a round is layered to first run the *AntMovement* subtransformation (Figure 1.7a), then when no more rule in this *CRule* is applicable, run the *GenerateCircle* (Figure 1.6b) subtransformation, and finally run the *EndOfRound* (Figure 1.7b) subtransformation.

At the level of the *AntMovement CRule*, the first rule looks for an ant to grab a food part. Having already explained in detail the description of the *ConnectNodes* rule, the reader is referred to the appendix for a description of all the rules. The name of every *ARule* block matches the name of the rule itself. If the *GrabFood* rule succeeds, this same ant moves one step toward the hill. Then the graph is sent back to the *GrabFood ARule* but via the *ANextIn* port to only choose the next matching and

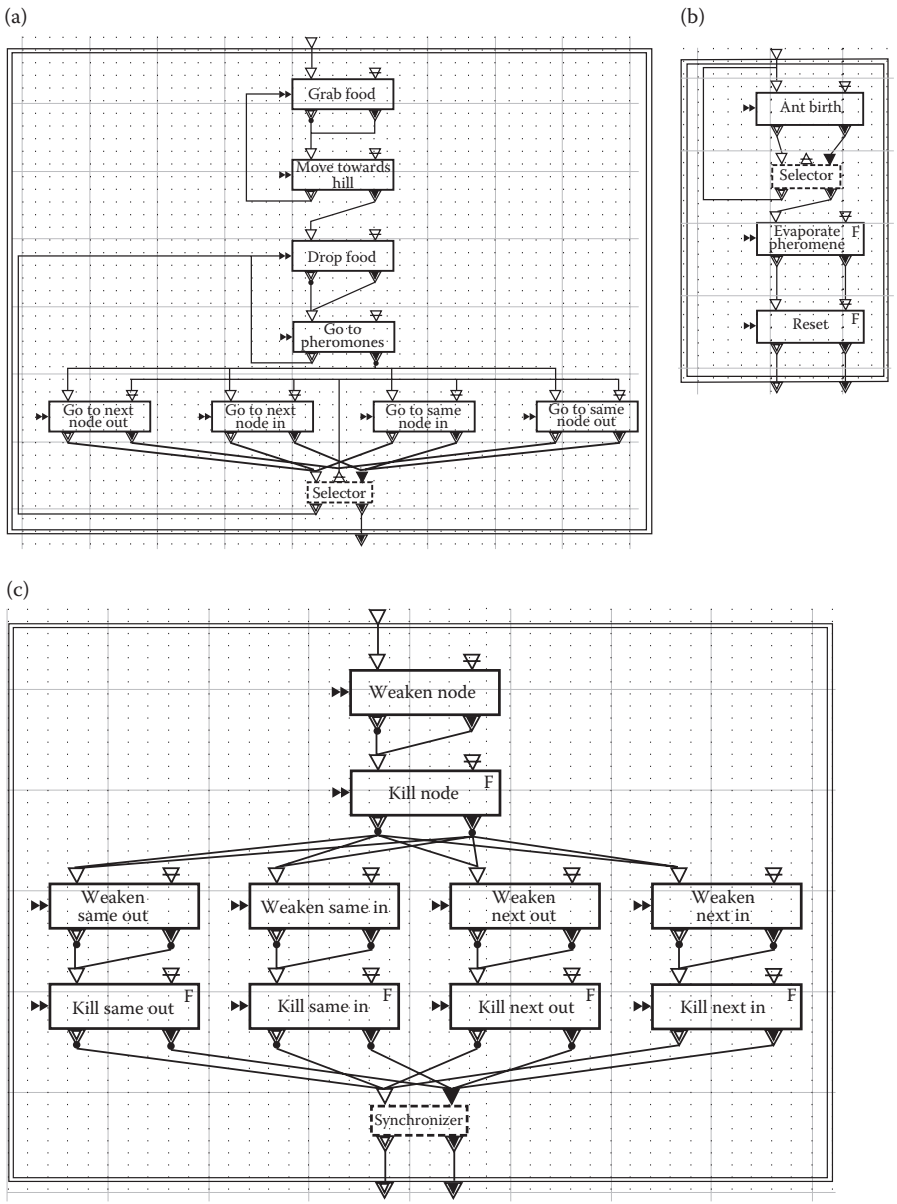


FIGURE 1.7 Submodels of the transformation model: the *AntMovements CRule* in (a), the *EndOfRound* in (b) *CRule*, and the *HumanStep* in (c) *CRule*.

apply the transformation. This loop continues till no more ants can move toward the hill. Afterward, an ant found in carry mode on the hill drops its food part and goes into search mode. When in search mode, the ant first tries to follow a pheromone trail or else moves randomly in any possible direction, either on a neighboring node on the same circle or on a neighboring node in the previous or following circle. This

TABLE 1.1**The Different Rules for the Extended AntWorld Case Study**

	Rule	Description
1	<i>GrabFood</i>	When an ant is on a node with food parts, remove a food part, add some pheromones and the ant becomes in carry mode.
2	<i>MoveTowardsHill</i>	When an ant is in carry mode and has not moved yet, make it move to the neighboring node on the previous circle.
3	<i>DropFood</i>	When an ant is on the hill and is in carry mode, increase the food parts of the hill by one and put the ant in search mode and allow it to move.
4	<i>GoToPheromones</i>	If the bound ant is in search mode and has not moved yet and the neighboring node on the next circle has more than nine pheromones, the ant moves to that node.
5	<i>GoToNextNodeOut</i>	If the bound ant is in search mode and has not moved yet, it moves to the neighboring node on the next circle.
6	<i>GoToNextNodeIn</i>	If the bound ant is in search mode and has not moved yet, it moves to the neighboring node on the previous circle, if it is not a hill.
7	<i>GoToSameNodeOut</i>	If the bound ant is in search mode and has not moved yet, it moves to the neighboring node on the same circle, in the direction of the link between the two nodes.
8	<i>GoToSameNodeIn</i>	If the bound ant is in search mode and has not moved yet on a node, it moves to the neighboring node on the same circle, in the opposite direction of the link between the two nodes.
9	<i>CheckOnOutCircle</i>	When an ant is on a node on the outmost circle, bind the node.
10	<i>GenerateMAN</i>	If the bound node is linked to a main axis node in the same circle and the latter node has no neighbor in the next circle, create three neighbors on the next circle linked to that node: the central new node being another main axis node. Bind the former main axis node.
11	<i>GenerateGN</i>	If the bound node is linked to a grid node in the same circle and the latter node has no neighbor in the next circle, create a neighbor on the next circle linked to that node. Bind the former grid node.
12	<i>CreateFoodMAN1</i>	If a grid node is linked to a main axis node that is the neighbor on the next circle of the bound main axis node and the node counter is nine, then that grid node is made to hold 100 food parts and the node counter is increased by three (modulo 10).
13	<i>CreateFoodMAN2</i>	If a main axis node is the neighbor on the next circle of the bound main axis node and the node counter is eight, then that main axis node is made to hold 100 food parts and the node counter is increased by three (modulo 10).
14	<i>CreateFoodMAN3</i>	If a grid node is linked to a main axis node that is the neighbor on the next circle of the bound main axis node and the node counter is seven, then that grid node holds 100 food parts and the node counter is increased by three (modulo 10).
15	<i>CreateFoodGN</i>	If a grid node is the neighbor on the next circle of the bound grid node and the node counter is nine, then that grid node holds 100 food parts and the node counter is increased by one (modulo 10).

TABLE 1.1 (CONTINUED)

The Different Rules for the Extended AntWorld Case Study

	Rule	Description
16	<i>UpdateNodeCtrMAN</i>	If a bound main axis node is found, then increase the node counter by three (modulo 10).
17	<i>UpdateNodeCtrGN</i>	If a grid axis node is found, then increase the node counter by one (modulo 10).
18	<i>ConnectNodes</i>	cf. Section 1.2.2.
19	<i>AntBirth</i>	When the hill has some food part, remove one food part and create an ant in search mode that has not moved yet.
20	<i>EvaporatePheromones</i>	When a node has some pheromones multiply the number of pheromones by 0.95, rounding to the next integer value.
21	<i>Reset</i>	When an ant has already moved, change it to not moved.
22	<i>WeakenNode</i>	When a node is found, divide its food parts by 2 and its pheromones by 10. Bind this node.
23	<i>KillOnNode</i>	If an ant is found on the bound node, delete the ant and keep the binding.
24	<i>WeakenSameOut</i>	When a node is linked to the bound node in the same circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
25	<i>KillSameOut</i>	If an ant is found on a node linked to the bound node in the same circle, delete the ant and keep the binding.
26	<i>WeakenSameIn</i>	When a node is linked to the bound node in the same circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
27	<i>KillSameIn</i>	If an ant is found on a node linked to the bound node in the same circle, delete the ant and keep the binding.
28	<i>WeakenNextOut</i>	When a node is linked to the bound node in the next circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
29	<i>KillNextOut</i>	If an ant is found on a node linked to the bound node in the next circle, delete the ant and keep the binding.
30	<i>WeakenNextIn</i>	When a node is linked to the bound node in the previous circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
31	<i>KillNextIn</i>	If an ant is found on a node linked to the bound node in the previous circle, delete the ant and keep the binding.

search mode behavior is achieved by the four *ARules* connected to the *Selector*, which finds their corresponding matches in parallel and only one is chosen randomly to apply its transformation. The scenario is repeated for every ant matched by the *DropFood* rule. When no ant is provided by *DropFood*, all (iteratively chosen in random fashion) ants in search mode that have not moved yet, will move to a neighboring node.

The (possibly) new graph is passed onto the *GenerateCircle CRule* when there are no more ants left to be moved. A check looking for an outermost circle node reached by an ant is first verified. If no node fulfils this criterion, the graph is passed onto the *EndOfRound CRule*. Otherwise, the generation of nodes of the next circle

in clockwise order is engaged. The creation of nodes starts at the node found by the *CheckOnOutCircle* rule. The transformation model takes care of the case where the creation happens at the level of a main axis node (three nodes are created, the middle one also being a main axis node) or a default grid node (only a default grid node is created). The four *CreateFood ARules* handle the specification that some food is placed on every 10th new node, the count of nodes being tracked by the *World* element. Once all nodes on the new circle are created, they get connected through *ConnectNodes*.

Finally, the end of the round is reached. For each food part on the hill one ant element is created; this is depicted by the *AntBirth*, *Selector* pair that makes the rule execute “for as long as possible.” Note how *EvaporatePheromones* and *Reset* are all *FRules*, denoted by the “F” in the *ARule* box. This forces the rule to transform all its matches before outputting its packet. Note that this is safe, since no two matchings can be critical pairs. Subsequently all pheromones are evaporated and a final clean up is made. The order of these three *ARules* is arbitrary and could, in principle, be executed in parallel.

1.4.2 THE HUMANSTEP BLOCK

The *HumanStep* semantics can be summarized by two actions: weakening and killing. Weakening a node reduces its food parts and pheromones, if any. Killing removes all ants on a node. When the *CRule* receives a graph, first an arbitrary node is chosen and the two actions are applied to it. *WeakenNode* sends *KillOnNode* the node it has chosen along with the transformed graph to apply the killing rule on this same node. Since all ants on the node are deleted, *KillOnNode* is an *FRule*. The same logic is repeated for each of the four neighboring nodes. In this case, every kill and weaken *ARule* pair on neighbors can also be executed in parallel since the rules are parallel independent. As stated by the Local Church-Russer theorem [7], two rules are said to be parallel independent if the matching elements that overlap between the two LHS are preserved by each of the rules application. Proving the parallel independence of eight rules* is not the focus of this chapter and is therefore not discussed any further.

1.4.3 THE CONTROLLER BLOCK

The *Controller* atomic DEVS encodes the coordination logic between the external input and the transformation model. It is the control that receives the graph to transform and the number of steps to be applied. It also notifies the user about termination. The *Controller* sends the graph to the *Round* subtransformation model and waits for a graph in return. The returned graph may or may not be modified. However this cycle is interrupted when it receives an event from its *control_in* port. It will send the graph to the *HumanStep* subtransformation model after *Round* has returned a graph. Every time a graph is received back, the *Controller* will notify the user by sending

* The four pairs *WeakenSameOut*, *KillSameOut*, *WeakenSameIn*, *KillSameIn*, *WeakenNextOut*, *KillNextOut*, and *WeakenNextIn*, *KillNextIn* are parallel independent.

it the graph it just received; which happens to be after each round. This is repeated depending on the “steps” requests received. Note that the system could, in principle, receive multiple graphs at any one time (thanks to the data flow nature of DEVS), but we restricted it to a control flow in our case. Also, the user could request more “steps” even when there are some steps left in the running transformation.

1.4.4 THE USER BLOCK

User is a coupled DEVS block that sends graphs and “steps” control signals and receives termination events. The graphs are Abstract Syntax Graphs (AToM³’s internal representation of models) of models in the AntWorld language. Steps represent the number of iteration cycles the user requests the simulator to perform in a row. The 0 ends the simulation. The ∞ runs the simulation in continuous mode, executing till termination (or until interrupted by an external signal). For our case study we let the simulation run “as long as possible.” The reception of a termination event means that either the requested number of steps have been performed or that the execution has reached its end. In the latter case, no more transformations can be applied to the graph. The inports and outports of the *User* block are connected to the *Controller* block only. The *User* block is composed of two atomic submodels: *UserInput* and *UserBehavior*. The *User* is separated into two submodels to distinguish the decision making of performing the *HumanStep* transformation from the interaction with the transformation system. Hence the *UserBehavior*’s time advance is randomized to emulate the random time aspect of the human stepping on a node. This separation of the user is the key for extending standard graph transformations to event-driven timed graph transformations. Note how this event-based selection of rules is different from “Event-driven Graph Rewriting,” as introduced in [8]. The authors [8] let the rule itself determine how to behave given an event. Hence it is the rule that “parses” the event. In our approach, the event is “parsed” by a separate atomic DEVS block and the appropriate rule is applied accordingly. This approach is therefore more modular.

1.4.5 SIMULATION RESULTS

For the simulation experiments, an initial model was used with the following setup: eight nodes, one hill, and one node counter and no ants. Figure 1.8 shows a snapshot of the model being transformed. Some performance measurements have been collected at the end of each round. Table 1.2 shows some results per round, the number of circles present on the grid, the total number of nodes, the number of food parts present on the grid, and the total number of ants alive. Furthermore, for each round we show how long the transformation step took in seconds. So the 165th round took 7 minutes and 18 seconds while the generation of the 13th circle (100 nodes) took about 48 minutes. The total execution time was 23,890 seconds. Also, on average over the first 165 rounds the transformation time is about 144 seconds and without considering the time consumed by the *GenerateCycle* block, the average is 104 seconds. These measurements were taken on a Windows Vista machine with an Intel Core 2 Duo CPU with 1.5 GHz of RAM.

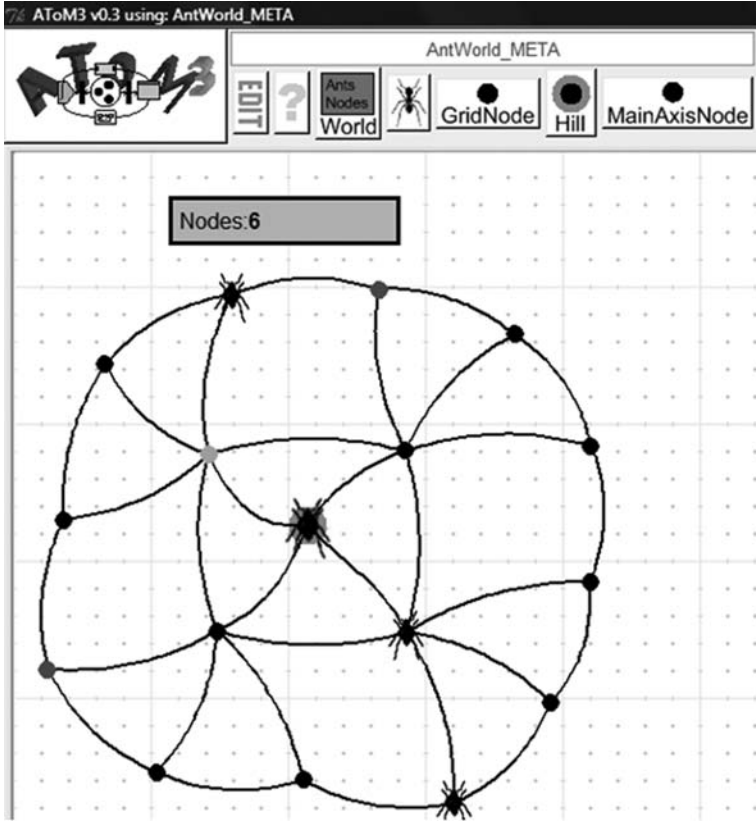


FIGURE 1.8 Snapshot of the model in ATOM³ while being transformed.

1.5 TIMED GRAPH TRANSFORMATION

In this section we briefly discuss the introduction of time in transformation languages and how our approach makes use of it.

1.5.1 INTRODUCTION OF TIME

DEVS is inherently a timed formalism. Hence, using DEVS as a semantic domain for graph transformation has a side effect of extending graph transformation with the notion of time.

Timed Graph Transformation, as proposed by Gyapay et al. [9], integrates time in only one particular theoretical foundation of graph transformation: the double push-out approach [7]. They extend the definition of a graph transformation rule by introducing, in the model and rules, a “chronos” element that stores the notion of time. Rules can monotonically increase time.

In our approach, time is modeled and embedded both at the block entity level. In contrast with Gyapay [9], it is the execution of a rule that can increase time and not

TABLE 1.2
Performance Measurements

Round #	Circles	Nodes	Food	Ants	Time (sec)	Round #	Circles	Nodes	Food	Ants	Time (sec)
1	2	16	0	8	0	43	7	196	1724	34	17
8	4	64	500	8	0	44	7	196	1713	35	7
9	5	100	900	8	15	55	7	196	1651	46	10
10	5	100	900	8	1	56	8	256	2237	47	207
11	5	100	900	8	1	65	9	324	2891	56	48
12	5	100	900	8	1	73	10	400	3640	64	698
13	5	100	899	8	0	103	10	400	3423	94	74
14	6	144	1297	8	39	104	11	484	4214	95	1218
15	6	144	1297	8	0	105	12	576	5107	96	1924
32	6	144	1267	23	1	106	12	576	5099	97	117
33	6	144	1264	24	2	107	13	676	6093	98	2881
38	6	144	1243	29	3	165	13	676	5523	156	438

the rule itself. This is done through the time advance of *ARules*. Hence, the control flow of the transformation has full access to it. As pointed out by Gyapay [9], time can be used as a metric to express how many time units are consumed to execute a rule. Having time at the level of the block containing a rule rather than in the rule itself does not lose this expressiveness.

Also, providing time to the control flow structure can enhance the semantics of the transformation. The AToM³, for example, provides control over execution time delay for animation. To have more realistic simulations in the AntWorld example, we can give meaning to the time delay between the executions of different rules by modeling the user. For example, the ant movement rules may take more time than the generation of new circles and the rules at the end of the round may take no time.

1.5.2 REAL-TIME MODEL TRANSFORMATION AND DEPLOYMENT

Having control over time for simulation purposes can be extended to real-time applications. In Syriani and Vangheluwe [10], we have shown how using DEVS for programmed graph rewriting with time allows for simulation-based design. This was achieved on a game example where first the *UserBehavior* block was enhanced with some artificial intelligence for path finding to optimize parameters such as the speed of the game. Then a web-based game was synthesized from the simulation model, where the *UserBehavior* block was replaced by an actual human user. A real-time simulator, our Python implementation of RT-DEVS [11], was used.

A similar approach has been employed for the extended AntWorld case study. From the meta-model of the AntWorld formalism designed in AToM³, an Ajax/SVG-based Web application was (mostly) synthesized (yet another model transformation). The web page consists of a visual representation of a given model. While the transformation runs, ants move and new node circles are created. Each node is equipped with an event handler listening to a mouse click from the user. This allows the user to “interrupt” the *Round* subtransformation and trigger the *HumanStep* subtransformation, passing the clicked node as initial pivot.

1.6 RELATED GRAPH TRANSFORMATION TOOLS

Many graph transformation tools and languages have been developed during the past decade. Hence, we present those that describe a transformation in a controlled way (i.e., programmed graph rewriting). The Graph Rewriting And Transformation (**GReAT**) tool [12–14] treats the source model, the target model, and the temporary objects created during the transformation as a single graph using a unified meta-model. Rules consist of a pattern graph described using UML Class Diagram notation where the elements can be marked to match a pattern (**Bind role**), to remove elements (**Delete role**), or to create elements (**CreateNew role**). A guard is associated with each production; this is an Object Constraint Language (OCL) expression that operates on vertex and edge attributes. An attribute mapping can also be defined to generate values of vertex and edge attributes with arithmetic and string expressions. GReAT’s control flow language uses a control flow diagram notation where a production is represented by a block. Sequencing is enabled by the use

of input and output interfaces (**Inports** and **Outports**) of a block. Packets (the graph model) are fed to productions via these ports. The **Inport** also provides an optimization in the sense that it specifies an initial binding for the start of the pattern matcher. Two types of hierarchical rules are supported. A **block** pushes all its incoming packets to the first internal rule, whereas a **forblock** pushes one packet through all its internal rules. Branching is achieved using test case rules, consisting of a left-hand side (LHS) and a guard only. If a match is found, the packet will be sent to the output interface. Parallel execution is possible when the **Outports** of a production are connected to different **Inports**. There is no notion of time.

In the Visual Modeling and Transformation System (**VMTS**) [15,16], the LHS and RHS of a graph transformation rule are represented as two separate graphs. They can be linked (internal causality) by Extensible Stylesheet Language scripts. These scripts allow attribute operations and represent the **create** and **modify** operation of the transformation step. Also, parameters and pivot nodes can be passed to a step for optimization. The programmed graph rewriting system of VMTS is the VMTS Control Flow Language (VCFL), a stereotyped Activity Diagram [17]. This abstract state machine handles pre- and postconditions of rules. Sequencing is achieved by linking transformation steps; loops are allowed. Branching in VCFL is conditioned by an OCL expression. In the case of multiple branching (a step connected to more than one step), only the first successfully evaluated branch will apply its transformation step. Iteration is controlled by loops in a sequence of steps. A branch can also be added to provide conditional loops. Hierarchical steps are composed of a sequence of primitive steps. A primitive step ends with success if the terminating state is reached and ends with failure when a match fails. However, in hierarchical steps, when a decision cannot be found at the level of primitive steps, the control flow is sent to the parent state or else the transformation fails. Parallelism is not yet implemented in VCFL. VMTS is language-oriented toward the .NET framework. There is no notion of time.

The PROgrammed Graph Rewriting System (**PROGRES**) [18] was the first fully implemented environment to allow programming through graph transformations. It has very advanced features not found in other tools such as backtracking. Insights gained through the development of PROGRES have led to **FUJABA** (From UML to Java and Back Again) [19,20], a completely redesigned graph transformation environment based on Java and UML. FUJABA's programmed graph rewriting system is based on Story Charts and of Story Diagrams [20]. An activity in such a diagram contains either graph rewrite rules, which adopt a Collaboration Diagram-like representation [17], or pure Java code. The graph schemes for graph rewriting rules exploit UML class diagrams. With the expressiveness of Story Charts, graph transformation rules can be sequenced (using success and failure guards on the linking edges) along with activities containing code. Branching is ensured by the condition blocks that act like an if-else construct. An activity can be a for-all story pattern, which acts like a for-loop on a transformation rule, iterating over its different matches. FUJABA's approach is implementation-oriented. Classes define method signatures and method content is described by Story Chart diagrams. All models are compiled to Java code. There is no notion of time.

The **MOFLON** [21] toolset uses the FUJABA engine for graph transformation, since the latter already features UML-like graph schemata. It provides an environment

where transformations are defined by Triple Graph Grammars (TGGs) [22]. These TGGs are subsequently compiled to Story Diagrams. This adds declarative power to FUJABA similar to that of the Object Management Group's (OMG) QVT (Query/View/Transformation*).

Although all these tools provide a control flow mechanism for graph transformations, many designed a new formalism for this purpose. Also, none of these exploit event-based transformations; MoTif not only allows that but the user and its interaction with the executing transformation can even be explicitly modeled, offering a user-centric approach to model transformations. Note that in the abovementioned tools, user-tool interaction is hard-coded. Furthermore, the notion of time is absent in these languages. Some do provide sophisticated, user friendly graphical interfaces while others are more efficient.

1.7 CONCLUSIONS

In this chapter, we have introduced the DEVS formalism as an appropriate semantic domain for “programmed” model transformation. As DEVS is a timed, highly modular, hierarchical formalism for the description of reactive systems, control structures such as sequence, choice, and iteration are easily modeled. Nondeterminism and parallel composition also follow from DEVS' semantics. Each rule of a model transformation is encoded in an atomic-DEVS block (this is comparable to the atomicity of the rules in transformation tools such as PROGRES). The encoding is done automatically, by compiling declarative transformation rules into appropriate atomic-DEVS functions. Model transformation building blocks send and receive events through their output and input ports, respectively. In those events, to-be-transformed graphs as well as optimization hints (such as pivot nodes in the tools GReAT and VMTS) are encapsulated.

Other events, related to information on the order in which rules are executed, are also fed to the channels (such as the event for resetting an *Arule*, for example). The DEVS formalism is compositional: the behavior of a DEVS block is independent of the context it is used in. This allows for modular reuse of building blocks and is one of the main reasons for choosing DEVS as a semantic domain for model transformation.

The use of DEVS allows for multilevel hierarchical modeling. Sequencing is treated as in GReAT by simply connecting block ports. Iteration and loops can thus be modeled. A given block can be a test block for branching if we give it such a semantics (i.e., no transformation occurs). Parallel execution is provided by the DEVS formalism when an output port is connected to multiple input ports. If true parallelism is needed, the parallel DEVS [23] formalism can be used.

A side effect of the use of DEVS as a semantic domain is the explicit introduction of the notion of time. This allows one to model a time advance for every rule as well as to interrupt (preempt) rule execution.

The proposed approach was illustrated through the modeling of an extended version of the AntWorld model transformation benchmark. We showed how the use of DEVS ultimately allows for real-time simulation and execution.

* <http://www.omg.org>

Performance-wise, the generated code for individual transformation rule needs to be more efficient. Recent results from other transformation tools, such as VIATRA [24], indicate how higher performance may be achieved. Increasing the expressiveness of the rule pattern specification language is also ongoing work.

REFERENCES

1. [Online]. Available at: <http://www.fots.ua.ac.be/events/grabats2008/>. Accessed on July 21, 2008.
2. Zeigler, B. *Multifaceted Modelling and Discrete Event Simulation*. New York: Academic Press, 1984.
3. Bolduc, J.-S., and H. Vangheluwe. "The modelling and simulation package pythonDEVS for classical hierarchical DEVS," MSDL Technical Report MSDL-TR-2001-01. Montreal, Quebec, Canada: McGill University, June 2001.
4. Syriani, E., and H. Vangheluwe. "Using MoTif for the AntWorld simulator case study," in *GraBaTs 2008 Tool Contest*. Edited by P. Van Gorp and A. Rensink. [Online]. Available at: <http://fots.ua.ac.be/events/grabats2008/>, Leicester, UK, 2008.
5. de Lara, J., and H. Vangheluwe. "AToM³: A tool for multi-formalism and meta-modelling," in *FASE'02*, series LNCS, Vol. 2306. Edited by R.-D. Kutsche and H. Weber, 174–88, April 2002, Grenoble, France. New York: Springer-Verlag, 2002.
6. Syriani, E., and H. Vangheluwe. "Programmed graph rewriting with DEVS," in *AGTIVE'07*, series LNCS. Edited by M. Nagl and A. Schürr. New York: Springer-Verlag, October 2007.
7. Ehrig, H., G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Edited by G. Rozenberg. Hackensack, NJ: World Scientific Publishing Co., 1997.
8. Guerra, E., and J. de Lara. "Event-driven grammars: Relating abstract and concrete levels of visual languages," *SoSym* 6, no. 6 (2007): 317–47.
9. Gyapay, S., R. Heckel, and D. Varró. "Graph transformation with time: Causality and logical clocks," in *Proceedings of ICGT 2002: 1st International Conference on Graph Transformation*, series LNCS, Vol. 2505, 120–34, October 2002, Barcelona, Spain. New York: Springer-Verlag, 2002.
10. Syriani, E., and H. Vangheluwe. "Programmed graph rewriting with time for simulation-based design," in *ICMT'08*, series LNCS, Vol. 5063. Edited by A. Pierantonio, A. Vallecillo, J. Bézivin, and J. Gray, 91–106, July 2008, Zürich, Switzerland. New York: Springer-Verlag, 2008.
11. Hong, J. S., H.-S. Song, T. G. Kim, and K. H. Park. "A real-time discrete event system specification formalism for seamless real-time software development," *DEDS* 7 (1997): 355–75.
12. Agrawal, A. "Metamodel based model transformation language," in *OOPSLA'03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 386–87, Anaheim, CA. New York: ACM Press, 2003.
13. Agrawal, A., G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo. "The design of a language for model transformations," *SoSym* 5, no. 3 (September 2005): 261–88.
14. Vizhanyo, A., A. Agrawal, and F. Shi. "Towards generation of high-performance transformations," in *Proceedings of the Third International Conference on Generative Programming and Component Engineering*, series LNCS, Vol. 3286. Edited by G. Karsai and E. Visser, 298–316. New York: Springer-Verlag, 2004.
15. Lengyel, L., T. Levendovszky, G. Mezei, and H. Charaf. "Control flow support in meta-model-based model transformation frameworks," in *EUROCON'05*, 595–98, November 2005, Belgrade, Serbia. New York: IEEE, 2005.

16. Lengyel, L., T. Levendovszky, G. Mezei, and H. Charaf. "Model transformation with a visual control flow language," *IJCS* 1, no. 1 (2006): 45–53.
17. Object Management Group. *Unified Modeling Language Superstructure*, [Online]. Available at: <http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>, Object Management Group, February 2009.
18. Blostein, D., and A. Schürr. "Computing with graphs and graph rewriting," *SPE* 9, no. 3 (1999): 1–21.
19. Nickel, U., J. Niere, and A. Zündorf. "Tool demonstration: The FUJABA environment," in *ICSE'00*, 742–45, June 2000, Limerick, Ireland. New York: ACM Press, 2000.
20. Fischer, T., J. Niere, L. Turunski, and A. Zündorf. "Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java," in *Theory and Application of Graph Transformations*, series LNCS, vol. 1764, 296–309, November 2000, Paderborn, Germany. New York: Springer-Verlag, 2000.
21. Amelunxen, C., A. Königs, T. Röttschke, and A. Schürr. "MOFLON: A standard-compliant meta-modeling framework with graph transformations," in *Model Driven Architecture—Foundations and Applications: Second European Conference*, series LNCS, Vol. 4066. Edited by A. Rensink and J. Warmer, 361–75. New York: Springer-Verlag, 2006.
22. Schürr, A. "Specification of graph translators with triple graph grammars," in *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, series LNCS, Vol. 903. Edited by G. Tinhofer, 151–163, June 1994, Heidelberg, Germany. New York: Springer-Verlag, 1994.
23. Chow, A.C.-H., and B. Zeigler. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *TSCS* 13 (1996): 55–67.
24. Varró, D., and A. Balogh. "The model transformation language of the VIATRA2 framework," *Science of Computer Programming* 68, no. 3 (2007): 214–34.

2 System State Identification Using DEVS

*Norbert Giambiasi, Diego Llarrull,
and Maximiliano Cristiá*

CONTENTS

2.1	Introduction	29
2.2	Preliminaries	31
2.2.1	Execution Fragments and Traces of DEVS Models	31
2.2.2	DEVS-Based Hierarchy Formalisms	33
2.2.3	Formalism Hierarchy	33
2.2.3.1	Interpretation: Simple.DEVS	36
2.3	Extending Mealy Machines Fault Detection Techniques To U.DEVS	37
2.3.1	Minimality—State Equivalence	38
2.3.2	Experiments on U.B.DEVS Models	40
2.3.3	Sequence Finding	43
2.4	Simple Bipartite DEVS (S.B.DEVS)	45
2.4.1	Minimization Procedure for S.B.DEVS Models	46
2.4.2	Fault Detection Techniques on S.B.DEVS Models	47
2.4.3	Experiments on S.B.DEVS Models	48
2.4.4	Sequence Finding	49
2.4.5	Testing Procedure	51
2.5	Conclusions and Future Work	53
	References	54

2.1 INTRODUCTION

In the 1960s, research was done on the problem of testing finite state machines to ensure their correct functioning and to discover aspects of their behavior. Due to its applications in testing communications protocols, the fault detection problem is still studied in the field of finite state machines and of some of their extensions.

An important class of testing problems is state identification (pioneered in the seminal 1956 paper of Moore) [1]. An extensive theory is available on this topic for finite state machines (see Lee and Yannakakis [2] for a survey and Kohavi's book for a good exposition of the major results published in papers on testing problems, in the 1960s

[3]). The testing problem was and still is studied in the context of timed Automata [4], which are used to model real-time systems with timed considerations [5].

Ordinary finite state machines are not powerful enough to model physical systems in an accurate way and, even though timed automata are well adapted for high level specifications, the Discrete Event System Specification (DEVS) formalism proposed by Zeigler et al. [8,9] (which can be seen as a general timed extension of finite state machines) seems more suitable for representing accurate timed behavior of dynamic systems. DEVS allows building discrete timed event abstractions of dynamic systems with clean simulation semantics and a clean correspondence between the real system and the basic concepts of the formalism (such as states, transitory states, state variables, events, etc.).

In this work, we propose an approach for system state identification using the DEVS formalism. In other words, we would like to deduce information on the states of the system by observing its input/output behavior. In fact, we are concerned with the following testing problems [2]:

- Determine the final state after the test
- Identify the initial state
- Verify the unknown initial state

A possible field of application for this work is the testing of discrete-event control systems for which timed considerations are generally needed, and a discrete-event abstraction of the system to be controlled is also needed in order to realize accurate timed analysis by simulation of a coupled model, which is composed of the control system and the system to be controlled.

In Dacharry et al. [6,7], a formal methodology for the design and verification of discrete-event control systems was proposed. Within this methodology, a high-level specification of a control system is given by a network of timed automata, and the corresponding implementation is expressed by a coupled DEVS model [8]. This allows the formal verification of the conformance of components (atomic DEVS models against timed automata) and the conformance of the whole model. Nevertheless, due to the state explosion problem that appears in the verification of models that deal with a dense time base, the automatic verification of the conformance between the high-level and the low-level models cannot be carried out in a large number of real cases. Therefore, a partial automatic validation of the conformance relation between an implementation and its specification can be possible by generating test cases on a high-level specification and applying these tests to the low-level model description or to an implementation. This kind of test, called conformance testing, is not developed in this work but the methods presented for minimization or for building some types of input sequences constitute a required step for conformance testing.

The chapter is organized as follows: In Section 2.0 we recall the existing theory, together with the tools and concepts that will be necessary to extend it. In Section 2.3, we introduce a subset of the DEVS formalism that we take under consideration, and we adapt and extend the existing methods, concepts, and definitions to this subset. In Section 2.4, we propose an extension of the first subset of models in order to

enlarge the spectrum of models to which the theory of fault detection can be applied, and we briefly show some considerations about the implementation of these testing methods. Finally, we conclude the chapter.

2.2 PRELIMINARIES

2.2.1 EXECUTION FRAGMENTS AND TRACES OF DEVS MODELS

We introduce the formal notion of executions or simulation runs of DEVS models and their traces as an alternative way to formally specify the full behavior of DEVS models [6]. These concepts are analogous to the ones that are commonly used for describing the behavior of timed automata.

Definition 2.2.1: Execution fragment of a DEVS model

An **execution fragment** of a DEVS model $\mathcal{D}$ is a finite alternating sequence $\Upsilon = v_0 z_1 v_1 z_2 v_2 \dots z_n v_n$, where:

1. pure time passage
Each v_i is a function from a real interval $I_i = [0, t_i]$ to the set of total states of $\mathcal{D}$, such that $\forall j, j' \in I_i \mid j < j'$, if $v_i(j) = (s, e)$ then $v_i(j') = (s, e + j' - j)$.
2. discrete-event transition
Each z_i is an input or output event, and if $(s, e) = v_{i-1}(\sup(I_{i-1}))$, $(s', 0) = v_i(\inf(I_i))$, one of the following conditions hold:
 - a. $z_i \in Y_{\mathcal{D}}$, $\delta_{int\mathcal{D}}(s) = s'$, $ta(s) = e$, and $\lambda(s) = z_i$
 - b. $z_i \in X_{\mathcal{D}}$, $\delta_{ext\mathcal{D}}(s, e, z_i) = s'$, and $e < ta(s)$.

Definition 2.2.2: Execution of a DEVS model

Let $\mathcal{D} = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$ be an atomic DEVS with the initial state s_0 . Then, an execution for $\mathcal{D}$ is an execution fragment of $\mathcal{D}$ that begins with the initial state s_0 .

We denote with $execs^*(\mathcal{D})$, $execs^\omega(\mathcal{D})$ and $execs(\mathcal{D})$ the sets of finite, infinite, and all executions of $\mathcal{D}$, respectively.

We denote *first* and *last* the functions giving the states of the first and the last state trajectories of an execution fragment, respectively:

$$last : execs^*(\mathcal{D}) \rightarrow S_p$$

$$last(\alpha) = s_j \quad \text{if } v_n(\sup(I_n)) = (s_j, t_n),$$

and

$$first : execs^*(\mathcal{D}) \rightarrow S_p$$

$$first(\alpha) = s_j \quad \text{if } v_0(0) = (s_j, 0).$$

The S_p is the set of passive states (i.e., states where $ta = \infty$) that belong to the state set S .

Definition 2.2.3: Trace of a DEVS model

A trace of an execution fragment $\Upsilon = v_0 z_1 v_1 z_2 v_2 \dots z_n v_n$, noted trace (Υ) of a DEVS model D is defined to be a tuple (θ_i, θ_o, t) such that θ_i and θ_o are sequences consisting of pairs of the form $(z_i, \text{sup}(I_{i-1}))$ where z_i is an input or an output event of (Υ), respectively, and their time of occurrence in (Υ) sorted in chronological order of occurrence. The t is the total time of execution, defined as $\sum_{0 \leq j \leq n} (\text{sup}(I_j))$. Formally, the time of occurrence of an event x_i of Υ is equal to $\sum_{0 \leq j \leq i} (\text{sup}(I_j))$, where I_j is the domain of v_j .

The set of all traces of a DEVS model, noted $\text{traces}(\mathcal{D})$, is defined as $\{\text{trace}(\Upsilon) \mid \Upsilon \in \text{execs}(\mathcal{D})\}$.

Remark: Note that for every DEVS model ($\mathcal{D}$), and given a sequence of pairs of input events and their respective time of occurrence, θ_i , its total time of execution t , and an initial state, s_0 , there exists a unique execution fragment $\gamma = v_0 x_1 v_1 \dots x_n v_n$ such that $\text{trace}(\gamma) = (\theta_i, \theta_o, t)$ and $v_0(0) = (s_0, 0)$.

This concept is known in the DEVS literature as determinism, and as such, all completely specified DEVS models are deterministic in this sense, considering that when an input event (an external transition) occurs at the same time that an internal transition is scheduled, we give the priority to the external transition.

For our test purpose of timed models, we need to introduce the concepts of slow timed execution fragment and slow timed input trace.

Definition 2.2.4: Slow Timed Execution Fragment

A timed execution fragment $\Upsilon = v_0 z_1 v_1 z_2 v_2 \dots z_n v_n$ of a DEVS atomic model $\mathcal{D} = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$ is a slow timed execution fragment if:

$$\forall v_j \in \Upsilon \text{ with } v_j = (s_k, e) \wedge s_k \in S_a, \text{sup}(I_j) = ta(s_k).$$

We denote as $\text{execs}^*_{slow}(\mathcal{D})$ as the set of the finite slow timed execution fragments of $\mathcal{D}$.

A slow timed execution fragment is then an execution fragment where external events are spaced in time in such a way that, whenever an input event is received by the model, the latter is already in a passive state.

Definition 2.2.5: Slow Timed Input Trace

An input trace θ_i is a slow timed input trace iff:

$$\exists \alpha_i, \alpha_i \in \text{execs}^*_{slow}(\mathcal{D}) \text{ so that } \text{trace}(\alpha_i) = (\theta_i, \theta_o, t)$$

Let us notice that, since only the output events are observed when black-box testing a DEVS model, then in order to be sure to have a slow input trace, we must wait τ time units between two input events with:

$$\tau > \max(ta(s_j) \mid s_j \in S_a).$$

2.2.2 DEVS-BASED HIERARCHY FORMALISMS

In Dacharry and Giambiasi [10], a hierarchy of discrete-event formalisms with increasing complexity was introduced, maintaining, at the same time, the clarity of concepts and cohesion in the structure. The proposed approach maintains cohesion and clarity by using a similar syntax in all proposed formalisms, and by formally specifying its semantic or operational behavior in order to avoid an incorrect interpretation of the formalism.

The syntax of all the proposed formalisms is based on the DEVS [9]. The DEVS formalism has been chosen for this purpose since it incorporates a solid mathematical basis along with a well-developed simulation infrastructure. Furthermore, it is considered to be a universal formalism for the modeling and the simulation of discrete-event systems.

The different formalisms that have been proposed range from a formalism based in the widely known concept of sequential machines (automata), to the more expressive DEVS formalism.

For our current purpose we recall, in the following, the two first formalisms of the hierarchy on which we have adapted testing methods initially proposed for Mealy machines.

2.2.3 FORMALISM HIERARCHY

The first and less expressive formalism we consider is an untimed discrete-event formalism (only the occurrence order of events is taken into account).

Definition 2.2.6: Untimed.DEVS (U.DEVS)

An atomic U.DEVS model is defined as a structure $\mathcal{D} = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$ where:

- X : finite set of input events
- Y : finite set of output events
- S : finite set of states, $S = S_a \cup S_p$
- δ_{int} : internal transition function ($\delta_{int} : S_a \rightarrow S_p$)
- δ_{ext} : external transition function ($\delta_{ext} : S_p \times X \rightarrow S_a$)
- λ : output function ($\lambda : S \rightarrow Y$)

In an U.DEVS model, as in the classical DEVS, the set of states is partitioned in two sets, S_a and S_p called active and passive state sets, respectively, $S = S_a \cup S_p$.

Only external transitions can occur in a passive state of a U.DEVS model (as in the classical DEVS), and only internal transitions can occur in an active state. In other words, δ_{ext} is not defined for the active states of an U.DEVS model.

Note that for U.DEVS, it is considered that X , Y , and S are finite sets.

Interpretation: U.DEVS

The model remains in a passive state s_i until it receives an input event x_j , the next state is given by $\delta_{ext}(s_i, x_j)$. When the model reaches an active state s_i it instantaneously

```

U.DEVS-atomic-simulator

when internal event, at time t:
if  $s \notin S_a$  then ERROR else
 $y = \lambda(s)$ 
send output event ( $y, t$ )
 $s = \delta_{int}(s)$ 

when receive input event  $x$ :
if  $s \notin S_p$  then ERROR else
 $s = \delta_{ext}(s, x)$ 

end U. DEVS-atomic-simulator

```

FIGURE 2.1 Pseudocode of a simulator for U.DEVS atomic models.

transits into the next state defined by $\delta_{int}(s_i)$ and emits the output event $\lambda(s_i)$. It is assumed that no input event can occur in an active state (Figure 2.1).

Remark: It should be clear that U.DEVS is very closely related to classical sequential machines. Therefore, in U.DEVS, the concepts of active (transitory) and passive (steady) states are clearly identified, which allows “clean” specifications of models with transitory and steady states.

Since U.B.DEVS is an untimed formalism (or it can be considered to be an untimed subset of the DEVS formalism), it is necessary to redefine the concepts of execution fragments, executions and traces for U.B.DEVS models, which we define next:

Definition 2.2.7: Execution Fragment of an U.B.DEVS Model

An **execution fragment** of a U.B.DEVS model $\mathcal{D}$ is a finite alternating sequence $Y = v_1 z_1 v_2 z_2 \dots z_n v_{n'}$ where:

- Each v_i is a possible state s_i of the system ($s_i \in S \forall i$).
- Each z_i is an input or output event and one of the following conditions hold:
 - a. $z_i \in Y_{\mathcal{D}}, \delta_{int}(v_i) = v_{i+1}, \lambda(s) = z_i$
(an output event is emitted by the model and then it internally transitions).
 - b. $z_i \in X_{\mathcal{D}}, \delta_{ext}(v_i, z_i) = v_{i+1}$
(an input event is received by the model and then it externally transitions).

Definition 2.2.8: Execution of an U.B.DEVS Model

Let $\mathcal{D} = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda)$ be an atomic U.B.DEVS model with the initial state s_0 . Then, an execution for $\mathcal{D}$ is an execution fragment of $\mathcal{D}$ that begins with the initial state s_0 .

We denote with $\text{execs}^*(\mathcal{D})$, $\text{execs}^\omega(\mathcal{D})$ and $\text{execs}(\mathcal{D})$ the sets of finite, infinite, and all executions of $\mathcal{D}$, respectively.

We denote *first* and *last* the functions giving the states of the first and the last state trajectories of an execution fragment, respectively:

$$\text{last} : \text{execs}^*(\mathcal{D}) \rightarrow S_p$$

$$\text{last}(\alpha) = s_j \quad \text{if } v_n = s_j,$$

and

$$\text{first} : \text{execs}^*(\mathcal{D}) \rightarrow S_p$$

$$\text{first}(\alpha) = s_j \quad \text{if } v_0 = s_j.$$

The S_p is the set of passive states (the states where input events are accepted according to the requirement that we previously stated) that belong to the state set S .

Definition 2.2.9: Trace of an U.B.DEVS Model

A trace of an execution fragment $\Upsilon = v_0 z_1 v_1 z_2 v_2 \dots z_n v_n$, noted trace (Υ), of a DEVS model $\mathcal{D}$ is defined to be a pair (θ_i, θ_o) such that θ_i and θ_o are sequences consisting of input and output events of (Υ), respectively, sorted in chronological order of occurrence (that is, if z_i and z_j are events of the same kind, i.e., either both input or both output events, and $i < j$, we say that z_i arrived “earlier in time” than z_j).

The set of all traces of a DEVS model, noted $\text{traces}(\mathcal{D})$, is defined as

$$\text{traces}(\mathcal{D}) = \{\text{trace}(\Upsilon) \mid \Upsilon \in \text{execs}(\mathcal{D})\}.$$

The next formalism in the hierarchy [10] is called Simple.DEVS. This formalism introduces the notions of:

- Phase
- Phase-lifetime

Phase is a state variable that ranges over a finite set. In fact, to each value of Phase a subset of the states of the model corresponds. Phase takes a finite number of values, and thus the partition of S defined by Phase is finite. A lifetime value is associated to each value p_i of Phase. This value is given by the lifetime function ta and is represented as $ta(p_i)$, which can be any positive real number, 0 or infinite. Phases with a 0 or real number lifetime value are called active phases, while the ones with an infinite lifetime are passive phases. Phase-lifetime is another state variable, $\text{Phase-lifetime} = ta(p_i)$ that equals the lifetime of the present value of Phase.

Notice that these definitions are identical to those of the classical DEVS, but for S.DEVS, the lifetime function depends only on the state variable Phase and

not on the other state variables. In fact, all the states with the same value of Phase belong to the same class and this means that all the states of a class have the same lifetime.

S.DEVS is a restricted formalism in the sense that it is not possible to model the temporal behavior of any discrete-event system. Its main restrictions are:

- The lifetime of a state depends only on one particular state variable, the Phase state variable.
- The output function depends only on Phase.

Definition 2.2.10: Simple.DEVS (S.DEVS)

An atomic Simple.DEVS model is defined as the structure:

$$\mathcal{D}_s = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where:

- X : finite set of input events
- Y : finite set of output events
- P : finite set of phases (possible values of the variable *Phase*)
- S : set of states
- $S = \{s_i = (s_{v1}, \dots, s_{vn}, Phase, \sigma) \mid s_{vi} \in V_i \wedge Phase \in P \wedge \sigma : \mathfrak{R}_0^+ \cup \{\infty\}, \sigma = ta(Phase)\}$
- δ_{int} : internal transitions ($\delta_{int} : S \rightarrow S$)
- δ_{ext} : external transitions ($\delta_{ext} : S \times X \rightarrow S$)
- λ : output function ($\lambda : S \rightarrow Y$)
- ta : lifetime function ($ta : P \rightarrow \mathfrak{R}_0^+ \cup \{\infty\}$), $ta(p_i)$ is the lifetime associated to the value p_i of *Phase*

A Simple.DEVS model is completely specified, if:

- ta is defined for all the possible values of *Phase*
- δ_{int} and λ are partial functions such that, $\forall p_i \in P$ with $ta(p_i) \neq \inf$, $\delta_{int}(s)$ and $\lambda(s)$ are defined
- δ_{ext} should be a function defined for all states and all input events

2.2.3.1 Interpretation: Simple.DEVS

When the model is in the state s_k with the value p_k of *Phase*, the maximum time it remains in the state s_k is given by $ta(p_k)$. If no external event occurs, when $ta(p_k)$ time is elapsed, the model generates the output event $\lambda(s_k)$ and it changes to the state $s_l = \delta_{int}(s_k)$. If before this internal transition an external event x occurs, the system transitions to the state $s_j = \delta_{ext}(s_k, x)$.

Notice that in S.DEVS, the next state does not depend on the elapsed time in the current state.

In the following, for the sake of simplicity we will concentrate only on these two subsets of atomic DEVS models. In addition, and without loss of generality, we consider models for which every state is totally defined by the two state variables, that is, *Phase* and *ta*.

2.3 EXTENDING MEALY MACHINES FAULT DETECTION TECHNIQUES TO U.DEVS

In order to make a progressive approach to the problem, we first present a procedure for obtaining an U.DEVS with the same behavior than a Mealy machine. The final objective remains to apply to DEVS models some fault detection techniques developed on the Mealy Machines.

We recall that a Mealy Machine [3] is formally stated as a quintuple $M = (I, O, S, \delta, \lambda)$ where I , O , and S are finite and nonempty sets of input symbols, output symbols, and states, respectively, $\delta : S \times I \rightarrow S$ is the state transition function and $\lambda : S \times I \rightarrow O$ is the output function.

The first step in defining an U. DEVS model with the same behavior than a given Mealy machine is to add an active state for every transition of the Mealy machine. Then, every state transition of a Mealy machine $\mathcal{M} = (I_{\mathcal{M}}, O_{\mathcal{M}}, S_{\mathcal{M}}, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}})$ that has the form:

$$s_i \xrightarrow{x/y} s_j, x \in I_{\mathcal{M}}, y \in O_{\mathcal{M}}, s_i, s_j \in S_{\mathcal{M}},$$

is translated into two transitions in the corresponding U.DEVS model $\mathcal{D} = \langle X_{\mathcal{D}}, Y_{\mathcal{D}}, S_{\mathcal{D}}, \delta_{int_{\mathcal{D}}}, \delta_{ext_{\mathcal{D}}}, \lambda_{\mathcal{D}} \rangle$:

- An external transition of the form $s_i \xrightarrow{x/y} s_{i,x}$ becomes $d_{extD}(s_i, x) = s_{i,x}$, $s_i \in S_{p_{\mathcal{D}}}$, $x \in X_{\mathcal{D}}$ and $s_{i,x} \in S_{a_{\mathcal{D}}}$.
- An internal transition of the form $s_{i,x} \xrightarrow{-/y} s_j$ becomes $\delta_{intD}(s_{i,x}) = s_j$, $\lambda(s_{i,x}) = y$, $s_{i,x} \in S_{a_{\mathcal{D}}}$, $y \in Y_{\mathcal{D}}$ and $s_j \in S_{p_{\mathcal{D}}}$.

An U.DEVS $\mathcal{D} = \langle X_{\mathcal{D}}, Y_{\mathcal{D}}, S_{\mathcal{D}}, \delta_{int_{\mathcal{D}}}, \delta_{ext_{\mathcal{D}}}, \lambda_{\mathcal{D}} \rangle$ and a Mealy machine $\mathcal{M} = (I_{\mathcal{M}}, O_{\mathcal{M}}, S_{\mathcal{M}}, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}})$ have the same behavior if:

1. Both models have the same input and output event sets, $I_{\mathcal{M}} = X_{\mathcal{D}} \wedge O_{\mathcal{M}} = Y_{\mathcal{D}}$.
2. The set of passive states of $\mathcal{D}$ is equal to the set of all states of $\mathcal{M}$: $S_{\mathcal{M}} = S_{p_{\mathcal{D}}}$.
3. For each possible transition $\delta_{\mathcal{M}}(s_i, x) = s_j$, of $\mathcal{M}$ there exist two transitions in $\mathcal{D}$:

$$\delta_{extD}(s_i, x) = s_{i,x} \text{ and } \delta_{intD}(s_{i,x}) = s_j.$$

4. For each possible output event $y \in I_{\mathcal{M}}$ such that $\lambda_{\mathcal{M}}(s_i, x) = y$, it is required that an analogous output event in $\mathcal{D}$

$$\lambda_{\mathcal{M}}(s_i, x) = y \Leftrightarrow \lambda_{\mathcal{D}}(s_{i,x}) = y.$$

Definition 2.3.1

An Untimed.DEVS is a bipartite DEVS iff:

- $\forall s_i \in S_p \wedge \forall x_i \in X, \delta_{\text{ext}}(s_i, x_i) = s_k \Rightarrow s_k \in S_a.$
- $\forall s_k \in S_a, \delta_{\text{int}}(s_k) = s_i \Rightarrow s_i \in S_p.$

For an Untimed Bipartite DEVS (U.B.DEVS), we introduce the concept of a passive transition function, noted δ_{pass} .

Definition 2.3.2

The passive transition function of a U.B.DEVS is defined as follows: $\forall x_i \in X, s_i \in S_p,$

$$\delta_{\text{pass}}(s_i, x_i) = \delta_{\text{int}}(\delta_{\text{ext}}(s_i, x_i)) = s_j \in S_p.$$

Analogously, we also introduce the concept of a passive output function, noted λ_{pass} .

Definition 2.3.3

The passive output function of a U.B.DEVS is defined as follows: $\forall x_i \in X, s_i \in S_p,$

$$\lambda_{\text{pass}}(s_i, x_i) = \lambda(\delta_{\text{ext}}(s_i, x_i)) = y_i \in Y.$$

2.3.1 MINIMALITY—STATE EQUIVALENCE

The definition and properties presented in Kohavi [3] for Mealy Machines can be easily extended to U.B.DEVS, by considering the following theorem.

Theorem 2.3.1

To every U.B.DEVS model a Mealy Machine corresponds with the same behavior and vice-versa.

Proof: It is straightforward to prove that, by considering only the passive transition and output functions of a U.B.DEVS, the existence of active states in this model is concealed. Thus, the model externally appears to have the same set of states than its corresponding Mealy Machine. Then, each pair of values $[\delta_{\text{pass}}(s_{ij}, x_j), \lambda_{\text{pass}}(s_{ij}, x_j)]$ corresponds to the pair $[\delta(s_{ij}, x_j), \lambda(s_{ij}, x_j)]$ of the Mealy Machine, which is equivalent to saying that they have the same behavior.

In Figure 2.2, we give an example of a state graph of a Mealy Machine (Figure 2.2a) and of the state graph of its corresponding U.B.DEVS (Figure 2.2b).

In order to apply testing methods from Mealy Machines to U.B.DEVS, we need first to define the basic concepts of k -distinguishable states, state equivalence, and state minimization for U.B.DEVS.

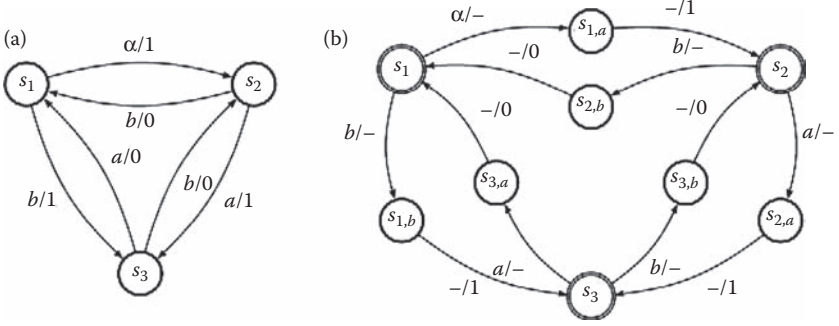


FIGURE 2.2 Mealy machine and its corresponding U.B.DEVS.

Definition 2.3.4: K-distinguishable Passive States

Two passive states s_i and s_j of a U.B.DEVS model $\mathcal{D}$ are **k-distinguishable** if and only if there exist at least two execution fragments α and β of $\mathcal{D}$ with traces $(\alpha) = (\theta_{i_\alpha}, \theta_{o_\alpha})$, traces $(\beta) = (\theta_{j_\beta}, \theta_{o_\beta})$ where:

$$\text{first}(\alpha) = s_i, \text{first}(\beta) = s_j$$

$$\theta_{i_\alpha} = \theta_{j_\beta}, \text{length}(\theta_{i_\alpha}) = k \text{ and } \theta_{o_\alpha} \neq \theta_{o_\beta}.$$

Definition 2.3.5: K-equivalence of States

Two **passive** states s_i and s_j of a U.B.DEVS model $\mathcal{D}$ are **k-equivalent** if and only if they are not **k-distinguishable**.

Definition 2.3.6: State Equivalence

Two **passive** states s_i and s_j of a U.B.DEVS model $\mathcal{D}$ are **equivalent** if and only if they are **k-equivalent** $\forall k \in \mathbb{N}$.

Definition 2.3.7: Equivalence of U.B.DEVS

Two U.B.DEVS models, $\mathcal{D}_1$ and $\mathcal{D}_2$, are **equivalent** if and only if, for every state in $\mathcal{D}_1$, there is a corresponding equivalent state in $\mathcal{D}_2$ and vice-versa.

Definition 2.3.8: Minimal U.B.DEVS

A U.B.DEVS model is **minimal** (reduced) if and only if no two states in it are equivalent.

The concept of minimality is important in the field of fault detection techniques due to the fact that most results are obtained under the hypothesis of minimality. We recall that the minimization procedure is applied under the hypothesis that the machine under consideration is *completely specified*; that is to say, for every state, there is a transition specified for every input event.

The minimization procedure detailed in Kohavi [3] can be applied to *completely specified* U.B.DEVS models if only the passive states, the passive transition function δ_{pass} and the passive output function λ_{pass} are considered. Whenever two equivalent passive states are found, it is necessary to remove not only one of these states, but also *every active state to which such state can transition*. Then, every transition that reaches a state that has been deleted has to be redirected to its equivalent state.

Extending the result from Mealy Machines we can say that every U.B.DEVS model $\mathcal{D}$ corresponds with a minimal U.B.DEVS $\mathcal{D}^*$, which is equivalent to $\mathcal{D}$ and is unique up to isomorphism, provided $\mathcal{D}$ is *completely specified* (the external transition function is defined for all possible input events).

The existence of a minimal form for every completely specified U.B.DEVS model ensures that fault detection techniques defined for Mealy Machines can be applied to U.B.DEVS models, after finding its minimal or reduced form.

Example

Let us consider the state graph of a U.B.DEVS (Figure 2.3a), its passive transition table (a table with the passive transition function and the passive output function) is represented in Figure 2.3b.

By applying the minimization procedure given in Kohavi [3], we obtain the minimal U.B.DEVS model by replacing each equivalent class of states by one state. In this example, the states s_2 and s_3 are equivalent; replacing these two states by s_3 , we obtain, for this U.B.DEVS model, the state graph and the transition table of Figure 2.3c and d.

2.3.2 EXPERIMENTS ON U.B.DEVS MODELS

In a testing problem, we consider an implementation (seen as a black box) of a U.B.DEVS about which we would like to deduce some information by observing its input/output behavior. An input event sequence is applied to the system and the output sequence is observed in order to infer the needed information. The input sequence can be preset if it is fixed ahead of time, or can be adaptive if the next input event depends on the previously observed outputs.

We will proceed by adapting concepts from the existing fault detection experiment theory for Mealy Machines presented in Kohavi [3]. In particular, we present the definitions of distinguishing, homing, and synchronizing sequences. Afterward, we propose the adapted definitions of the experiments introduced in Kohavi [3] that we utilize in this work.

In the following, we denote with $execs_k(\mathcal{D})$ the set of finite executions of $\mathcal{D}$ with the same input trace θ_{I_k} (let us recall that θ_I means “input trace” while θ_O means “output trace”).

Definition 2.3.9: Distinguishing Sequence

Let us consider two execution fragments α and β of a U.B.DEVS $\mathcal{D}$ with traces $(\alpha) = (\theta_{I_\alpha}, \theta_{O_\alpha})$, traces $(\beta) = (\theta_{I_\beta}, \theta_{O_\beta})$ such that:

$$first(\alpha) = s_i, first(\beta) = s_j$$

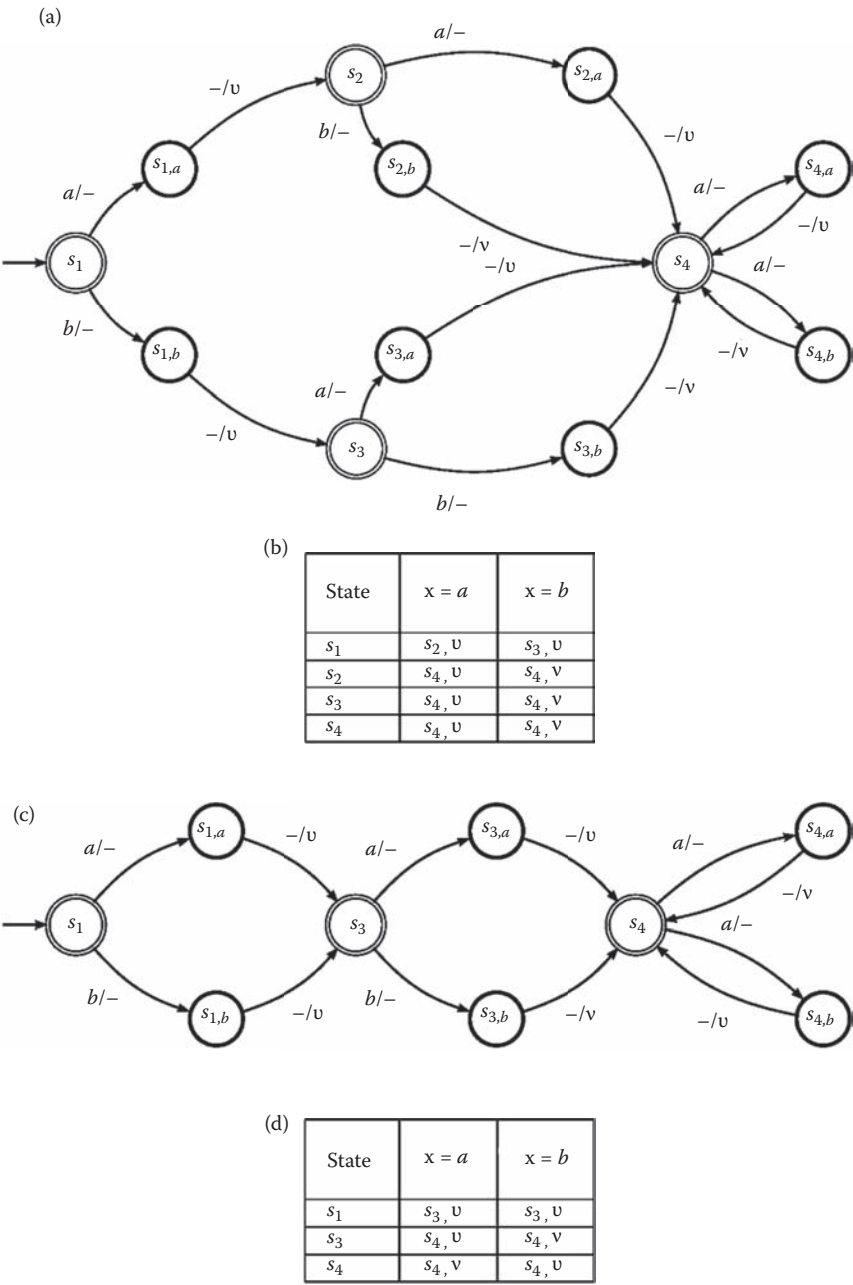


FIGURE 2.3 An U.B.DEVS model (a) with its associated transition table (b). Resulting minimal U.B.DEVS model (c) after the deletion of s_2 (which is equivalent to s_3), $s_{2,a}$, and $s_{2,b}$, and the resulting transition table (d).

$$\text{and } \theta_{\alpha} = \theta_{\beta}$$

Then, $\theta_{\alpha} (= \theta_{\beta})$ is called a **distinguishing sequence** of the pair (s_i, s_j) iff

$$\theta_{O_{\alpha}} \neq \theta_{O_{\beta}}.$$

If the shortest distinguishing sequence of the pair (s_i, s_j) has a length k , then, s_i and s_j are said to be **k -distinguishable**.

The aim of a homing sequence is to determine the final state of a system observing its outputs. The homing sequence problem was completely solved for sequential machines Kohavi [3].

Definition 2.3.10: Homing Sequence

Let $\text{execs}_k(\mathcal{D})$ be the subset of finite execution fragments with the same input trace θ_{ik} , then θ_{ik} is a homing sequence iff:

$$\forall \alpha_i, \alpha_j \in \text{execs}_k(\mathcal{D}) \text{ such that } \theta_{O_i} = \theta_{O_j}, \text{ then } \text{last}(\alpha_i) = \text{last}(\alpha_j).$$

It should be clear that only reduced U.B.DEVS have homing sequences, since equivalent states cannot be distinguished. Every reduced U.B.DEVS has a homing sequence.

A synchronizing sequence gives the same final state, regardless of the initial state.

Definition 2.3.11: Synchronizing Sequence

An input trace θ_{ik} is a synchronizing sequence iff there exists a unique state $s_i \in S_p$ such that $s_j = \text{last}(\alpha_j) \forall \alpha_j \in \text{execs}_k(\mathcal{D})$.

In others words, this definition states the following: if there exists a sequence of input events such that every finite execution having this sequence as its input trace has the same final state, then that sequence (when viewed as an input trace) is a synchronizing one.

Having defined the kinds of sequences that are necessary for our test purposes, we can now introduce the different types of test experiments on a U.B.DEVS.

Definition 2.3.12: Preset Experiment

A preset experiment for a U.B.DEVS is any input event sequence π of the form $x_1 x_2 \dots x_n$ where $x_i \in X_{\mathcal{D}} \forall i = 1..n$. The sequence of output events (the output trace) that $\mathcal{D}$ generates in response to π is the result of the experiment.

Definition 2.3.13: Adaptive Experiment

An adaptive experiment for a U.B.DEVS model $\mathcal{D}$ is any input event sequence $\pi = x_1 x_2 \dots x_n$ where $x_i \in X_{\mathcal{D}} \forall i = 1..n$ provided that there exists a function

$$f : Seq(X_D) \times Y_D \rightarrow X_D$$

such that $\forall i = 2..n, \exists y_i \in Y_D \mid x_i = f(\langle x_0, \dots, x_{i-1} \rangle, y_i)$.

In other words, the value of the i th input event of an adaptive experiment π depends on all the previous input events and on an (unspecified) output event. The sequence of output events (the output trace) that $\mathcal{D}$ generates in response to π is the result of the experiment.

Since there are many possible definitions for f , in order to clearly identify those functions that are of interest for our purposes, we define the concept of a *valid adaptive experiment*.

Definition 2.3.14: Valid Adaptive Experiment

A valid adaptive experiment $\pi = x_1 x_2 \dots x_n$ is one in which the adaptive function f verifies the following property:

$$x_1 = f(\langle x_0 \rangle, y_0) \Leftrightarrow y_0 = \lambda_D(\delta_{\text{ext } \mathcal{D}}(s_0, x_0))$$

$$x_i = f(\langle x_0, \dots, x_{i-1} \rangle, y_{i-1}) \Leftrightarrow y_{i-1} = \lambda_D(\delta_{\text{ext } \mathcal{D}}(f(\langle x_0, \dots, x_{i-2} \rangle, y_{i-2}), x_{i-1})) \forall i = 2..n.$$

In this way, the value of the i th input event of π depends on all the previous input events and the last output event the model has generated. This is equal to saying that the i th input event depends on all the output events that the U.B.DEVS model has generated so far.

We make this distinction between *adaptive* and *valid adaptive experiments* because we consider that, for other purposes, our definition of a valid adaptive experiment would not be necessarily the most suitable. Other kinds of functions can be applied so as to change the behavior of the experiment. Still, all these experiments must share the property of being *adaptive*, hence the reason to make this definition explicit, though in the rest of this work it won't be mentioned again.

Definition 2.3.15: Distinguishing, Homing, Synchronizing Experiment

A valid experiment (that is, a preset experiment or a valid adaptive experiment) is distinguishing, homing, or synchronizing if the input sequence is distinguishing, homing, or synchronizing, respectively.

2.3.3 SEQUENCE FINDING

In order to perform a *homing experiment* on a U.B.DEVS model, the procedure described in Kohavi [3] can be applied considering only the passive states.

Theorem 2.3.2

A preset homing sequence, whose length is at most $(n - 1)^2$, exists for every minimal U.B.DEVS model $\mathcal{D}$, where n is the number of passive states in $\mathcal{D}$.

Figure 2.4 is an example of a U.B.DEVS model and the homing sequence obtained by the procedure [3] considering only the passive states of the model.

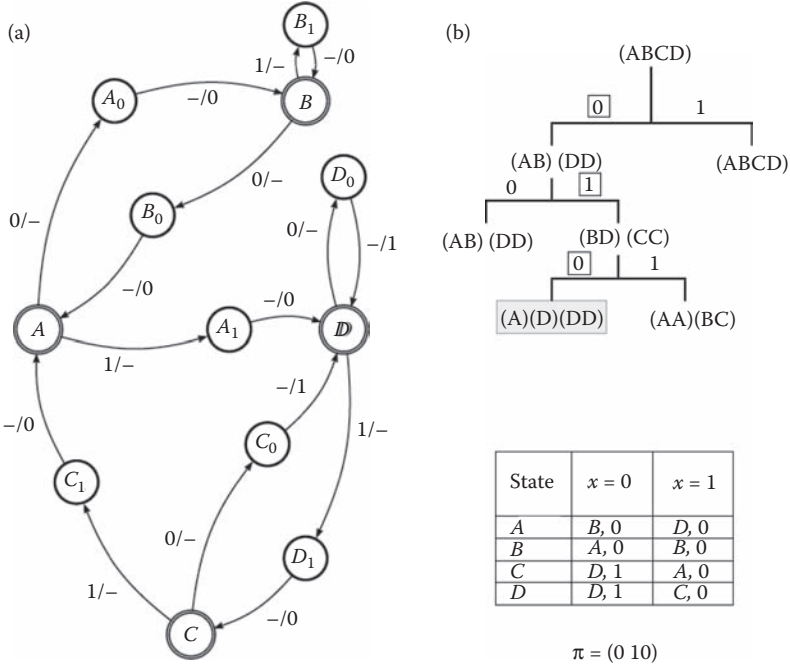


FIGURE 2.4 An U.B.DEVS model (a) with its homing tree, associated transition table, and a homing sequence (π) for it (b).

Distinguishing and *synchronizing* sequences can be obtained by the methods presented in Kohavi [3]. It is straightforward to prove that all the properties and theorems are valid for U.B.DEVS models considering only the passive transition function and the passive output function. In particular, the following result is valid also for U.B.DEVS models.

Theorem 2.3.3

If there exists a synchronizing sequence for a U.B.DEVS model $\mathcal{D}$ that has n passive states, then its length is at most $(n - 1)^2 n/2$.

Proof: See Kohavi [3], p. 458.

The following result sums up the preceding discussion.

Theorem 2.3.4

Let $\pi = \langle x_1\ x_2\ \dots x_n \rangle$ be either a synchronizing, homing, preset distinguishing, or adaptive distinguishing sequence for a Mealy Machine M . Then the sequence π is, respectively, either a synchronizing, homing, preset distinguishing or adaptive distinguishing sequence for the U.B.DEVS model $\mathcal{D}$ obtained from M .

Up to this point, we have shown it is possible to define concepts within U.B.DEVS theory that are equivalent to concepts from Mealy Machines. In the following section

we extend all these results to timed models, considering the Simple.DEVS subset of DEVS models.

2.4 SIMPLE BIPARTITE DEVS (S.B.DEVS)

It should be evident that Untimed.DEVS represents a tiny subset of the systems than can be modeled using the DEVS formalism. Then, we need to expand this subset in order to apply fault detection techniques to a wider range of DEVS models. In this first approach, the models considered are Simple.DEVS models (see Definition 2.2.10):

Definition 2.4.1: Simple Bipartite DEVS (S.B.DEVS)

A Simple.DEVS model is a S.B.DEVS if and only if

$$S_{\mathcal{D}} = S_{a\mathcal{D}} \cup S_{p\mathcal{D}} \text{ where } S_{a\mathcal{D}} \cap S_{p\mathcal{D}} = \emptyset.$$

$$\delta_{ext\mathcal{D}} : S_{p\mathcal{D}} \times X \rightarrow S_{a\mathcal{D}}$$

$$\delta_{int\mathcal{D}} : S_{a\mathcal{D}} \rightarrow S_{p\mathcal{D}}$$

$$\lambda_{\mathcal{D}} : S_{a\mathcal{D}} \rightarrow Y_{\mathcal{D}}$$

$$ta_{\mathcal{D}} : S_{\mathcal{D}} \rightarrow \mathfrak{R}_0^+ \cup \{\infty\}, \text{ where}$$

$$\forall s_i \in S_{p\mathcal{D}} \bullet ta_{\mathcal{D}}(s_i) = \infty, \text{ and}$$

$$\forall s_j \in S_{a\mathcal{D}} \bullet ta_{\mathcal{D}}(s_j) = k, k \in \mathfrak{R}_0^+.$$

The S.B.DEVS models constitute a subset of the set of Simple.DEVS models, having the following restrictions:

- Two consecutive states cannot belong to the same class, the next states of a passive state are active and vice-versa:

$$s_i \in S_a, \delta_{int}(s_i) = s_k \Rightarrow s_k \in S_p \text{ and } s_i \in S_p, \\ \forall x_i \in X, \delta_{ext}(s_i, x_i) = s_k \Rightarrow s_k \in S_a.$$

- No external event can occur in an active state:

$$\forall s_i \in S_a, \delta_{ext} \text{ is undefined.}$$

Remark: In fact, the “no external event can occur in an active state” hypothesis is the classical hypothesis used in the field of sequential machines: “no input event can occur during a transitory state.”

Definition 2.4.2: Distinguishing Sequence

Two passive states, s_i and s_j of a S.B.DEVS model $\mathcal{D}$ are distinguishable if and only if there exists at least two execution fragments $\alpha = v_{\alpha_0} x_{\alpha_1} v_{\alpha_1} x_{\alpha_2} v_{\alpha_2} \cdots x_{\alpha_n} v_{\alpha_n}$

and $\beta = v_{\beta_0} x_{\beta_1} v_{\beta_1} x_{\beta_2} v_{\beta_2} \cdots x_{\beta_n} v_{\beta_n}$ of D with trace $(\alpha) = (\theta_{i_\alpha}, \theta_{o_\alpha}, t_\alpha)$, trace $(\beta) = (\theta_{i_\beta}, \theta_{o_\beta}, t_\beta)$ where $v_{\alpha_0}(e) = (s_i, ta(s_i)) \forall e \in I_{\alpha_0}$, $v_{\beta_0}(e) = (s_j, ta(s_j)) \forall e \in I_{\beta_0}$, $\theta_{i_\alpha} = \theta_{i_\beta}$ and $\theta_{o_\alpha} \neq \theta_{o_\beta}$. The timed sequence θ_{i_α} (and θ_{i_β}) is called a distinguishing sequence of the pair (s_i, s_j) . If there exists for the pair (s_i, s_j) , a distinguishing sequence of length k , then the states in (s_i, s_j) are said to be k -distinguishable.

As we did for the U.B.DEVS models, we introduce the concepts of passive transition function, noted δ_{pass} , and passive output function, noted λ_{pass} for S.B.DEVS.

Definition 2.4.3: Passive Transition Function

The passive transition function of a S.B.DEVS model is defined as follows:

$$\forall x_i \in X, s_i \in S_p, \quad \delta_{pass}(s_i, x_i) = (\delta_{int}(\delta_{ext}(s_i, x_i)), ta(\delta_{int}(s_i)))$$

Definition 2.4.4: Passive Output Function

The passive output function of a S.B.DEVS model is defined as follows:

$$\forall x_i \in X, s_i \in S_p, \quad \lambda_{pass}(s_i, x_i) = \lambda(\delta_{ext}(s_i, x_i)) = y_i \in Y.$$

2.4.1 MINIMIZATION PROCEDURE FOR S.B.DEVS MODELS

A state transition table with the passive transition function can be used to represent a S.B.DEVS model, in this case, by adding the value of the lifetime of the next active state (Figure 2.5).

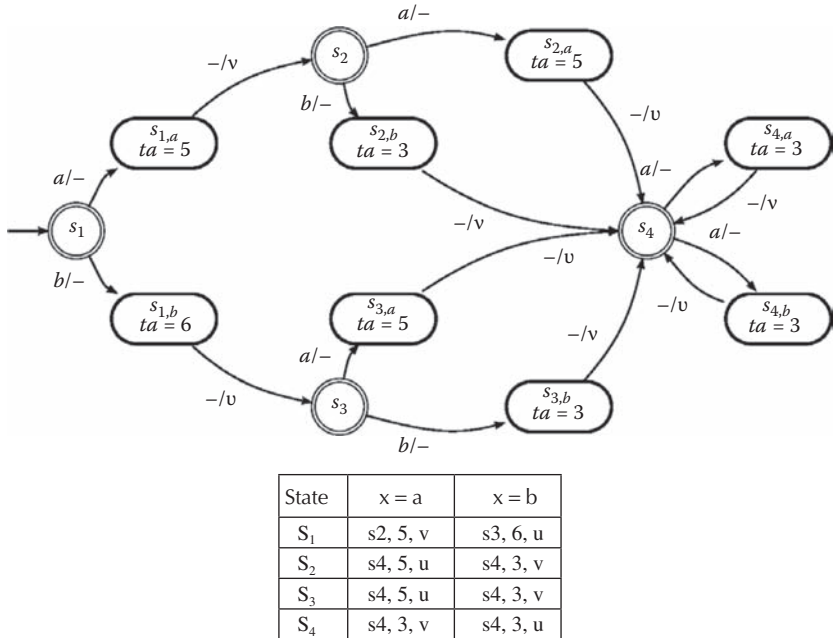


FIGURE 2.5 The S.B.DEVS model with its associated transition table.

The minimization procedure defined in Kohavi [3] is extended in order to take into account the value of $ta(s_{i,x})$ (lifetime of the active state following a passive state s_i that received the input event x).

In the first step of this procedure, the set of passive states is partitioned in such a way that each class contains one-equivalent passive states. We recall that two passive states s_i and s_j are one-equivalent iff for every input event they produce the same output event *at the same time*, formally:

$$\forall s_k \in \text{Successor}(s_i), \exists s_l \in \text{Successor}(s_j) \text{ such that } \lambda(s_k) = \lambda(s_l) \wedge ta(s_k) = ta(s_l)$$

$$\Rightarrow s_i \text{ and } s_j \text{ are 1-equivalent.}$$

In the example of Figure 2.5, the first partition of the state set according to the output values and the lifetime of the considered active states is:

$$P_1 = \{(s_1), (s_4), (s_2, s_3)\}.$$

Notice that s_1 and s_4 are not in the same class because the lifetime of their next active states are different, then the time stamps of the corresponding output events are different. These two states cannot be distinguished by the values of the output events but by the occurrence times of these events. On the other hand, s_2 and s_3 are one-equivalent.

By building the second partition (formed by two-equivalent states), we conclude that s_2 and s_3 are two-equivalent, since both transition to s_4 after the first input event (and after its corresponding output event). Moreover, since every input event in s_4 triggers a loop, s_2 and s_3 are k -equivalent $\forall k \geq 3$. Then, we conclude that s_2 and s_3 are two-equivalent states, then we can choose s_3 to represent the equivalence class. Figure 2.6 gives the reduced form of the S.B.DEVS example.

2.4.2 FAULT DETECTION TECHNIQUES ON S.B.DEVS MODELS

In order to respect the hypothesis that no external event can occur in an active state, all the input traces considered are slow timed input traces.

The definitions of homing and synchronizing sequences must be extended taking into account the timed aspects of the formalism.

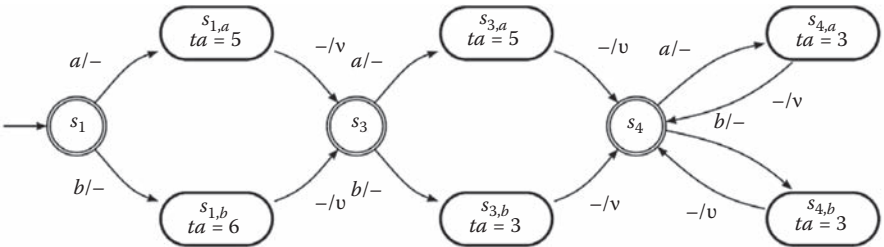


FIGURE 2.6 Resulting minimal S.B.DEVS model.

In the following, we denote with $\text{execs}_k(\mathcal{D})$ the set of slow timed and finite executions of $\mathcal{D}$ with the same timed input trace θ_{I_k} .

Definition 2.4.5: Homing Sequence

Let $\text{execs}_k(\mathcal{D})$ the subset of slow timed execution fragments with the same input trace $\theta_{I_k} = v_{i0}e_{i0}v_{i1}e_{i1}\dots\dots e_{in}v_{in}$, then θ_{I_k} is a homing sequence iff:

$$\forall \alpha_i, \alpha_j \in \text{execs}_k(\mathcal{D}) \text{ such that } \theta_{O_i} = \theta_{O_j}, \text{ then } \text{last}(\alpha_i) = \text{last}(\alpha_j).$$

Definition 2.4.6: Synchronizing Sequence

A timed input trace θ_{I_k} is a synchronizing sequence iff there exists a unique state $s_i \in S_p$ such that $(s_i, 0) = v_{in}(\text{Inf}(I_{I_k})) \quad \forall \alpha_i^k \in \text{execs}_k(\mathcal{D})$.

2.4.3 EXPERIMENTS ON S.B.DEVS MODELS

The definition of a preset experiment (PX) for S.B.DEVS models is adapted for handling the timed aspects of such models. We introduce the concept of a timed relative input trace in order to refer not to the absolute time of each event, but to its relative time with respect to the previous input event. That is, given a timed input trace $\theta_I = \langle (x_0, t_0), (x_1, t_1), \dots, (x_n, t_n) \rangle$, we refer not to θ_I , but to $\text{rel}(\theta_I)$, where the function rel is defined as follows:

$$\text{rel}(\langle (x_0, t_0), (x_1, t_1), \dots, (x_n, t_n) \rangle) = \langle (x_0, 0), (x_1, t_1 - t_0), \dots, (x_n, t_n - t_{n-1}) \rangle.$$

It is easy to show that rel is bijective, so it is equivalent to talk about either θ_I or $\text{rel}(\theta_I)$.

Definition 2.4.7: Preset Experiment

A preset experiment for a S.B.DEVS model $\mathcal{D}$ is any pair (θ_I, τ) where $\theta_I \in (X_D \times \mathfrak{R}_0^+)^*$ is a timed input trace of the form $\langle (x_0, t_0), \dots, (x_n, t_n) \rangle$ and $\tau \in \mathfrak{R}_0^+$ is the time during which the experiment must go on after the last event in the input trace. The output trace that $\mathcal{D}$ generates in response to θ_I after l units of time is the result of the experiment, and l is the length of the experiment, where $l = \sum_{i=0}^n t_i + \tau$.

It is straightforward to see that not all sequences that belong to $(X_D \times \mathfrak{R}_0^+)^*$ can be applied to a given S.B.DEVS model. The subset of sequences that will be accepted constitutes valid experiments.

Definition 2.4.8: Valid Preset Experiment

A valid preset experiment for a S.B.DEVS model $\mathcal{D}$ is any pair (θ_I, τ) , where $\theta_I \in (X_D \times \mathfrak{R}_0^+)^*$ is a **slow timed input trace** and $w > t = \max\{ta(s) \mid s \in S_{aD}\}$.

Definition 2.4.9: Adaptive Experiment

An adaptive experiment for a S.B.DEVS model $\mathcal{D}$ is any pair (θ, τ) , where $\theta_i \in (X_{\mathcal{D}} \times \mathfrak{R}_0^+)^*$ is a slow timed relative input sequence of the form $\langle (x_0, t_0), (x_1, t_1), \dots, (x_n, t_n) \rangle$ and $w \in \mathfrak{R}_0^+$, provided that there exists a function

$$f : seq(X_{\mathcal{D}}) \times Y_{\mathcal{D}} \rightarrow X_{\mathcal{D}}$$

$$\text{such that } \forall i = 2..n \bullet \exists y_i \in Y_{\mathcal{D}} \mid x_i = f(\langle x_0, \dots, x_{i-1} \rangle, y_i).$$

That is to say that the value of the i th input event in θ_i depends on all the previous input events and on an (unspecified) output event. The sequence of output events (the output trace) generated by $\mathcal{D}$ in response to θ_i is the result of the experiment.

Since there are many possible definitions for f and in order to clearly identify those functions that are of interest for our test purposes, we define the concept of a *valid adaptive experiment*.

Definition 2.4.10: Valid Adaptive Experiment

An adaptive experiment (θ, τ) for a S.B.DEVS model $\mathcal{D}$ is a valid experiment iff θ_i is slow on $\mathcal{D}$, $w \geq \max\{ta_{\mathcal{D}}(s_{a_i}) \mid s_{a_i} \in S_{a\mathcal{D}}\}$, and the function f of this experiment satisfies the following property:

$$x_1 = f(\langle x_0 \rangle, y_0) \Leftrightarrow y_0 = \lambda_{pass}(s_0, x_0)$$

$$x_i = f(\langle x_0, \dots, x_{i-1} \rangle, y_{i-1}) \Leftrightarrow y_{i-1} = \lambda_{pass}(f(\langle x_0, \dots, x_{i-2} \rangle, y_{i-2}), x_{i-1}) \quad \forall i = 2..n.$$

In other words the value of the i th input event in θ_i depends on all the previous input events and the last output event the model has generated. This is equal to saying that the i th input event depends on all the output events that the S.B.DEVS model has generated so far.

Definition 2.4.11: Distinguishing, Homing, Synchronizing Experiments on S.B.DEVS Models

A valid preset or adaptive experiment (θ, τ) on a S.B.DEVS model $\mathcal{D}$ is distinguishing, homing, or synchronizing iff θ_i is a distinguishing, homing, or synchronizing sequence for $\mathcal{D}$, respectively.

2.4.4 SEQUENCE FINDING

In order to perform a *homing experiment* on a S.B.DEVS model, the procedure described in Kohavi [3] can be utilized in a straightforward way, by considering the passive transition function and the concept of slow timed execution fragments.

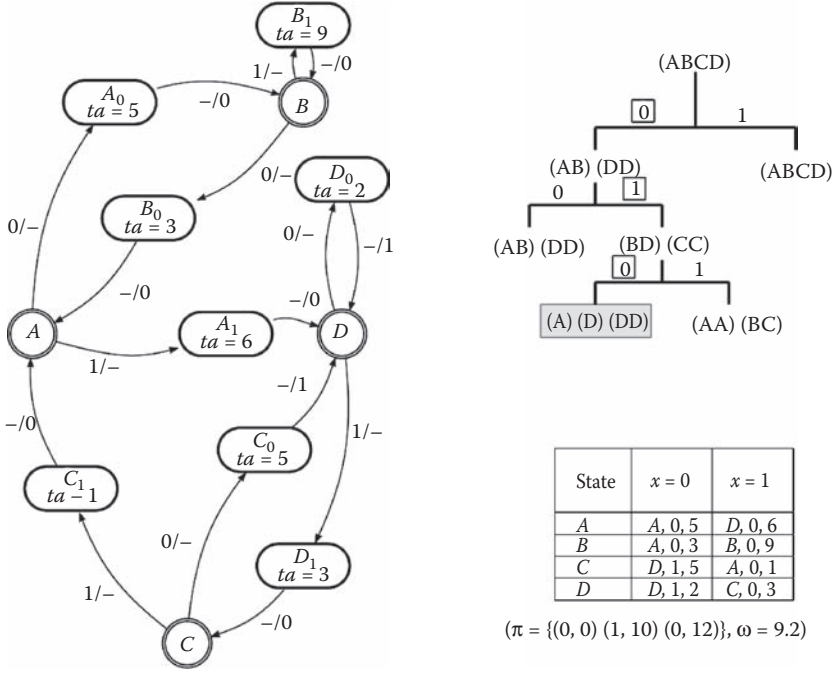


FIGURE 2.7 The S.B.DEVS model (a) with its homing tree, associated transition table, and one possible homing sequence (π) for it (b).

For a minimal S.B.DEVS, a preset homing sequence is built, without timing considerations, using the passive transition function. Then, to each input event x_k of this untimed sequence we provide a **relative** time stamp t_k , such that $t_k > t = \max\{ta(s) \mid s \in S_{a\mathcal{D}}\}$ in order to obtain a slow timed relative input trace. Finally, choosing a waiting time w such that $\tau > t$, the pair (θ, τ) defines a valid preset experiment.

Example

Let us consider the S.B.DEVS represented in Figure 2.7. Using the method presented in [3] we obtain the following untimed homing sequence: $\langle 0, 1, 0 \rangle$. Notice that $t = \max\{ta(s) \mid s \in S_{a\mathcal{D}}\} = 9$; a possible timed relative homing sequence for this S.B.DEVS is then: $\pi = \langle (0, t = 0), (1, t = 10), (0, t = 9.2) \rangle$, and $(\pi, 9.1)$ is a valid experiment for this model.

Theorem 2.4.1

A preset homing sequence, whose length is at most $(n - 1)^2$, exists for every minimal S.B.DEVS model $\mathcal{D}$, where n is the number of passive states in $\mathcal{D}$.

Given the previous considerations on the timed nature of sequences, both *distinguishing* and *synchronizing* sequences can be obtained using the methods and algorithms described in Kohavi [3], and afterward adjoining the time stamps to the events as we did with homing experiments. It is straightforward to prove that all the

properties and theorems in Kohavi [3] are valid for S.B.DEVS models. In particular, the following result is valid.

Theorem 2.4.2

If there exists a synchronizing sequence for a S.B.DEVS model $\mathcal{D}$ that has n passive states, then its length is at most $(n - 1)^2 n/2$.

Proof: See Kohavi [3], p. 458.

The following result sums up the preceding discussion.

Theorem 2.4.3

Let $\pi = \langle x_1 x_2 \dots x_n \rangle$ be either a synchronizing, homing, preset distinguishing or adaptive distinguishing sequence for a Mealy Machine M . Then the sequence $v = \langle (x_1, t_1), (x_2, t_2), \dots, (x_n, t_n) \rangle$ obtained from π is, respectively, either a synchronizing, homing, preset distinguishing or adaptive distinguishing sequence for the S.B.DEVS model $\mathcal{D}$ with the same input-output behavior than M , if $t = \max\{t_{a_D}(p) | p \in S_{a_D}\} < t_i, \forall i = 1..n$.

2.4.5 TESTING PROCEDURE

Now, we need to define a testing procedure to test U.B.DEVS and S.B.DEVS models. For the first type of models the procedure is straightforward since the inputting process is independent of time.

For S.B.DEVS models, the testing procedure can be formally represented by a tester [11–13]. For the sake of constancy, we choose to specify the tester by a DEVS model. The implementation of the S.B.DEVS under testing receives the test input sequence from the tester output tester at specific times. The general scheme of the coupling between a tester and the model under test is given in Figure 2.8.

The tester for a given PX is defined as follows.

Definition 2.4.12: DEVS Tester Model for Preset Experiments

Given a PX (π, τ) where $\pi = \langle (x_0, t_0), (x_1, t_1), \dots, (x_n, t_n) \rangle$, its associated tester is the DEVS model $\mathcal{T} = \langle X_{\mathcal{T}}, Y_{\mathcal{T}}, S_{\mathcal{T}}, \delta_{int_{\mathcal{T}}}, \delta_{ext_{\mathcal{T}}}, \lambda_{\mathcal{T}}, ta_{\mathcal{T}} \rangle$, with:

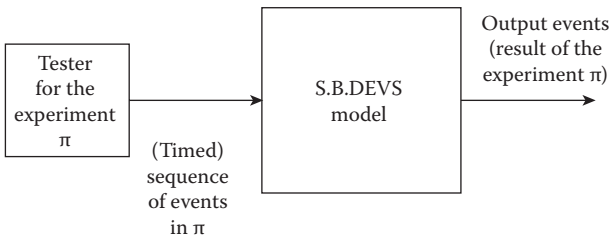


FIGURE 2.8 Coupling scheme of a S.B.DEVS model and a valid tester for it.

- $X_T = \{\text{Reset}\}$ (Event that restores the tester to its initial state)
- $Y_T = x_0, x_1, \dots, x_n$
- $S_T = s_0, s_1, \dots, s_n \cup s_{STOP}$
- $\delta_{\text{ext}_T}(s_i, e, x) = s_0$ (Restores the tester to its initial state)
- $\delta_{\text{int}_T}(s_i) = \begin{cases} s_{i+1} & \text{if } i < n \\ s_{STOP} & \text{if } i = n \vee s_i = s_{STOP} \end{cases}$
- $ta_T(s_i) = \begin{cases} 0 & \text{if } i = 0 \\ t_i & \text{if } i = 1..n \wedge s_i \neq s_{STOP} \\ \infty & \text{if } s_i = s_{STOP} \end{cases}$
- $\lambda_T(s_i) = \begin{cases} x_i & \text{if } s_i \neq s_{STOP} \\ 0 & \text{if } s_i = s_{STOP} \end{cases}$ (This case never happens as $ta_T(s_{STOP}) = \infty$)

Example

The tester that implements the experiment $(\pi = (a, 0) (b, 2) (c, 1), \tau = 4)$ is represented by the state graph in Figure 2.9. If we take into account the fact that after the tester sends the output event x_i , the S.B.DEVS model under test will transition up to the passive state s_{i+1} , then it is assured that the S.B.DEVS model will remain in the same state until the tester issues the output event x_{i+1} in case it exists.

Example: In order to implement a **valid adaptive experiment** for a S.B.DEVS model, the basic idea is to define a tester that represents the decision tree associated with the adaptive experiment, which is an extended version of the one given in (Kohavi, p. 461 [3]) taking into account the elapsed time in the states. In the tester, it is required to have one active state for each output event that the experiment will issue. Additionally, each of these states will receive an external event that represents the response of the model under test. That is, it will be able to receive any of the possible output events that the S.B.DEVS model will generate (in order to do this, the output of the tested model needs to be connected to the tester's input).

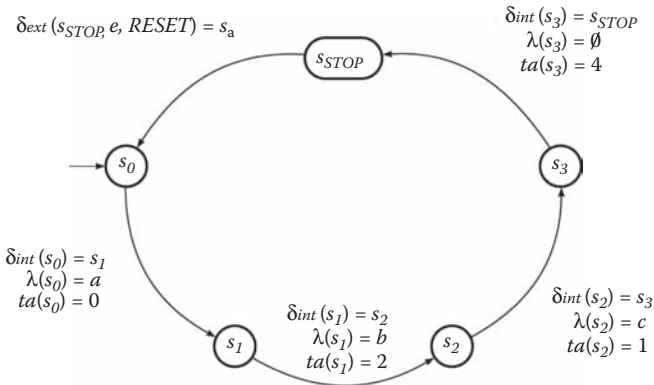


FIGURE 2.9 Tester for the preset experiment (PX) $(\pi = (a, 0) (b, 2) (c, 1), \tau = 4)$.

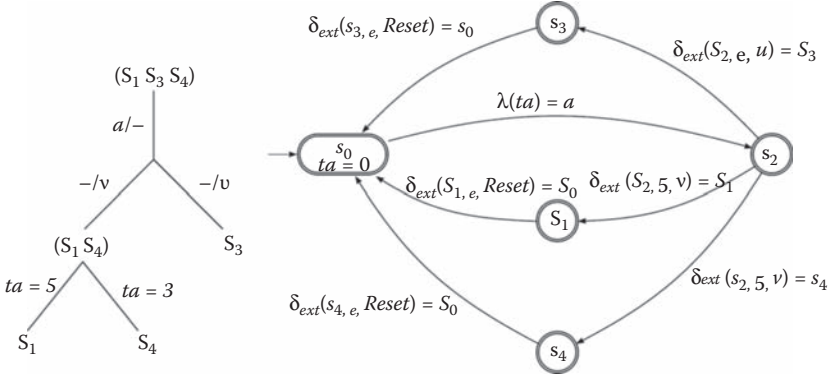


FIGURE 2.10 Sample adaptive distinguishing experiment and corresponding DEVS tester model for the S.B.DEVS model in Figure 2.6.

Depending on the value of the received event, the tester will transition to one of the active states that represent the consequential uncertainties. Finally, all the leaf nodes have to be represented as passive states, which only accept the RESET event in order to reinitialize the experiment. For a complete specification of the tester, all unacceptable behavior of the tested model (that is, if the tested model outputs an unexpected event or an expected event at an unexpected time) must force the tester to transition to an error state. We omit this state in Figure 2.10 so as to make it easier to read, but it should be always added in the final specification. Figure 2.10 shows a concrete example of an adaptive tester.

2.5 CONCLUSIONS AND FUTURE WORK

The definitions and methods presented in this work allow for a proper identification of the states in useful subsets of DEVS models. All the states of a Mealy Machine are steady states; we have introduced U.B.DEVS to allow a clean specification of transitory states in untimed models, since they represent those models that can be properly specified in Mealy Machines, and thus provide a clear way to port Mealy Machines to a timed formalism such as DEVS. Additionally, U.B.DEVS models represent the smallest useful subset of timed models whose states can be properly identified and fault checked. This subset constitutes the first step toward a formal theory of fault detection techniques for the more general DEVS models.

The second step toward this theory consists in expanding the fault detection techniques to the S.B.DEVS subset of models, which is a set of timed models of practical utility, and considerably wider than what initially appears to be, given the fact that in this work we have only considered simple events. As a consequence, S.B.DEVS models might seem to only allow a strict alternance between input and output events. However, these models can be expanded in a straightforward way in order to work with complex compound states without altering this theory and thus allowing for different combinations of input events, waiting times in active states, and the consequential output events.

Aside from the definition of the before mentioned subsets, the importance of formally defining the concepts of distinguishing, homing, and synchronizing sequences lies in the possibility of checking random sequences for a given DEVS model in order to determine whether they belong to any of the three relevant categories. A method to obtain such sequences could then be designed so as to provide a mechanical, automatic means to find relevant sequences of timed input events. The process of obtaining new sequences can also be automated since it is straightforward to find the highest value of the time advance function ta in a model, and then design the proper tester for every possible slow timed input sequence that the model accepts. This ensures the plausibility of automating a stage of the conformance testing procedure.

Future work involves expanding the U.B.DEVS subset, possibly up to clearly stating the broadest subset of DEVS models that can be fault checked with traditional techniques, as well as targeting the problem of fault detection in coupled DEVS models.

REFERENCES

1. Moore, E. *Gedanken Experiments on Sequential Machines*. Princeton, NJ: Princeton University Press, 1956.
2. Lee, D., and Yannakakis, M. Principles and methods of testing finite state machines—A survey, in *Proceedings of the IEEE*. New York: IEEE Press, 1996.
3. Kohavi, Z. *Switching and Finite Automata Theory: Computer Science Series*. Edited by R. W. Hamming and E. A. Feigenbaum. New York: McGraw-Hill Higher Education, 1980.
4. Alur, R., and Dill, D. A theory of timed automata, *Theoretical Computer Science* 126 (1994): 183.
5. Springintveld, J., Vaandrager, F., and D'Argenio, P. R. Testing timed automata, *Theoretical Computer Science* 254 (2001).
6. Dacharry, H. P., and Giambiasi, N. Formal verification with timed automata and DEVS models: A case study, in *ASSE 2005 Simposio Argentino de Ingenieria de Software—34 JAAIO Jornadas Argentinas de Informatica e Investigacion Operativa*, Rosario, Argentina, 2005.
7. Dacharry, H. P., and Giambiasi, N. Discrete event modeling through a multi-formalism approach, from a user-oriented perspective, in *Proceedings of the 2007 Spring Simulation Multiconference*, 207. Vista, CA: Society for Computer Simulation International, 2007.
8. Zeigler, B. P. *Theory of Modeling and Simulation*, 1st ed. Malabar, FL: Krieger Publishing, 1984.
9. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, 2nd ed. New York: Academic Press, 2000.
10. Dacharry, H. P., and Giambiasi, N. DEVS based timed hierarchy of formalisms, in *Proceedings of the International Modeling and Simulation Multiconference 2007 (IMSM07)*. —Vista, CA: The Society for Modeling and Simulation International, 2007.
11. Krichen, M., and Tripakis, S. State identification problems for timed automata, in *The 17th IFIP International Conference on Testing of Communicating Systems (TestCom'05)*. New York: Springer, 2005.
12. Krichen, M., and Tripakis, S. State-Identification Problems for Finite-State Transducers, in *Formal Approaches to Software Testing and Runtime Verification*, 148. Berlin: Springer/Heidelberg, 2006.
13. Tripakis, S., and Yovine, S. Analysis of timed systems using time-abstracting bisimulations, *Formal Methods in System Design* 18 (2001).

3 A Formal Definition of Dynamic Structure Discrete-Flow Components

Fernando Barros

CONTENTS

3.1	Introduction	55
3.2	Preliminaries	56
3.2.1	Hyperreal Numbers	56
3.2.2	Partial States	57
3.2.3	Noninstantaneous Propagation Assumption.....	58
3.2.3.1	Logical Clocks	58
3.2.4	Events and Transitions	58
3.3	Discrete-Flow Systems	59
3.3.1	DFSS Basic Model.....	59
3.3.2	DFSS Basic Component	59
3.3.3	DFSS Network Model.....	61
3.3.4	Executive Component	62
3.3.5	Network Component	63
3.3.6	Component Simulation	67
3.4	Related Work	68
3.4.1	DEVS	68
3.4.2	Dynamic Structure Modeling	69
3.4.3	Simulation Time	71
3.5	Conclusions.....	71
	Acknowledgment	72
	References.....	72

3.1 INTRODUCTION

The representation of discrete-event systems has been subjected to intense research during the last decade. While the field is relatively well understood, some key concepts still lack a rigorous definition leading to undefined semantics. In this chapter we

present new solutions to some of these issues. In particular we focus on the semantics of what is commonly called *simultaneous events*, in the presence of dynamic topologies. We provide a unified definition of time, based on the *set of hyperreals* [9], which is able to describe models and components while accommodating deterministic executions under structural changes. Hyperreals provide a physical interpretation for virtual clocks used in distributed systems [14] and warped simulations [11], offering a unified view for time representation.

Although the term *component* is widely used, a formal definition of discrete simulation component is lacking. The term component is commonly used interchangeably with the concept of *model*. We present a formal definition of these two concepts making their meaning clear.

The Discrete Flow System Specification (DFSS) is a modular formalism able to describe hierarchical event-based systems with time-varying structure. The DFSS support for structural changes ranges from the addition/deletion of components and connections to the representation of mobile components. This formalism is based on the Discrete Event System Specification (DEVS) [24] while introducing new concepts, like topology adaptation, and more precise semantics for simultaneous events. The DFSS introduces the concept of *sequence of influencers* of a component in discrete systems. Influencers supersede the concept of bag of inputs introduced in P-DEVS [7], providing a simpler and more powerful construct for model reuse. DFSS influencers provide a *physical* interpretation for *Null Messages* used in conservative distributed simulations for deadlock recovery [16]. Influencers have also enabled a unifying description of interconnections in hybrid systems based on both discrete events and generalized sampling [5].

3.2 PRELIMINARIES

In this section we establish the basic assumptions used in this chapter. These assumptions play a critical role in defining formalism semantics enabling a deterministic simulation in a large variety of conditions, for example, when several components are scheduled simultaneously at a topology change. We consider that the failure to establish these basic assumptions is responsible to the undefined behavior exhibited by many simulation formalisms.

3.2.1 HYPERREAL NUMBERS

We provide an intuitive definition of hyperreal numbers and the semantics of the addition and comparison operators. We revise first the notation for real intervals since we develop the definition of hyperreals based on the interval notation. The expression $a \geq t$, $t \in \mathbf{R}$ defines the interval $[t, +\infty)$, and $a > t$ defines the interval $(t, +\infty)$. If we denote the left extreme of the interval $(t, +\infty)$ by $t+$ we can rewrite the interval $(t, +\infty)$ as $[t^+, +\infty)$. To simplify the notation used in the next sections we define the infinitesimal number $\varepsilon = t+ - t$. An alternative and canonical definition of infinitesimals in the realm of nonstandard analysis can be found by Goldblatt [9].

Given two numbers $a + m\epsilon$ and $b + n\epsilon$ with $a, b \in \mathbf{R}$ and $m, n \in \mathbf{N}$, we define the add operation by:

$$(a + m\epsilon) + (b + n\epsilon) = (a + b) + (m + n)\epsilon.$$

The minus operation is defined by:

$$(a + m\epsilon) - (b + n\epsilon) = (a - b) + (m - n)\epsilon.$$

The number zero can be written by:

$$0 \equiv 0 \pm 0\epsilon.$$

We define the set of hyperreals reals by:

$$\mathbf{H} = \{r \pm i\epsilon \mid r \in \mathbf{R}, i \in \mathbf{N}_0\} \cup \{\pm\infty\}.$$

The equality operator is defined for two numbers $x, y \in \mathbf{H}$ by:

$$x = y \quad \text{if } x - y = 0.$$

The relational operator $<$ is defined for $r + i\epsilon \in \mathbf{H}$ and 0 by:

$$r + i\epsilon < 0 \quad \text{if } r < 0 \vee (r = 0 \wedge i < 0).$$

The relational operator $<$ is defined for two numbers $x, y \in \mathbf{H}$ by:

$$x < y \quad \text{if } x - y < 0.$$

A number $a + n\epsilon \in \mathbf{H}$ is considered *standard* iff $n = 0$. Given a real number $r \in \mathbf{R}$, we define $\mathbf{H}(r)$ as the set of all nonstandard numbers centered in r by:

$$\mathbf{H}(r) = r \pm i\epsilon, i = 0, 1, 2, \dots$$

3.2.2 PARTIAL STATES

A component can be in a state $s \in S$ with $S = \{(p, e) \mid p \in P, 0 \leq e \leq \omega(p)\}$ where P is the set of partial states (p-states), and $\omega : P \rightarrow \mathbf{H}_0^+$ is the maximum time the component can stay in p-state $p \in P$. We constrain our study to discrete machines where the number of p-state transitions a component can undergo is finite in a finite time interval. We limit our discussion to models that can be represented in digital computers.

Analog computers do not impose these constraints and can provide a continuous update of p-states, but are out of scope in this study.

3.2.3 NONINSTANTANEOUS PROPAGATION ASSUMPTION

We consider the noninstantaneous assumption that describes the time semantics of modeling instances (components) when subjected to a transition.

Assumption. A component performing a transition at time t only changes its p-state at time $t^+ = t + \varepsilon$.

Taking, as an example, a component undergoing one single transition at time $t \in (-\infty, +\infty)$, the component will be in p-state p during the interval $(-\infty, t]$ and in a new p-state p' in the interval $[t^+, +\infty)$ as depicted in Figure 3.1. Noninstantaneous propagation gives an operational definition of causality by defining the time between the cause and its effect as ε . Given the operations on infinitesimals it is possible to define a total order for transitions and to provide deterministic semantics for simultaneous transitions as we show in the next sections.

3.2.3.1 Logical Clocks

Hyperreals provide a physical interpretation for logical clocks commonly used in distributed systems [14]. Considering that processing takes zero time, logical clocks are limited to the set $\mathbf{H}(0)$, and we can identify a clock n with the hyperreal $0 + n\varepsilon$ and to derive the rules to clock advance [8].

3.2.4 EVENTS AND TRANSITIONS

The term event has several meanings in modeling and simulation. It can be used as synonymous of transition (i.e., a component changes its p-state due to the application of a transition). An event has also been used as synonymous of a value that can trigger a transition. In this case it is commonly called an external event. In Barros [6], we have introduced a new way of triggering a transition using the concept of synchronous influencees. In this case, a component that belongs to the set of synchronous influencees of another component that undergoes a transition can also suffer a transition. However, there is no value, or event in the conventional notation, which forces the transition. Given these different definitions, we only will be using the word transition to indicate a change in p-state.

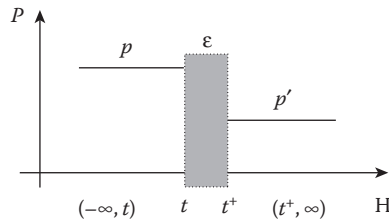


FIGURE 3.1 Noninstantaneous assumption.

3.3 DISCRETE-FLOW SYSTEMS

The DFSS is a formalism aimed to represent piecewise constant partial state systems. This characteristic enables its implementation on digital computers. DFSS supersedes the classical discrete time machines while the representation of discrete events is based on the DEVS [24].

3.3.1 DFSS BASIC MODEL

We consider $\hat{B}$ as the set of names corresponding to basic DFSS models. A DFSS basic model associated with name B is defined by:

$$M_B = (X, Y, P, \omega, s_0, \delta, \lambda), \text{ for } B \in \hat{B},$$

where

X is the set of discrete input flow values

Y is the set of discrete output flow values

P is the set of partial states (p-states)

$\omega : P \rightarrow \mathbf{H}_0^+$ is the time-to-output function

$S = \{(p, e) | p \in P, 0 \leq e \leq \omega(p)\}$ is the state set

$s_0 = (p_0, e_0) \in S$ is the initial state

$\delta : S \times X^\varnothing \rightarrow P$ is the transition function,

where $X^\varnothing = X \cup \{\emptyset\}$

$\emptyset$ represents the null value (absence of value)

$\lambda : P \rightarrow Y$ is the partial discrete output function.

The DFSS components have their behavior ruled by DFSS models and their semantics are presented in the next section.

3.3.2 DFSS BASIC COMPONENT

Models themselves do not provide the semantics of DFSS entities since they do not specify how rules are applied to dynamic entities that evolve over time. To obtain timed semantics we introduce the concept of the DFSS component. A component is a computer-based representation of a dynamic entity that evolves in time domain. Components have their behavior governed by DFSS models but they introduce the rules on how models are interpreted. A basic component associated with name B and with model M_B is defined by:

$$\Xi_B = (< s_m, s >, T, \Delta, \Lambda), \text{ for } B \in \hat{B},$$

where

$M_B = (X, Y, P, \omega, (p_0, e_0), \delta, \lambda)$ is the component model.

The model is not explicitly included in the component definition since we assume that given a name $B \in \hat{B}$, a model M_B and a component Ξ_B can be associated.

We define first $S = \{(p, t) | p \in P, t \in \mathbf{H}\}$ as the component state set and $S^\varnothing = S \cup \{\varnothing\} < s_m, s >$ with $s_m \in S^\varnothing$ and $s \in S^\varnothing$ as the component state with

s_m is the component previous state
 $s = (p, t)$ the component current state,

where

p is the component current p-state
 t is the time of component last transition.

In this description we have used the notation $\langle a, b, \dots \rangle$ to declare variables $a, b, \dots$

These variables are hidden in the sense they can only be accessed by the other members of the structure. This notation builds a bridge to object-oriented programming where there is a distinction between public and private members and where variables are explicitly expressed. In the usual notation, however, structures only declare invariant public values (i.e., values that can be accessed by external entities).

$T: \{\varnothing\} \rightarrow \mathbf{H}$ is the time limit for the next transition defined by:

$$T = t + \omega(p), \text{ where } (p, t) \in s \text{ is the current state.}$$

$\Delta: \mathbf{H} \times X$ is the component transition action defined by:

$$\begin{aligned} \Delta(t, x) &\triangleq \\ &s_m \leftarrow s, \text{ assuming } s = (p, \tau) \\ &s \leftarrow (\delta((p, t - \tau), x), t + \varepsilon), \end{aligned}$$

where by the noninstantaneous assumption, the component changes only at $t + \varepsilon$.

The $\Lambda: \mathbf{H} \rightarrow Y^\varnothing$, with $Y^\varnothing = Y \cup \{\varnothing\}$ is the component output function defined by:

$$\Lambda(t) = \begin{cases} \lambda(p) & \text{if } t - \tau = \omega(p) \\ \varnothing & \text{otherwise} \end{cases}$$

where

$$(p, \tau) = \begin{cases} (p_1, \tau_1) & \text{if } t \geq \tau_1 \\ (p_2, \tau_2) & \text{if } t < \tau_1 \end{cases}$$

with $(p_1, \tau_1) = s$ and $(p_2, \tau_2) = s_m$, restricted to $t \geq \tau_2$.

This last constraint is necessary since only the last state is stored. The component keeps both the last state s_m and the current state s . The last state is required to allow a nondeterministic invocation of the transition action. If two or more transitions occur at the same time, the order of transition invocation becomes relevant since it changes the values received by other components. Since a component keeps the past state, its output value can still be obtained even if the component has already changed its state and advanced its current time.

3.3.3 DFSS NETWORK MODEL

The DFSS network models are a composition of DFSS models (basic or DFSS network models). Let $\hat{N}$ be the set of names corresponding to DFSS network models, with $\hat{N} \cap \hat{B} = \emptyset$. Formally, a DFSS network model associated with name N is defined by:

$$M_N = (X, Y, \eta), \text{ for } N \in \hat{N},$$

where

N is the network name

X is the set of network discrete input flows

Y is the set of network discrete output flows

$\eta \in \hat{\eta}$ is the name of the dynamic structure network executive,

with

$\eta \in \hat{\eta}$ representing the set of all names associated with DFSS executive models, constrained to $\hat{\eta} \cap \hat{B} = \hat{\eta} \cap \hat{N} = \emptyset$.

Executives are uniquely assigned to network models; that is,

$$\forall_{i,j \in \hat{N}, i \neq j} \eta_i \neq \eta_j \text{ with } M_k = (X_k, Y_k, \eta_k), \forall_{k \in \hat{N}}.$$

The model of the executive is a modified DFSS model, defined by:

$$M_\eta = (X_\eta, Y_\eta, P, \omega, s_0, \delta, \lambda, \hat{\Sigma}, \gamma), \text{ for } \eta \in \hat{\eta},$$

where

$\hat{\Sigma}$ is the set of network structures

$\gamma: P \rightarrow \hat{\Sigma}$ is the structure function.

The network structure $\Sigma_\alpha \in \hat{\Sigma}$ corresponding to the p-state $p_\alpha \in P$ is given by:

$$\Sigma_\alpha = \gamma(p_\alpha) = (C_\alpha, \{I_{i,\alpha}\} \cup \{I_{\eta,\alpha}, I_{N,\alpha}\}, \{E_{i,\alpha}\} \cup \{E_{\eta,\alpha}, E_{N,\alpha}\}, F_{i,\alpha} \cup \{F_{\eta,\alpha}, F_{N,\alpha}\}),$$

where

C_α is the set of names associated with the executive state p_α
 for all $i \in C_\alpha \cup \{\eta\}$
 $I_{i,\alpha}$ is the sequence of asynchronous influencers of i
 $E_{i,\alpha}$ is the set of the synchronous influencees of i
 $F_{i,\alpha}$ Fit(X) is the input function of i
 $I_{N,\alpha}$ is the sequence of network influencers
 $E_{N,\alpha}$ is the set of synchronous network influencees
 $F_{N,\alpha}$ is the network output function.

For all $i \in C_\alpha$

$M_i = (X_i, Y_i, P_i, \omega_i, s_{0,i}, \delta_i, \lambda_i)$ if $i \in \hat{B}$
 $M_i = (X_i, Y_i, \eta_i)$ if $i \in \hat{N}$.

Variables are subjected to the following constraints for every $p_\alpha \in P_\alpha$

$$N \notin C_\alpha, N \notin I_{N,\alpha}, \eta \notin C_\alpha$$

$$N \notin E_{i,\alpha} \text{ for all } i \in C_\alpha \cup \{\eta, N\}$$

$$F_{N,\alpha} : \times_{k \in I_{N,\alpha}} Y_k \rightarrow Y^\emptyset$$

$$F_{i,\alpha} : \times_{k \in I_{i,\alpha}} V_k \rightarrow X_i^\emptyset,$$

where

$$V_k = \begin{cases} Y_k^\emptyset & \text{if } k \neq N \\ X^\emptyset & \text{if } k = N \end{cases}$$

$$F_{N,\alpha}(\emptyset, \emptyset, \dots) = \emptyset$$

$$F_{i,\alpha}(\emptyset, \emptyset, \dots) = \emptyset.$$

These two last constraints are a characteristic of discrete systems and impose that non-null values cannot be created from a sequence composed exclusively by null values.

3.3.4 EXECUTIVE COMPONENT

The DFSS executive components are associated with executive models and defined by:

$$\Xi_\eta = (< s_m, s >, T, \Delta, \Lambda, \Gamma), \text{ for } \eta \in \hat{\eta},$$

where

$M_\eta = (X_\eta, Y_\eta, P, \omega, s_0, \delta, \lambda, \hat{\Sigma}, \gamma)$, is the model of the executive

$\Gamma : \mathbf{H} \rightarrow \hat{\Sigma}$, is the executive component structure function defined by:

$$\Gamma(\tau) = \gamma(p)$$

with

$$(p, t) = \begin{cases} (p_1, t_1) = s & \text{if } \tau \geq t_1 \\ (p_2, t_2) = s_m & \text{if } \tau < t_1 \end{cases}$$

restricted to $\tau \geq t_2$.

When computing the structure at time τ , the structure function uses the past state when τ is smaller than the time of the last transition t_1 . If several components, including the executive, are scheduled to change their state at the same time, the order by which transitions occur becomes irrelevant, since if the executive state is changed first, its previous state will be used to compute the current network structure.

3.3.5 NETWORK COMPONENT

A network is composed of the executive and a set of other components. These components and their interconnections can change accordingly to the current state of the executive. Components can be basic or other DFSS networks, making it possible to define networks hierarchically. A DFSS network component is defined by:

$$\Xi_N = (T, \Delta, \Lambda), \text{ for } N \in \hat{N}$$

with $M_N = (X, Y, \eta)$,

$$\text{and } M_\eta = (X_\eta, Y_\eta, P, \omega, s_0, \delta, \lambda, \hat{\Sigma}, \gamma),$$

and where

T: $\{\emptyset\} \rightarrow \mathbf{H}$ is the maximum time allowed in the current state defined by:

$$T = \min\{\Xi_k.T \mid k \in C \cup \{\eta\}\}$$

$$\text{with } (C, \{I_i\}, \{E_i\}, \{F_i\}) = \Xi_\eta.\Gamma(t_\eta),$$

and $s_\eta = (p_\eta, t_\eta)$ is the current executive state.

The $\Delta : \mathbf{H} \times X$ is the component transition action defined by:

$$\begin{aligned} \Delta(t, x) &\triangleq \\ &H \leftarrow \{k | k \in C \cup \{\eta_N\}, x_k \neq \emptyset \vee t = \Xi_k.T\} \\ &L \rightarrow \text{transitive-closure}(H \cup E_N) \\ &\textbf{for all } k \in L \textbf{ do } \Xi_k.(t, x_k) \\ &\textbf{for all } c \in C' \setminus C \textbf{ do } \text{create-component}(c, t + \varepsilon) \\ &\textbf{for all } c \in C' \setminus C \textbf{ do } \text{create-component}(c) \end{aligned}$$

with

$$\begin{aligned} x_k &= F_k(\times_{i \in I_k} v_i) \\ v_i &= \begin{cases} \Xi_i.\Lambda(t) & \text{if } i \neq N \\ x & \text{if } i = N \end{cases} \\ (C, \{I_i\}, \{E_i\}, \{F_i\}) &= \Xi_\eta.\Gamma(t) \text{ and} \\ (C', \{I'_i\}, \{E'_i\}, \{F'_i\}) &= \Xi_\eta.\Gamma(t + \varepsilon). \end{aligned}$$

The $\Lambda : \mathbf{H} \rightarrow Y$ is the network component output function defined by:

$$\begin{aligned} \Lambda(t) &= F_N(\times_{i \in I_N} \Xi_i.(t)) \\ \text{with } (C, \{I_i\}, \{E_i\}, \{F_i\}) &= \Xi_\eta.\Gamma(t). \end{aligned}$$

The transitive closure is defined by:

$$\begin{aligned} \text{transitive-closure}(a) &= \text{transitive closure}(a, \{\}, \{\}) \\ \text{transitive-closure}(\{\}, \_, d) &= d \\ \text{transitive-closure}(\{h|r\}, a, d) &= \\ &\quad \text{transitive-closure}(r, a, d) \text{ if } h \in a \\ \text{transitive-closure}(\{h|r\}, a, d) &= \\ &\quad \text{transitive-closure}(r \cup E_h \setminus (a \cup \{h\}), a \cup \{h\}, d \cup E_h) \\ &\quad \text{if } h \notin a. \end{aligned}$$

The set of components that undergo a transition action is composed by the set H that includes the components that have a non-null discrete event input flow plus the set of components whose maximum time in the current state has arrived. Set H is further extended with the components synchronously linked to the components in this set [6].

Thus, for example, if component c is part of the synchronous influencees of b and b is part of the influencees of a , then a change in component a will cause a change in b and also in c by the transitive closure.

The creation of a network component is achieved by the action create-network-component: $\hat{N} \times \mathbf{H}$ defined by:

create-network-component $(N, t) \triangleq$

$\Xi_N \leftarrow (T, \Delta, \Lambda)$

$\Xi_\eta \leftarrow (<\emptyset, (p_0, t - e_0 + \epsilon)>, T, \Delta, \Lambda)$

for all $c \in \Xi_\eta \cdot \Gamma(t)$ **do** create-component(c, t)

with $M_N = (X, Y, \eta)$

and $M_\eta = (X_\eta, Y_\eta, P, \omega, s_0, \delta, \lambda, \hat{\Sigma}, \gamma)$.

The creation of components is accomplished by action create-component: $(\hat{B} \cup \hat{N}) \times \mathbf{H}$ defined by:

create-component $(n, t) \triangleq$

if $\Xi_n \neq \emptyset$ **then** error

if $n \in \hat{B}$ **then**

$\Xi_n \leftarrow (<\emptyset, (p_0, t - e_0 + \epsilon)>, T, \Delta, \Lambda)$

with $M_n = (X, Y, P, \omega, (p_0, e_0), \delta, \lambda)$

else create-network-component(n, t).

The destruction of components is made by action destroy-component: $(\hat{B} \cup \hat{N})$ defined by:

destroy-component $(n) \triangleq$

if $n \in \hat{B}$ **then** $\Xi_n \leftarrow \emptyset$

else destroy-network-component(n).

The destruction of network components is made by the action destroy-network-component: $\hat{N}$ defined by:

destroy-network-component $(n) \triangleq$

for all $d \in \Xi_\eta \cdot \Gamma(t)$ **do** destroy-component(d)

$\Xi_\eta \leftarrow \emptyset$

$\Xi_N \leftarrow \emptyset$

with $M_n = (X, Y, \eta)$.

Example

To illustrate the semantics of DFSS networks, we consider the DFSS component represented in Figure 3.2 that starts at the topology represented in Figure 3.2a and it can change to the topology of Figure 3.2b. This network has no input and outputs. We consider that the set of synchronous influencees is empty for all components. For simplicity, we omit the specifications of the input functions and the models of components. We consider that network components can be either in transitory or in passive p-states. These constraints permit analyzing a network exhibiting zero-time delays, simultaneous transitions while undergoing structural changes. Topology T_0 is associated with executive p-state p_0 and it is given by:

$$\Sigma_0 = \gamma(p_0) = (C_0, \{I_{i,0}\} \cup \{I_{\eta,0}, I_{N,0}\}, \{E_{i,0}\} \cup \{E_{\eta,0}, E_{N,0}\}, F_{i,0} \cup \{F_{\eta,0}, F_{N,0}\}),$$

where

$$\begin{aligned} C_0 &= \{A, B\} \\ I_{A,0} &= \{B, \eta\} \\ I_{B,0} &= \{A\} \\ I_{\eta,0} &= \{B\}. \end{aligned}$$

Topology T_1 is associated with executive p-state p_1 and it is given by:

$$\Sigma_1 = \gamma(p_1) = (C_1, \{I_{i,1}\} \cup \{I_{\eta,1}, I_{N,1}\}, \{E_{i,1}\} \cup \{E_{\eta,1}, E_{N,1}\}, F_{i,1} \cup \{F_{\eta,1}, F_{N,1}\}),$$

where

$$\begin{aligned} C_1 &= \{A, B\} \\ I_{A,1} &= \langle B \rangle \\ I_{B,1} &= \langle \eta \rangle \\ I_{\eta,1} &= \langle A \rangle. \end{aligned}$$

The behavior of the network components A , B , and η is depicted in Figure 3.3 where component p-state, discriminating only transitory and passive p-states, is represented. Since only zero time delays are involved, the time diagram is limited to $H(0)$. At time 0 the executive η exhibits a non-null discrete output flow and it forces a transition on component A at the same time, since we assume no delay in

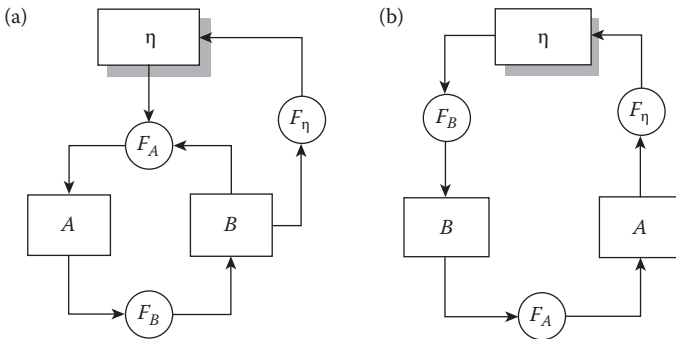


FIGURE 3.2 Dynamic structure topology.

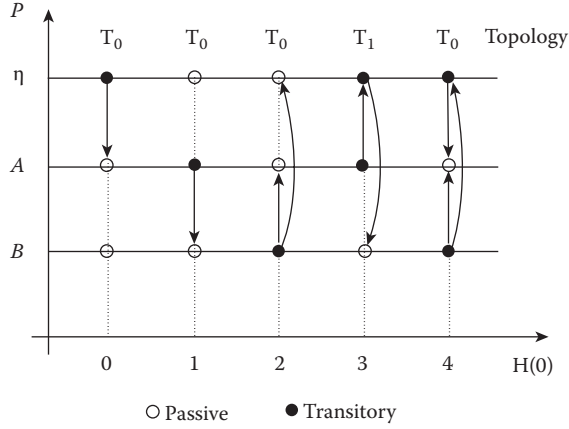


FIGURE 3.3 Network component behavior.

interconnection communication. Component A changes its p-state at time ϵ due to the noninstantaneous propagation assumption. Since $A \in I_B$, A induces a transition in B at time ϵ . Component B changes its p-state at time 2ϵ . At this time the output of component B is non-null and this forces transitions at components A and η . These two components undergo a transition at time 2ϵ and a new p-state is set at time 3ϵ . At this time the executive modifies the network structure to topology T_1 represented in Figure 3.2b. In this topology, A ceases to be an influencer of B and becomes an influencer of η . Component η undergoes another transition and at time 4ϵ the network returns to topology T_0 depicted in Figure 3.2a.

The noninstantaneous assumption was introduced in Barros [4] since the alternative interpretation—considering that a component changes its p-state at transition time—would lead to undefined behavior at structural modifications. Taking the transition at time 2ϵ , we verify that the executive is triggered since $B \in I_\eta$. If executive p-state was changed at 2ϵ then $B \notin I_\eta$ and the transition would not have occurred, leading to a contradiction.

As shown in this example, the noninstantaneous propagation provides a simple but powerful mechanism to guarantee determinism in all situations; namely, loops of zero-time delays and structural changes.

3.3.6 COMPONENT SIMULATION

Since basic and networks have the same transition action signature, the simulation of basic and network components can be made by the same action. The simulation of components is done by the action:

$$\text{simulate} : (\hat{B} \cup \hat{N}) \times \mathbf{R},$$

defined by:

$$\begin{aligned} \text{simulate}(n, t) &\triangleq \\ \text{create-component}(n, t) \end{aligned}$$

while $\Xi_n . T \neq \infty$ **do** $\Xi_n . \Delta (\Xi_n . T, \emptyset)$
 destroy-component(n).

The simulation creates the component Ξ_n associated with name n and executes component transition action while the component time next is finite. At the end of the simulation the component is destroyed.

3.4 RELATED WORK

3.4.1 DEVS

The DEVS semantics in the presence of simultaneous transitions was defined by the select function that chooses just one of the components to undergo the transition [24]. The behavior in the presence of both an external value and an internal transition where formally defined in Parallel-DEVS (P-DEVS) that has introduced the *confluent transition* to be triggered when these conditions occur simultaneously [7]. P-DEVS formalism requires components to receive bags of values to handle all the inputs arriving at the same time. The combination of *events* was first introduced in Yaddes [19] and later used in the High Level Architecture [22].

The DFSS supports a simpler interface since it applies the input function to all the input values before sending the result to the receiving component. It is thus possible to compose arbitrary components without the creation of new adaptor components. If, for example, several components produce pairs $\langle port, value \rangle$ to a component D , and if D is designed to receive a list of values for each port, we can develop an input function that performs an operation on the set of inputs. This function can produce the single list required by D . We consider the network of Figure 3.4a where component D receives values from components A , B , and C . Input function $F_D(\langle out, v_A \rangle, \langle out, v_B \rangle, \langle out, v_C \rangle) = \langle in, \langle v_A, v_B, v_C \rangle \rangle$ can perform the required adaptation between these components. On the contrary R-DEVS would require the creation of an additional component T , Figure 3.4b, to make the adaptation possible, reducing thus the possibility of component reuse. In fact, DEVS output-to-input functions offer little help here since these functions can only handle the values separately being unable to perform their combination. In this example DFSS input function replaces DEVS output-to-input functions $Z_{A,T}$, $Z_{B,T}$, $Z_{C,T}$, $Z_{T,D}$, and avoids the creation of the new component T . DFSS enables simpler and reusable components.

The sequence of asynchronous influencers can explain the concept of *Null Messages* used in conservative distributed simulations [16] for deadlock resolution. In fact, DFSS input functions require the values to form *all* influencer components, including those values that are *Null* ($\emptyset$). Influencers provide a *physical* interpretation to Null Messages that were missing in Misra [16].

It looks like the set of influencees introduced by DEVS is a useful optimization for discrete systems, since the probability of simultaneous transitions is commonly very low. However, this optimization has prevented revealing the nature of the relationships among components.

DFSS components combine the concepts of closure under coupling and abstract simulators eliminating some of the redundancy existing between these two concepts.

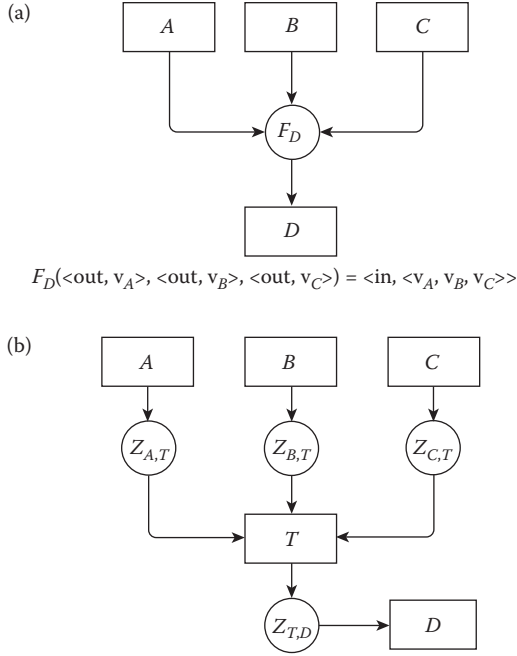


FIGURE 3.4 (a) DFSS versus (b) DEVS.

This combination makes possible the introduction of hyperreals for time representation since DFSS is not constrained by the absence of an absolute time reference commonly required for establishing closure under coupling [24]. Further research is required to investigate if the *Noninstantaneous Assumption* and hyperreals can be introduced in DEVS classical closure under coupling.

3.4.2 DYNAMIC STRUCTURE MODELING

The semantics of the discrete-event dynamic structure systems was first established in Barros [2]. The representation of dynamic structure has been subjected to research work and limited support has been recently introduced in some tools.

In R-Charon [13], a set of operators are introduced to support dynamic topologies. All components can perform arbitrary changes in the structure being that the structure is stored in global variables. This approach is nonmodular and brings the issue of nondeterminism, since it becomes critical that the order for structural operators are executed if two or more components want to change the topology at the same time. The use of global variables has long been regarded as a poor programming style since it can hardly scale and jeopardizes component interoperability, mobility, and reuse.

Ptolemy [15] relies on a similar construct to the DFSS executive called the *Director*, but with less structural power since no modular communication can be established between the Director and the other components.

The issue of nonpredictability in structural changes has also been raised in Dyn-DEVS [21] where the network structure is inferred by a structure function from the state of all components. This work intended to extend the structure function introduced in Barros [3] to allow topology modifications to be made by every component currently located in the network. Although this extension looks like an improvement over the executive concepts, it introduces very negative side effects. Since each component is not aware of the structure function nor the state of the other components, the resultant structure emerges in the individual component perspective in an unpredictable manner from the global state. Dyn-DEVS is thus a nondeterministic formalism from the topology perspective. Dyn-DEVS is also a nonmodular formalism since the structure function maps the state of all components into structure. Dyn-DEVS structure function becomes nonmodular. On the contrary, DFSS components are fully modular since they can only be accessed by their interface, promoting reuse and component.

Nonmodularity through the use of object-oriented message passing was employed by Hu, Zeigler, and Mittal [10]. In this work components can communicate with the coupled component (CC) by messages. Unfortunately, messages are not compatible with the DEVS type of communication since there is no guarantee that all messages sent at the same time will be grouped and received together at the same time. This system is thus incompatible with R-DEVS. The noninstantaneous propagation assumption does not hold in this work. This approach becomes nondeterministic since there is no guarantee of what messages are handled first by the CC, making the structure unpredictable if two or more messages are sent at the same time to the CC. This work mixes discrete flow messages devoted to component-to-CC communication with object-oriented messages for component-to-competent communication and has no apparent benefits and many limitations. This implementation closely follows the High Level Architecture (HLA) [22], which also distinguishes federate-to-federate communication with federate-to-RTI communication. Additionally, the coupled component used by Hu, Zeigler, and Mittal [10], similar to the HLA, supports a fixed set of messages reducing the flexibility of the approach.

AnyLogic [23], relies on a set of calls to make changes to component topology. This approach is similar to Hu, Zeigler, and Mittal [10] and the U.S. Modeling and Simulation Office[22] by exhibiting the same limitations.

The work in Pawletta and Pawletta [18] provides an alternative representation for structural changes. Coupled components are considered to have their own behavior and thus they should be able to change their structure. However, this approach is not efficient since the coupled component needs to analyze all messages to check their final destination. This problem arises when the formalism does not make a distinction between inputs to the coupled component from the inputs to its inner components. The same limitation applies to output values.

An alternative approach close to the DFSS dynamic structure models has been developed [1]. In this work the executive was moved from the coupled model and is located in an external coupled model composed solely by the dynamic structure coupled model and the executive. Although the rationale seems to be the preservation of the original DEVS formalism, we foresee no benefits in this approach since the executive must break hierarchical barriers to control the composition of a network it

does not belong to. Modeling becomes more complex since networks are not homogeneous with the introduction of a new type of coupled model and the representation of dynamic structure topologies requiring an additional hierarchical level. Additionally, the claim that the original DEVS is preserved does not hold, since the state of the external executive needs to be mapped into the network structure, a feature not supported by the classical DEVS.

The use of special values to communicate with the coupled models was advocated by Kim and Kim [12]. This approach lacks the ability to support the communication between the coupled model and the other models reducing the expensiveness of the dynamic structure formalism.

The problem with these solutions is their inability to obey all basic assumptions and principles required to describe dynamic structure simulation models. Although it seems easy to develop new formalisms for representing time-varying topologies, this easiness is mostly superficial since it comes at the expense of relaxing some key constraints like modularity or determinism. However, a formalism that does not violate any constraint seems not an easy task and we conjecture if there can exist alternative formalisms that obey the *same* principles.

In the face of the large number of formalisms developed in the last decade we found that DFSS provides the most general representation of dynamic structure models, and under the assumptions of the last paragraph, it is currently the only formalism that obeys *all* constraints imposed by a general, deterministic, hierarchical, and modular simulation framework.

3.4.3 SIMULATION TIME

The noninstantaneous assumption was formulated in Barros [4]. The state of a component at the transition time was left undefined in previous work [7,24]. Variations of logical clocks have been used in modeling and simulation for dealing with virtual time in optimistic simulators. This work, however, poses a different question since it is incompatible with the semantics defined here of infinitesimals. In this approach, a tag n is reset to zero when times advances by a number larger than zero [20]; that is, $(a + n\epsilon) + k = a + k$, where $(a + n\epsilon) \in \mathbf{H}$, $k \in \mathbf{R}$ and $k > 0$, being inconsistent with our definition. The same basic flaw, under the hyperreal interpretation, can be found in Nutaro [17].

3.5 CONCLUSIONS

We have developed a formal definition of DFSS models and components. DFSS semantics are based on the concept of the component of a construct that combines the information commonly split in abstract simulators and closure under coupling. DFSS extends time based to the set of hyperreal numbers that enable a unified and deterministic treatment of simultaneous transitions. DFSS supports all key features of a full-fledged framework for representing dynamic structure models, namely, modularity, determinism, and bidirectional communication between models and the structure representing construct, implemented in DFSS by the executive.

ACKNOWLEDGMENT

This work was partially funded by the Portuguese Science and Technology Foundation under project POSI/SRI/41601/2001.

REFERENCES

1. Baati, L. Simulators for DEVS models with dynamic structure. In *International Modeling and Simulation Multiconference*, 29–34, 2007.
2. Barros, F. Dynamic structure discrete event specification: A new formalism for dynamic structure modeling and simulation. In *Winter Simulation Conference*, 781–85, 1995.
3. Barros, F. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation* 7, no. 4 (1997): 502–15.
4. Barros, F. Handling simultaneous events in dynamic structure models. In *SPIE Proceedings: Enabling Methodologies for Simulation*, Vol. 3369, 355–63, 1998.
5. Barros, F. Modeling and simulation of dynamic structure heterogeneous flow systems. *SIMULATION: Transactions on the SCS* 78, no. 1 (2002): 18–27.
6. Barros, F. Representing synchronous variable stepsize integration algorithms in the HFSS formalism. In *Conceptual Modeling and Simulation Conference*, 77–82, 2005.
7. Chow, A. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *SCS Transactions on Simulation* 13, no. 2 (1996): 55–102.
8. Coulouris, G., J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Reading, MA: Addison-Wesley, 2005.
9. Goldblatt, R. *Lectures on the Hyperreals: An Introduction to Nonstandard Analysis*. New York: Springer, 1998.
10. Hu, X., B. P. Zeigler, and S. Mittal. Variable structure in DEVS component-based modeling and simulation. *SIMULATION: Transactions of the SCS* 81, no. 2 (2005): 91–102.
11. Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, no. 3 (1985): 404–25.
12. Kim, J., and T. Kim. Framework for modeling/simulation of mobile agent systems. In *AI, Simulation and Planning in High Autonomy Systems*, 53–59, 2000.
13. Kratz, F., O. Sokolsky, G. Pappas, and I. Lee. R-Charon: A modeling language for reconfigurable hybrid systems. In *Hybrid Systems Computation and Control*, Vol. 3972 of *LNCS*, 392–406, 2006.
14. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, no. 7 (1978): 558–65.
15. Lee, E., and H. Zheng. Operational semantics of hybrid systems. In *Hybrid Systems Computation and Control*, Vol. 3414 of *LNCS*, 392–406, 2005.
16. Misra, J. Distributed discrete-event simulation. *ACM Computing Surveys* 18 (1986): 39–65.
17. Nutaro, J. A unified view of time and causality and its application to distributed simulation. In *Summer Simulation Conference*, 419–25, 2003.
18. Pawletta, T., and S. Pawletta. A DEVS-based simulation approach for structure variable hybrid systems using high accuracy integration methods. In *Conceptual Modeling and Simulation Conference*, 368–73, 2004.
19. Preiss, B. The Yaddes distributed discrete event simulation specification language and execution environments. In *SCS Multiconference on Distributed Simulation*, 139–44, 1989.
20. Rönnegren, R., and M. Liljenstam. On event ordering in parallel discrete event simulation. In *Distributed Simulation and Real-Time Applications*, 252–58, 1997.

21. Uhrmacher, A. Dynamic structures in modeling and simulation: A reflective approach. *ACM Transactions on Modeling and Computer Simulation* 11, no. 2 (2001): 206–32.
22. U.S. Modeling and Simulation Office. *High Level Architecture Run Time Infrastructure*, 2001.
23. XJ Technologies. *AnyLogic: Users Manual*, 2005.
24. Zeigler, B. P. *Theory of Modelling and Simulation*. New York: Wiley, 1976.

4 Continuous System Simulation and Control

*Ernesto Kofman, François E. Cellier,
and Gustavo Migoni*

CONTENTS

4.1	Introduction	75
4.2	Quantized State Systems	76
4.2.1	Introductory Example	76
4.2.2	Quantized Systems and DEVS	78
4.2.3	Hysteresis and Quantized State Systems	81
4.2.4	Accuracy of QSS Approximations	84
4.2.5	Theoretical Properties of QSS Methods	86
4.2.6	Higher-Order QSS Methods	88
4.2.7	Implementation of the QSS Methods	90
4.3	Discontinuous, Stiff, and Marginally Stable Systems	92
4.3.1	Discontinuity Handling	93
4.3.2	Stiff Systems and Backward QSS	94
4.3.3	Marginally Stable Systems and Centered QSS	99
4.4	QSS Methods and Digital Control	101
4.5	Conclusions	105
	Acknowledgment	105
	References	105

4.1 INTRODUCTION

Continuous dynamic systems, usually described by sets of ordinary differential equations (ODEs) or by sets of partial differential equations (PDEs) or sets of differential and algebraic equations (DAEs) or any combination thereof, change their state infinitely often within any finite time window.

Therefore, when numerically simulating such a system on a digital computer, we cannot hope to ever determine the solution exactly. The solution must be approximated in some way or other, as no digital computer is capable of performing infinitely many operations within finite time.

Traditionally, numerical ODE solvers convert the ODEs to difference equations, thereby converting the former continuous-time system to an “equivalent” discrete-time system, such that the solution of the discrete-time system at the

sampling points approximates that of the original continuous-time problem at the same time instants.

Classical numerical ODE solvers thus work with time-slicing. They discretize the time axis, whereas they keep the values of the state variables continuous (i.e., real-valued). They answer the following question:

Given the state and state derivative at the current time and at a set of previous sampling time instants, what will be the value of the state at the next sampling instant?

However, this is not the only way how a continuous-time system can be approximated.

In this chapter, we shall introduce a different class of numerical ODE solvers that discretize the state variables, while keeping the time variable continuous. They answer the following question:

Given that the state at the current time has a value of x_k , the k th discretization level of that variable, what is the earliest time at which the state will assume a value of either x_{k+1} or x_{k-1} ?

We shall demonstrate that this approximation does not lead to an equivalent discrete-time system, but rather to an equivalent discrete-event system, and that the discretization can be conveniently described using the discrete event system specification (DEVS) formalism. We shall discuss the numerical stability and approximation accuracy properties of this new class of numerical ODE solvers that have been coined Quantized State System (QSS) simulators. We shall show that the QSS simulators exhibit some striking properties that make them superior to classical ODE solvers in dealing with important classes of engineering systems, such as the simulation of systems with heavy and frequent discontinuities. Finally, QSS simulators represent an important application of DEVS methodology, as they allow, for the first time, to deal with mixed continuous and discrete systems in a fully unified framework.

4.2 QUANTIZED STATE SYSTEMS

In this section, we shall introduce the family of QSS methods.

4.2.1 INTRODUCTORY EXAMPLE

The following equations describe a second-order continuous system that is usually being referred to as a harmonic oscillator:

$$\begin{aligned}\dot{x}_{a_1}(t) &= x_{a_2}(t) \\ \dot{x}_{a_2}(t) &= -x_{a_1}(t).\end{aligned}\tag{4.1}$$

If we know the set of initial states, $x_{a_1}(t_0)$ and $x_{a_2}(t_0)$, we can easily find the analytical solution, which takes the form $x_{a_i}(t) = c_i \sin(t) + d_i \cos(t)$ for certain constants c_i and d_i .

Obtaining the analytical solution is possible because the problem is linear, but in most cases we must make use of numerical integration methods to get our problem solved. For instance, Euler's method approximates the system Equation 4.1 as:

$$\begin{aligned} x_1(k+1) &= x_1(k) + hx_2(k) \\ x_2(k+1) &= x_2(k) - hx_1(k), \end{aligned} \quad (4.2)$$

where h is the time step size. System Equation 4.2 takes the form of a set of difference equations (i.e., a discrete-time system). From a system theoretical point of view, we can think of Euler's method as a technique that approximates a continuous time system by a discrete-time system. This observation holds in fact for all conventional numerical methods. Let us see what happens if we modify Equation 4.1 as follows:

$$\begin{aligned} \dot{x}_1(t) &= \text{floor}[x_2(t)] \triangleq q_2(t) \\ \dot{x}_2(t) &= -\text{floor}[x_1(t)] \triangleq -q_1(t) \end{aligned} \quad (4.3)$$

where $\text{floor}(x_i)$ rounds x_i to the nearest integer value below.

This new system, although being nonlinear and discontinuous, can be easily solved. Consider, for instance, the initial states $x_1(0) = 4.5$, $x_2(0) = 0.5$. First notice that $q_1(0) = 4$ and $q_2(0) = 0$ and that these values remain unchanged until x_1 or x_2 cross through the nearest integer value. Consequently, we have $\dot{x}_1(0) = 0$ and $\dot{x}_2(0) = -4$ meaning that x_1 is constant and x_2 decreases with a constant slope equal to -4 . After $t_1 = 0.5/4 = 0.125$ units of time, x_2 crosses through 0 and q_2 assumes a value of -1 . Thus, the slope of x_1 changes to $\dot{x}_1(t_1^+) = -1$. The situation changes again when x_2 reaches -1 at time $t_2 = t_1 + 1/4$. At that moment, we have $x_1(t_2) = 4.5 - 1/4 = 4.25$ and $\dot{x}_1(t_2^+) = -2$.

The next change occurs when x_1 reaches 4 at time $t_3 = t_2 + 0.25/2$. Then, $q_1(t_3^+) = 3$ and the slope in x_2 now becomes -3 . This analysis then continues in a similar way. Figures 4.1 and 4.2 show the results of this simulation. These results look, in fact, similar to the analytical solution of the original system Equation 4.1. Apparently, when we replace x_i by $q_i = \text{floor}(x_i)$ in a system such as Equation 4.1, we obtain an approximation to the original problem that can be solved in a finite number of steps and exhibits a behavior similar to that of the original system. Moreover, we can associate the solution of the system of Equation 4.3 with the behavior of a discrete-event system. Here the events correspond to the discretization level crossings of the states x_i , and when they occur, they may provoke changes in the state derivatives, which may then lead to a rescheduling of the next event times.

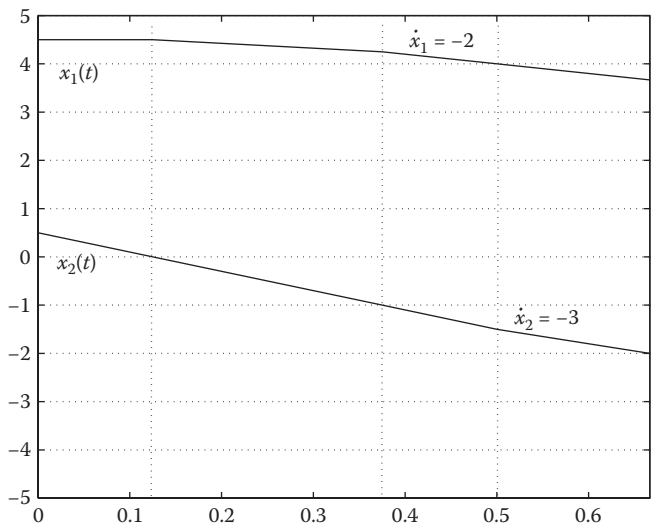


FIGURE 4.1 Simulation trajectories of the system of Equation 4.3 (startup).

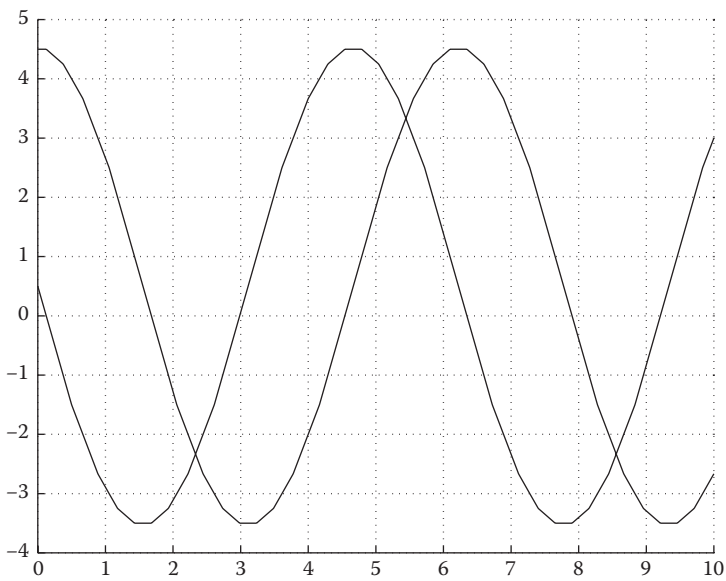


FIGURE 4.2 Simulation trajectories of the system of Equation 4.3.

4.2.2 QUANTIZED SYSTEMS AND DEVS

The original system of Equation 4.1 can be represented by the block diagram of Figure 4.3. Similarly, the block diagram of the modified system of Equation 4.3 is shown in Figure 4.4. The modification of Equation 4.1 with the addition of a “floor” function is usually referred to as *quantization*, and the block in Figure 4.4 that

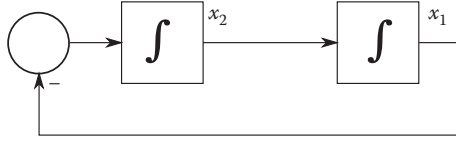


FIGURE 4.3 Block diagram representation of Equation 4.1.

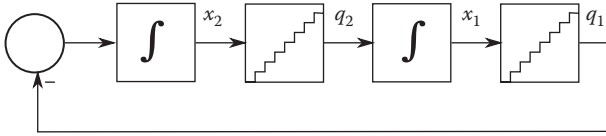


FIGURE 4.4 Block diagram representation of Equation 4.3.

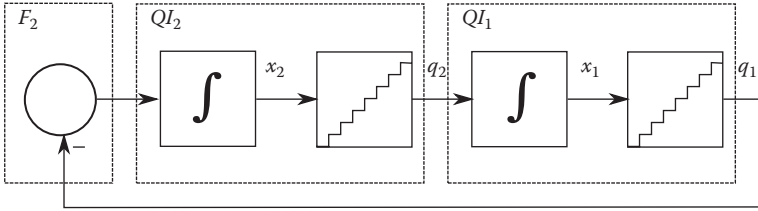


FIGURE 4.5 Block diagram representation of Equation 4.3.

computes the function $\text{floor}(x_i)$ is called *quantizer*. A system like that of Equation 4.3 is called *quantized system*.

The connection between the *quantization* of continuous systems and the DEVS formalism has been studied since the end of the 1990s [24,25]. The basic idea is that quantization enforces variables q_i and state derivatives $\dot{x}_i$ to follow piecewise constant trajectories. Then, these trajectories can be represented by sequences of events. For instance, if we split the block diagram of Figure 4.4 as depicted in Figure 4.5, we can recognize three subsystems with piecewise constant input and output trajectories: F_2 , QI_1 , and QI_2 .

Each subsystem QI_i integrates a piecewise constant input trajectory $\dot{x}_i(t)$ to compute a piecewise linear state trajectory $x_i(t)$ and outputs a piecewise constant quantized trajectory $q_i(t) = \text{floor}[x_i(t)]$. This output only changes when the piecewise linear state trajectory $x_i(t)$ crosses through a discretization level. This behavior can be easily represented by a DEVS model. A possible representation is given below:

$$QI = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ where}$$

$$X = Y = \Re$$

$$S = \Re^2 \times \mathbb{Z} \times \Re_0^+$$

$$\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, q, \sigma) = \left(x + \sigma \cdot d_x, d_x, q + \text{sign}(d_x), \frac{1}{|d_x|} \right)$$

$$\delta_{\text{ext}}(s, e, v) = \delta_{\text{ext}}(x, d_x, q, \sigma, e, v) = (x + e \cdot d_x, v, q, \tilde{\sigma})$$

$$\lambda(s) = \lambda(x, d_x, q, \sigma) = q + \text{sign}(d_x)$$

$$ta(s) = ta(x, d_x, q, \sigma) = \sigma,$$

where

$$\tilde{\sigma} = \begin{cases} \frac{q+1-x}{v} & \text{if } v > 0 \\ \frac{q-x}{v} & \text{if } v < 0 \\ \infty & \text{otherwise} \end{cases}.$$

This model stores in the state s the value of the state variable x , the value of its derivative d_x , the value of the quantized variable $q = \text{floor}(x)$, and the time to the next event σ (i.e., the time to the next change in variable q).

The DEVS model QI is an exact representation in terms of DEVS of the behavior of an integrator followed by a *quantizer*. This is why model QI is usually called *quantized integrator*. The subsystem F_2 in Figure 4.5 has an even simpler behavior. It receives a piecewise constant quantized trajectory $q_2(t)$ and computes a piecewise constant state derivative $\dot{x}_1(t) = -q_2(t)$. The corresponding DEVS model can be obtained in a straightforward manner and so it is not described here. This model performs a *static* operation over $q_2(t)$ and therefore it is called *static function*.

Since the DEVS models QI_i exactly represent the behavior of the integrators followed by quantizers and a DEVS model F_2 exactly represents the behavior of a static function, the coupling of the DEVS models QI_i and F_2 shown in Figure 4.5 exactly represents the behavior of the system of Equation 4.3.

It is clear now that a DEVS model can exactly represent a quantized system like that of Equation 4.3, which is in turn an approximation of the continuous system of Equation 4.1. Yet, can we proceed in this fashion with *every* continuous system?

Consider, for instance, the following first-order ODE:

$$\dot{x}_a(t) = -x_a(t) - 0.5 \quad (4.4)$$

with initial state $x(0) = 0$. If we proceed as before, replacing $x_a(t)$ by $q(t) \triangleq \text{floor}[x(t)]$ at the right-hand side of Equation 4.4, we obtain the following equation:

$$\dot{x}(t) = -q(t) - 0.5. \quad (4.5)$$

Let us try to solve this equation. At time $t = 0$, we have $x(0) = 0$ and therefore $q(0) = 0$. Then, the derivative is $\dot{x}(0) = -0.5$, which means that $x(0^+)$ is negative. Thus, $q(0^+) = -1$ and then $\dot{x}(0^+) = +0.5$.

As the state derivative is now positive, $x(t)$ immediately returns back to zero, and so does $q(t)$. Hence the system returned to the initial condition $x = 0$ without ever advancing the simulation clock. This means that we obtain a permanent oscillation in which q toggles between 0 and -1 . The problem is that the time period of the oscillation is 0 (i.e., the frequency is infinite).

From a mathematical point of view, Equation 4.5 does not have a solution. From a DEVS point of view, the equivalent system (coupling a quantized integrator and a static function) results are *illegitimate*. If we couple a quantized integrator with a static function that computes $-q(t) + 0.5$, we obtain an illegitimate behavior, and the simulation performs an infinite number of events without advancing the simulation clock at all.

The methodology of approximating a continuous system by a quantized system and finding its DEVS equivalent does evidently not constitute a generally applicable method for the simulation of continuous systems as there may appear infinitely fast oscillations. Unfortunately, what happened in this example is not exceptional. It occurs with most systems. For instance, if we replace the constant value of -0.5 in Equation 4.5 by any noninteger value, we encounter the same problem. However as we shall see in the next section, this problem can be easily solved, and a new family of numerical integration algorithms can be derived based on this methodology.

4.2.3 HYSTERESIS AND QUANTIZED STATE SYSTEMS

If we try to analyze the infinitely fast oscillations in the system of Equation 4.5, we can see that they are caused by the changes in $q(t)$. An infinitesimally small variation in $x(t)$ can produce, due to the quantization, a significant oscillation with an infinitely fast frequency in $q(t)$.

One possible solution to this problem is based on the use of hysteresis in the quantization. If we add hysteresis to the relationship between $x(t)$ and $q(t)$, the oscillations in $q(t)$ can only be produced by *large* oscillations in $x(t)$ that cannot occur instantaneously, as long as the state derivative $\dot{x}(t)$ remains finite. Figure 4.6 shows the way a quantization function with hysteresis relates two variables $x_i(t)$ and $q_i(t)$ to each other. We shall call the block that computes $q_i(t)$ from $x_i(t)$ in this fashion a *hysteretic quantizer*.

Notice that a hysteretic quantizer has memory. That is, it computes $q_i(t)$ as a function not only of the present value $x_i(t)$ but also of the previous values of x_i . Thus, as Figure 4.6 shows, the quantized variable q_i can adopt two different values for the same value of the state x_i . Formally, we say that a piecewise constant trajectory $q_i(t)$ is related to a continuous trajectory $x_i(t)$ by a hysteretic quantization function if:

$$q_i(t) = \begin{cases} q_i(t^-) & \text{if } |q_i(t^-) - x_i(t^-)| < \Delta Q_i \\ x_i(t) & \text{otherwise.} \end{cases} \quad (4.6)$$

In this case, ΔQ_i is called *quantum*.

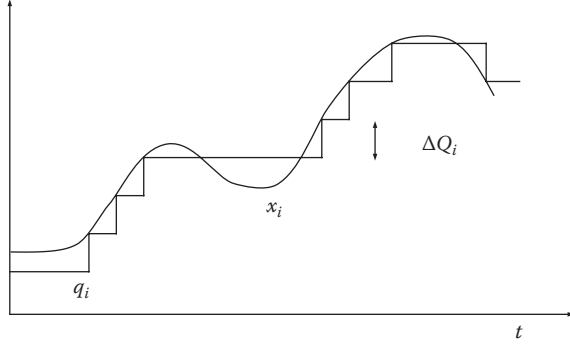


FIGURE 4.6 Hysteretic quantization with *quantum* and *hysteresis width* ΔQ_i .

The *hysteresis width* in Figure 4.6 and Equation 4.6 result in being equal to the quantum. Although the two constants could be chosen differently, this choice is optimal for quantization-based simulation, as has been shown by Kofman, Lee, and Zeigler [17]. When $q_i(t)$ is computed from $x_i(t)$ using a hysteretic quantization function, two consecutive changes in q_i require that the value of x_i changes at least by ΔQ_i . This property avoids the appearance of infinitely fast oscillations.

Now, we are ready to define the QSS method [16]. Given a time invariant system in state equation form:

$$\begin{aligned}
 \dot{x}_{a_1} &= f_1(x_{a_1}, x_{a_2}, \dots, x_{a_n}, u_1, \dots, u_m) \\
 \dot{x}_{a_2} &= f_2(x_{a_1}, x_{a_2}, \dots, x_{a_n}, u_1, \dots, u_m), \\
 &\vdots \\
 \dot{x}_{a_n} &= f_n(x_{a_1}, x_{a_2}, \dots, x_{a_n}, u_1, \dots, u_m)
 \end{aligned} \tag{4.7}$$

with known piecewise constant input functions $u_j(t)$, the QSS method approximates it by the following system:

$$\begin{aligned}
 \dot{x}_1 &= f_1(q_1, q_2, \dots, q_n, u_1, \dots, u_m) \\
 \dot{x}_2 &= f_2(q_1, q_2, \dots, q_n, u_1, \dots, u_m), \\
 &\vdots \\
 \dot{x}_n &= f_n(q_1, q_2, \dots, q_n, u_1, \dots, u_m)
 \end{aligned} \tag{4.8}$$

where $q_i(t)$ is related to $x_i(t)$ by a hysteretic quantization function with quantum ΔQ_i .

Figure 4.7 shows the block diagram representation of the generic *QSS* of Equation 4.8. As before, the whole system can be split into *hysteretic quantized integrators* and static functions, which in turn can be exactly represented by DEVS models. The DEVS atomic model of the hysteretic quantized integrator is similar to *QI* in the previous section:

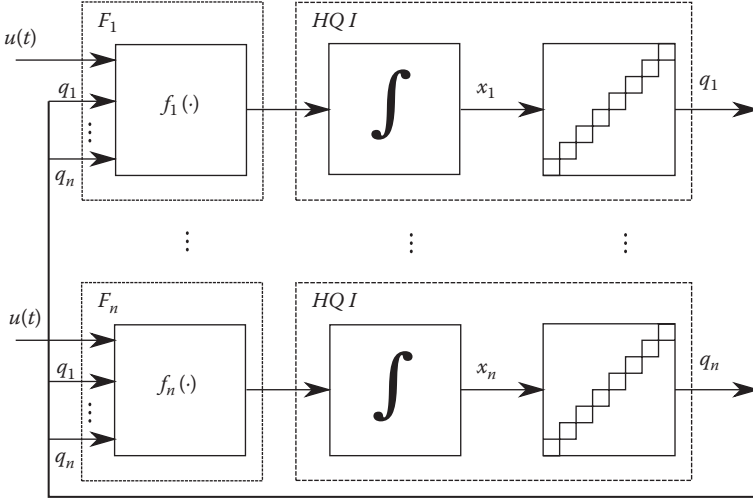


FIGURE 4.7 Block diagram representation of a quantized state system.

$HQI = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta)$, where

$$X = Y = \Re$$

$$S = \Re^2 \times \mathbb{Z} \times \Re_0^+$$

$$\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, q, \sigma) = \left(x + \sigma \cdot d_x, d_x, q + \text{sign}(d_x) \cdot \Delta Q, \frac{\Delta Q}{|d_x|} \right)$$

$$\delta_{\text{ext}}(s, e, v) = \delta_{\text{ext}}(x, d_x, q, \sigma, e, v) = (x + e \cdot d_x, v, q, \tilde{\sigma})$$

$$\lambda(s) = \lambda(x, d_x, q, \sigma) = q + \text{sign}(d_x) \cdot \Delta Q$$

$$ta(s) = ta(x, d_x, q, \sigma) = \sigma,$$

where

$$\tilde{\sigma} = \begin{cases} \frac{q + \Delta Q - x}{v} & \text{if } v > 0 \\ \frac{q - \Delta Q - x}{v} & \text{if } v < 0 \\ \infty & \text{otherwise} \end{cases}.$$

Besides the generalization of the value of ΔQ (in QI we assumed that it was equal to 1), the presence of hysteresis introduces a small change in the calculation of σ at the external transition function. When the state x decreases, the next event is scheduled at the time when x reaches $q - \Delta Q$ instead of q .

4.2.4 ACCURACY OF QSS APPROXIMATIONS

The coupling of hysteretic quantized integrators and static functions shown in Figure 4.1 can exactly simulate the QSS of Equation 4.8. Since our primary goal was to simulate the system of Equation 4.7, we must study the relationship between the solutions of both systems.

We shall start our analysis comparing the analytical and the quantized state solutions of an example system. The following system represents a spring–damper–mass mechanical system:

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= (-k \cdot x_1 - b \cdot x_2 + F)/m.\end{aligned}\tag{4.9}$$

Here, the state variables x_1 and x_2 represent the position and velocity of the mass, respectively. The parameters m , b , and k correspond to the mass, the damping, and the spring constants. The signal $F(t)$ is a known constant force applied to the system. Since this system is linear, its analytical solution can be easily computed. For the set of parameters $b = k = m = F = 1$ and taking initial states $x_1(0) = x_2(0) = 0$, we have

$$\begin{aligned}x_1(t) &= 1 - \frac{\sqrt{3}}{3}e^{-t/2} \sin \frac{\sqrt{3}}{2}t - e^{-t/2} \cos \frac{\sqrt{3}}{2}t \\ x_2(t) &= \frac{\sqrt{12}}{3}e^{-t/2} \sin \frac{\sqrt{3}}{2}t,\end{aligned}\tag{4.10}$$

for all $t \geq 0$.

We first simulate this system using the QSS method with quanta $\Delta Q_1 = \Delta Q_2 = 0.1$. The results are displayed in Figure 4.8. We then reduce the quantum by a factor of 5 in both variables. The new results are shown in Figure 4.9. It is evident that the global error is reduced with respect to the previous simulation. It is also clear that the number of steps has grown.

We continue reducing the quantum, and in each case, we compute the maximum error (i.e., the global simulation error) and the number of steps performed. The results are summarized in Table 4.1. As can be seen, the number of steps (i.e., changes in q_i) is inversely proportional to the quantum. This result is rather obvious. If we proceed from 0 to 10 with jumps of amplitude 1, we need 10 jumps. If

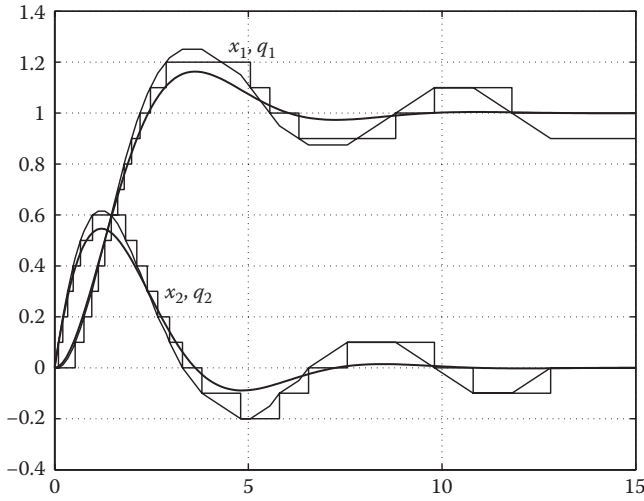


FIGURE 4.8 QSS simulation of Equation 4.9 with $\Delta Q_1 = \Delta Q_2 = 0.1$. The bold line corresponds to the analytical solution.

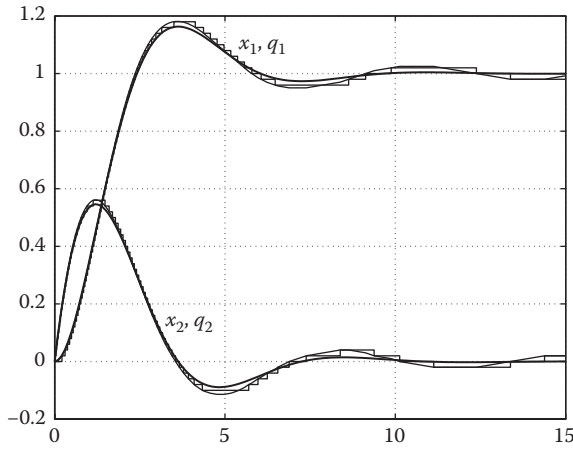


FIGURE 4.9 QSS simulation of Equation 4.9 with $\Delta Q_1 = \Delta Q_2 = 0.02$. The bold line corresponds to the analytical solution.

the jumps are of amplitude 0.1, we need 100 jumps. Another remark is that the error seems to be proportional to the quantum. We shall prove this fact in the next section.

The consequence of both remarks is that the number of steps grows linearly with the global accuracy, and this is coherent with the fact that QSS performs a first-order approximation. This property has bad consequences when we want to obtain

TABLE 4.1
Simulation of Equation 4.9 with the QSS
Method

$\Delta Q_{1,2}$	# of Steps	Maximum Error
0.1	40	0.113
0.01	282	0.0129
0.001	2704	0.00143
0.0001	26,946	0.0001385

accurate results. Yet, we shall subsequently discuss two higher-order methods, in which the number of steps grows with the square and cubic roots of the accuracy.

4.2.5 THEORETICAL PROPERTIES OF QSS METHODS

Using vector notation, the original system of Equation 4.7 takes the form:

$$\dot{\mathbf{x}}_a(t) = \mathbf{f}(\mathbf{x}_a(t), \mathbf{u}(t)), \quad (4.11)$$

while its QSS approximation of Equation 4.8 can be written as:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)). \quad (4.12)$$

Defining $\Delta \mathbf{x}(t) = \mathbf{q}(t) - \mathbf{x}(t)$, the latter can be rewritten as:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t) + \Delta \mathbf{x}(t), \mathbf{u}(t)), \quad (4.13)$$

which only differs from the original system Equation 4.11 in the presence of a perturbation term $\Delta \mathbf{x}(t)$. A fundamental property of the hysteretic quantization is that $x_i(t)$ and $q_i(t)$ cannot differ from each other by more than the quantum ΔQ_i (see Figure 4.6). Thus, each component $\Delta x_i(t)$ of the vector $\Delta \mathbf{x}(t)$ is bounded by the quantum ΔQ_i . The consequence of this fact is that the effect of using the QSS approximation can be analyzed as a problem of bounded perturbations in an ODE.

Based on this idea, the first property proven in the context of the QSS method was that of *convergence* [16]. The analysis shows that the solutions of Equation 4.12 approach those of Equation 4.11 when the largest quantum ΔQ_i is chosen sufficiently small. The importance of this property lies in the fact that an arbitrarily small simulation error can be achieved, when a sufficiently small quantization is being used.

A sufficient condition that ensures that the trajectories of the system of Equation 4.12 converge to the trajectories of Equation 4.11 is that the function $\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))$ is

locally Lipschitz. Hence the convergence of the QSS method is a property satisfied by nonlinear systems in general. Although convergence constitutes an important theoretical property, it does not offer any quantitative information about the relationship between the quantum and the error, and it does not establish any condition for the stability domain.

The stability properties of the QSS method were studied in Kofman and Junco [16] by finding a Lyapunov function for the perturbed system. The analysis shows that, when the system of Equation 4.11 has an asymptotically stable equilibrium point for any arbitrarily small region around that equilibrium point, a quantization can be found, so that the solutions of Equation 4.12 finish inside that region. Moreover, an algorithm can be derived from this analysis that allows calculating the appropriate quantum.

A sufficient condition for ensuring numerical stability of the QSS method in the proximity of an analytically stable equilibrium point is that the function $\mathbf{f}$ be continuous and continuously differentiable. Hence the stability condition is a bit stronger than the convergence condition. Thus, the QSS method offers tools that can be applied to nonlinear systems for choosing a quantum that ensures that the steady-state simulation error is smaller than a desired bound. Although this result represents an important advantage over the classical discrete-time methods, where stability is usually studied in the context of linear time-invariant (LTI) systems only, the algorithm is quite involved and requires the use of a Lyapunov function of Equation 4.11 that cannot be easily derived in general cases. Thus, the importance of this stability analysis is more of a theoretical than a practical nature.

Like in discrete-time methods, the most interesting properties of the QSS method come from the analysis of its application to LTI systems. The main result of that analysis, performed by Kofman [10], states that the error in the QSS simulation of an asymptotically stable LTI system is always bounded. The error bound, which can be calculated from the quantum and some geometrical properties of the system, does not depend on either the initial condition or the input trajectory and remains constant during the simulation.

An LTI system can be written as:

$$\dot{\mathbf{x}}_a(t) = \mathbf{A} \cdot \mathbf{x}_a(t) + \mathbf{B} \cdot \mathbf{u}(t), \quad (4.14)$$

with $\mathbf{A}$ and $\mathbf{B}$ being $n \times n$ and $m \times n$ real-valued matrices, respectively, where n is the order and m denotes the number of inputs of the system.

A QSS approximation of this system is given by:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{q}(t) + \mathbf{B} \cdot \mathbf{u}(t). \quad (4.15)$$

Let $\mathbf{x}_a(t)$ be the solution of Equation 4.14 and $\mathbf{x}(t)$ the solution of the QSS approximation of Equation 4.15 starting from identical initial conditions, $\mathbf{x}_a(t_0) = \mathbf{x}(t_0)$. For

an analytically stable LTI system, the global simulation error is bounded by the following inequality:

$$\|\mathbf{x}(t) - \mathbf{x}_a(t)\| \leq \|V\| \cdot \|\operatorname{Re}\{\Lambda\}^{-1} \cdot \Lambda\| \cdot \|V^{-1}\| \cdot \Delta Q, \quad (4.16)$$

for all $t \geq t_0$.

In this formula, V and Λ are defined by the spectral decomposition of A . The vector $\Delta Q \triangleq [\Delta Q_1, \dots, \Delta Q_n]^T$ concatenates the set of quanta used for each of the states. The symbol $\|\cdot\|$ denotes here the componentwise absolute value of a vector or matrix. Similarly, the symbol “ $\leq$ ” performs a componentwise comparison between real-valued vectors of equal length.

Thus, the expression of Equation 4.16 offers a bound for the global simulation error of each component of the state vector. This error bound depends linearly on the quanta, as we had already observed in the previous simulation examples. For example, the error bound formula applied to the system of Equation 4.9 gives:

$$\begin{aligned} |x_1(t) - x_{a_1}(t)| &\leq 2.3094 \cdot \Delta Q_1 + 2.3094 \cdot \Delta Q_2 \\ |x_2(t) - x_{a_2}(t)| &\leq 2.3094 \cdot \Delta Q_1 + 2.3094 \cdot \Delta Q_2. \end{aligned} \quad (4.17)$$

In the simulation results shown in Figure 4.8, where $\Delta Q_{1,2} = 0.1$, this theoretical error bound is equal to 0.462 for both states. In the simulation of Figure 4.9, the error bound is found to be 0.046. In both cases, the theoretical result turned out to be a bit conservative.

It follows that:

The QSS method offers an intrinsic error control without requiring the use of adaptation rules.

It is also worth noticing that the existence of a global error bound implies that the numerical solutions cannot diverge. The last (and most important) remark is that:

The QSS method remains always numerically stable without involving implicit formulae at all.

Besides the perturbation-based analysis presented here, an alternative way to prove stability of the QSS method was developed in Nutaro and Zeigler [22] making use of multirate, discrete-time equivalents of the QSS approximations.

4.2.6 HIGHER-ORDER QSS METHODS

As can be expected from a first-order algorithm, QSS cannot offer decent accuracy without requiring a large number of simulation steps. Thus for most practical applications, higher-order approximations are needed.

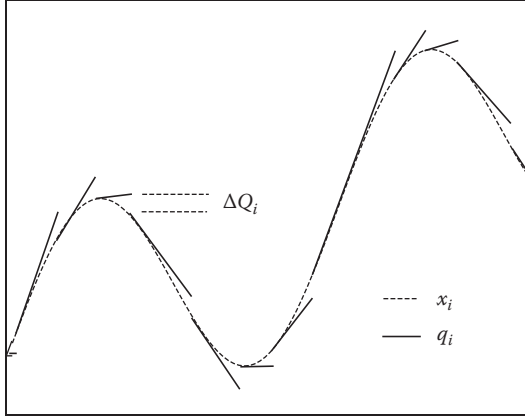


FIGURE 4.10 First-order quantization function.

An n th order accurate method must take into account information up to the n th time derivative of the state vector $\mathbf{x}(t)$. In the QSS method, when replacing x_i by the piecewise constant trajectory q_i obtaining Equation 4.12, we threw away all information concerning the higher time derivatives of the state variables.

However, we can modify the QSS method in order to preserve the second derivative by defining $q_i(t)$ as a piecewise linear approximation of $x_i(t)$. Moreover, if we can ensure that $x_i(t)$ and $q_i(t)$ will never differ from each other by more than the quantum ΔQ_i , we shall also preserve all of the theoretical properties that we analyzed in the previous section.

Following these ideas, we can define the *first-order quantization function* as shown in Figure 4.10. Notice that this function is also hysteretic since $q_i(t)$ changes when the absolute value of the distance from $x_i(t)$ becomes equal to the quantum. A formal definition of the *first-order quantization function* can be found in Cellier and Kofman [4]. Replacing the original hysteretic quantization function of QSS by a first-order quantization function, we obtain the *second-order QSS* (QSS2) method [10].

The definition of QSS2 is almost identical to that of QSS. The two methods only differ in the way in which $q_i(t)$ is computed from $x_i(t)$. Taking into account that QSS2 quantized variables follow piecewise linear trajectories, then it results that the state derivatives $\dot{x}_i(t)$ also follow piecewise linear trajectories* and the state trajectories $x_i(t)$ result in the piecewise parabolic.

Just like the first-order QSS method, the QSS2 also can be implemented by DEVS models of quantized integrators and static functions. However, the new atomic models now must take into account not only the values but also the slopes of their input and output trajectories. The resulting atomic DEVS models can be found in Kofman [10] and Cellier and Kofman [4]. In order to corroborate the second-order nature of QSS2 we repeated the simulations of the system of Equation 4.9 for different quantum values. The results are summarized in Table 4.2.

* To be rigorous, this is only true for LTI systems. In other cases, the state derivatives are approximated by piecewise linear trajectories.

TABLE 4.2
Simulation of Equation 4.9 with the QSS2 Method

$\Delta Q_{1,2}$	# of Steps	Maximum Error
0.1	31	0.1006
0.01	54	0.0077
0.001	139	0.00091
0.0001	449	0.0000885
0.00001	1434	0.0000090

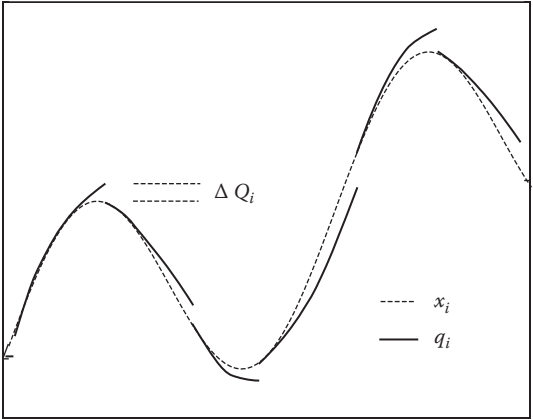


FIGURE 4.11 Second-order quantization function.

The advantage of QSS2 with respect to QSS is evident, particularly for small error tolerances. Now, as can be observed, the number of steps grows with the square root of the accuracy. We can further increase the approximation order using piecewise parabolic trajectories for the quantized variables $q_i(t)$. Figure 4.11 shows this idea, which defines the *second-order quantization function*.

The usage of second-order quantization functions in the relationship between x_i and q_i leads to the third-order accurate QSS method (QSS3) [11], which only differs from QSS2 in the way that q_i is being computed. In order to illustrate the third-order nature of the QSS3 method, we also repeated the simulations of system Equation 4.9, obtaining the results shown in Table 4.3.

4.2.7 IMPLEMENTATION OF THE QSS METHODS

The whole family of QSS methods (including the methods introduced in the next section) were implemented in PowerDEVS, a DEVS-based simulation platform specially adapted to simulate hybrid systems based on QSS methods [2].

TABLE 4.3
Simulation of Equation 4.9 with the QSS3 Method

$\Delta Q_{1,2}$	# of Steps	Maximum Error
0.1	36	0.0476
0.01	39	0.00907
0.001	64	0.000948
0.0001	114	0.0000751
0.00001	241	0.0000060
0.000001	522	0.0000010

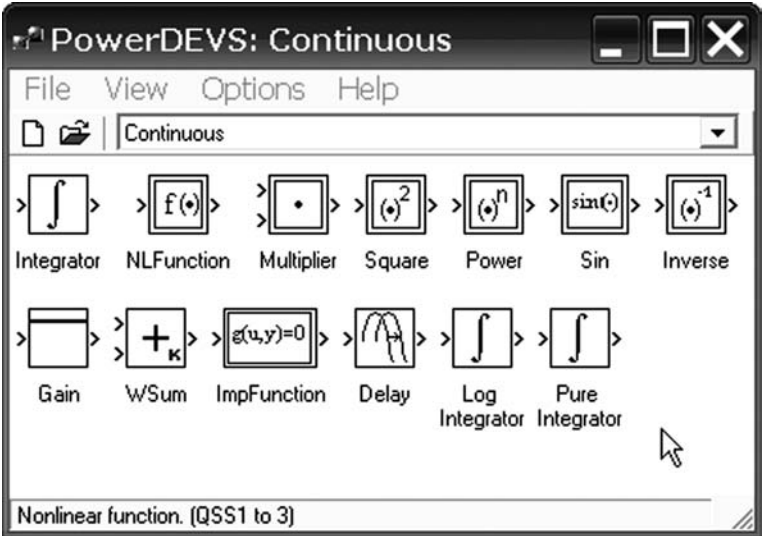


FIGURE 4.12 PowerDEVS Main Window.

PowerDEVS has a graphical user interface similar to that of Simulink® [26], which permits editing block diagrams. At the lowest hierarchy level, each block has a normal atomic DEVS description in C++ language that can be easily edited (with the PowerDEVS atomic editor tool). Yet, the standard distribution already contains all the blocks needed to model continuous and hybrid systems (integrators, math functions, discontinuity handlers, sources, sinks, etc.). Thus, a user can model and simulate with the QSS methods without knowing anything about DEVS, just as he would model and simulate in Simulink.

The library browser of PowerDEVS, showing the continuous blocks, is depicted in Figure 4.12. A model of a bouncing ball, analyzed in the next section, is shown in Figure 4.13. PowerDEVS models can exchange parameters, variables, and

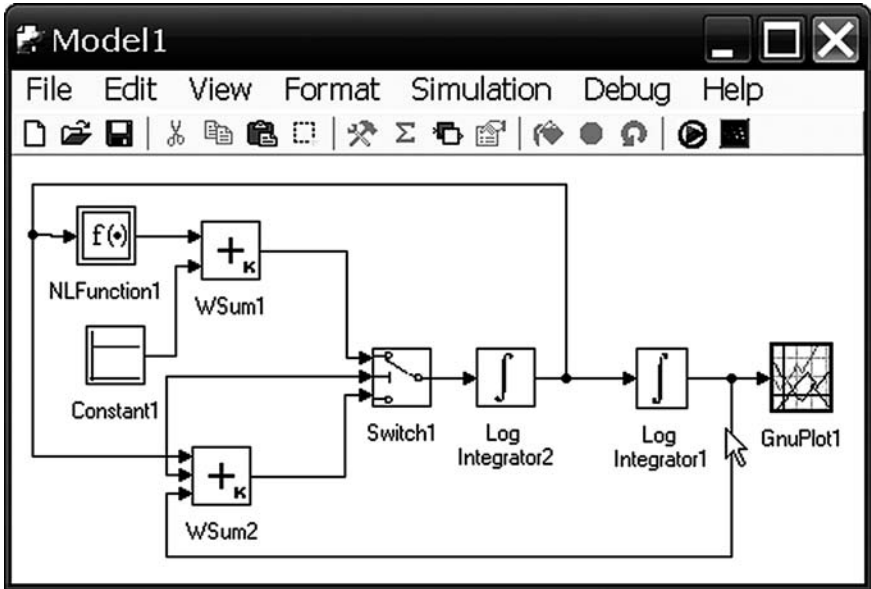


FIGURE 4.13 Bouncing ball model in PowerDEVS.

information with Scilab during the simulation in the same way that Simulink® and MATLAB® [26] do it. This interaction allows exploiting all of the data processing, plotting, math operations, and matrix manipulation capabilities of Scilab, also providing an interactive workspace to the user.

Although primarily designed to work under Windows OS, an ad-hoc version of Kubuntu 8.04 was developed, which includes PowerDEVS and Real Time Linux (RTAI) modules. The PowerDEVS simulation engine includes modules that use RTAI functions for real-time synchronization, interrupt handling, physical time measurement, and so on. These real-time modules are also included in new atomic DEVS blocks that permit synchronizing the simulations with a real-time clock with an assured accuracy of roughly 2 μ sec, detecting hardware interrupts, accessing I/O ports, and measuring the physical time with a resolution of the order of nano-seconds [2].

Thus, a user can perform simulations based on the QSS methods in real time in a very convenient fashion. In addition to their implementation in PowerDEVS, the explicit QSS methods of orders 1–3 were also implemented in Modelica [1], Open Source Physics, a Java-based simulation tool [7], and implementations of the first-order QSS methods can also be found in CD++ [6] and VLE [23].

4.3 DISCONTINUOUS, STIFF, AND marginally STABLE SYSTEMS

Many systems exhibit features that complicate the job of numerical integration algorithms. Among those features, we may mention discontinuities, stiffness, and

marginal stability. In this section we shall analyze these problems in the context of the QSS methods.

4.3.1 DISCONTINUITY HANDLING

Figure 4.14 shows a bouncing ball in two possible situations. While the ball is in the air, we take into account the influence of gravity and air friction. When the ball is on the floor, we consider a spring–damper model. This system can be modeled by the following equations:

$$\begin{aligned} \dot{y}(t) &= v(t) \\ \dot{v}(t) &= \begin{cases} -g - \frac{b_a}{m} \cdot v(t) \cdot |v(t)| & \text{if } y(t) > 0, \\ -g - \frac{k}{m} \cdot y(t) - \frac{b}{m} \cdot v(t) & \text{otherwise} \end{cases} \end{aligned} \quad (4.18)$$

where g is the gravity acceleration, m is the mass, k is the spring constant, b_a is the air friction coefficient, and b is the damper constant. The problem with this model is that the right-hand side of the last equation is discontinuous. If a numerical integration method performs an integration step that crosses through the discontinuity, the result may have an unacceptable error. Conventional numerical methods solve this problem by finding the instant of time, at which the discontinuity occurs (this is usually referred to as *root solving*). They advance the simulation up to that point, and then restart the integration from the new initial condition after the event.

Although this approach works well, it adds computational cost: root solving implies performing some iterations, and also restarting the simulation can be quite expensive. How do QSS methods handle this problem? As we have already seen, the trajectories are piecewise linear, parabolic, or cubic depending on the order. Thus, the event time can be analytically detected.

Although the event associated with a discontinuity must occur at the correct time instant, the methods do not need to restart after that. After all, discontinuities in QSS occur regularly, as the trajectories of $q_i(t)$ are discontinuous. Thus for QSS approximations, a discontinuity has the same effect as a normal step [13].

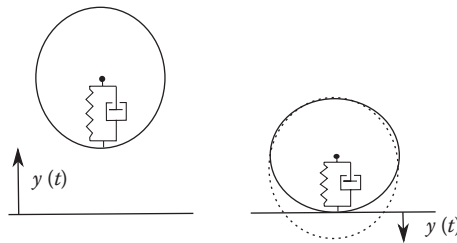


FIGURE 4.14 Bouncing ball.

Let us illustrate this point analyzing the PowerDEVS implementation of the bouncing ball example shown in Figure 4.13. The block “Switch1” is in charge of modeling and handling the discontinuity. It receives the successive values (and slopes) of $y(t)$ through its second input port. When a new value arrives, it solves a quadratic equation (the integrators use QSS3, which produces piecewise parabolic quantized trajectories) to compute the next crossing time, and it schedules its next internal transition to occur at that time. Meanwhile, the rest of the system continues with the simulation. When a new value of y arrives, the switch reschedules its next event time. When the event time finally arrives, the switch sends an output event with the new value of the derivative $\dot{v}(t)$. The integrator that computes $v(t)$ receives this event as a normal change in $\dot{v}(t)$ and treats it as a normal step.

In other words, the atomic DEVS model “Switch1” locally handles the discontinuities, while the remaining blocks are unaware of their presence. For these reasons, QSS methods are in general more efficient than conventional methods for handling discontinuities. In systems where the occurrence of discontinuities is faster than the continuous dynamics as this happens frequently in power electronic circuits, for example, QSS methods can significantly reduce the computational costs compared with classic ODE solvers [13].

In the example analyzed, the event condition depends on one state variable (y). When a discontinuity depends on more than one state variable, the problem is almost the same. A static block (like NLFunction or WSum in Figure 4.13) can compute the piecewise parabolic crossing trajectory, and that trajectory can be sent to the switch block.

4.3.2 STIFF SYSTEMS AND BACKWARD QSS

The system

$$\begin{aligned}\dot{x}_1(t) &= 0.01x_2(t) \\ \dot{x}_2(t) &= -100x_1(t) - 100x_2(t) + 2020,\end{aligned}\tag{4.19}$$

has eigenvalues $\lambda_1 \approx -0.01$ and $\lambda_2 \approx -99.99$. The second eigenvalue is very fast compared to the first one (i.e., the system is stiff). Considering initial conditions $x_1(0) = 0$, $x_2(0) = 20$ together with the quanta $\Delta Q_1 = \Delta Q_2 = 1$, the QSS method produces the results shown in Figures 14.15 and 14.16.

Although the results are correct, there are fast oscillations of q_2 that provoke a total of 15,995 transitions in that variable, whereas q_1 only changes 21 times. Consequently, the total number of steps needed to complete the simulation is greater than 16,000. This number is of the same order as the number of steps needed by any classical explicit numerical ODE solver.

Evidently, the QSS method is unable to efficiently solve the system of Equation 4.19. The situation becomes even worse when using QSS2. For the same quanta, we obtain 9 changes in q_1 and 65,444 changes in q_2 . Using QSS3 doesn't help either, as we observe 415 and 92,733 changes in q_1 and q_2 , respectively.

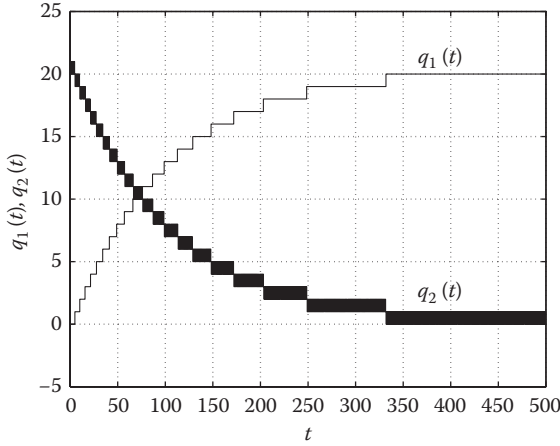


FIGURE 4.15 QSS simulation.

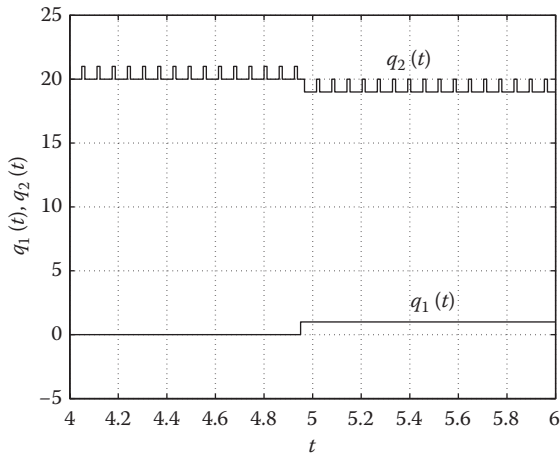


FIGURE 4.16 QSS simulation (detail).

As we already know from the theory of classic numerical ODE solvers, an efficient solution of stiff systems requires using implicit algorithms that evaluate the state derivatives at future instants of time. This idea, when applied to QSS methods, would imply that the components of $\mathbf{q}(t)$ in Equation 4.12 are quantized versions of future values of $x(t)$. In other words, given $x_i(t)$, $q_i(t)$ should be a quantized value in the neighborhood of $x_i(t)$, such that $x_i(t)$ evolves toward $q_i(t)$. Applying this idea to the previous example of Equation 4.19 using the same initial conditions and quantization as before, the following simulation is obtained.

At $t = 0$, we choose either $q_2(0) = 19$ or $q_2(0) = 21$ depending on the sign of $x_2(0)$. In both cases, it results that $\dot{x}_1(0) > 0$ and the *future* quantized value of x_1 is $q_1(0) = 1$.

If we choose $q_2(0) = 21$, it results that $\dot{x}_2(0) = -180 < 0$, and consequently, x_2 does not evolve toward q_2 . On the other hand, choosing $q_2(0) = 19$ implies that $\dot{x}_2(0) = 20 > 0$, and once again, x_2 does not evolve toward q_2 . Hence, it is not possible to choose q_2 such that x_2 moves toward q_2 . However, the fact that the sign of $\dot{x}_2$ changes taking $q_2 = 19$ and $q_2 = 21$ implies that there must exist a point, $\hat{q}_2$, between those two values, for which $\dot{v}(t) \dot{x}_2 = 0$. In this case, we set arbitrarily $q_2 = 21$, but we let $\dot{v}(t) \dot{x}_2 = 0$, as if q_2 had adopted the (unknown) value $\hat{q}_2$. We could have chosen $q_2 = 19$ instead. We would have received a different approximation in this way with very similar results.

The next change in q_1 is thus scheduled at $t = 1/0.21 \approx 4.762$, whereas the next change in q_2 is scheduled at $t = \infty$. Hence, the next step is evaluated at $t = 4.762$. Here, it results that $x_1 = 1$ and $x_2 = 20$. Then, $q_1(4.762) = 2$ (because $\dot{x}_1 > 0$). We evaluate again $\dot{x}_2$ for $q_2 = 19$ and $q_2 = 21$. Now the value is negative in both cases. Thus, the correct value is $q_2(4.762) = 19$ since in that way, x_2 moves toward q_2 . With these new values of q_1 and q_2 , it results that $\dot{x}_1 = 0.19$ and $\dot{x}_2 = -80$. The next change in q_1 is then scheduled at $t = 4.762 + 1/0.19 = 10.025$, whereas the next change in q_2 is scheduled at $t = 4.762 + 1/80 = 4.774$. Thus, the next step is performed at $t = 4.774$, when x_2 reaches q_2 .

The calculations continue in the same way. The algorithm is similar to that of QSS. The difference is that we try to choose q_i such that x_i evolves toward q_i . When this is not possible, this means that there must exist a point near x_i , for which $\dot{x}_i = 0$. In that case, we enforce that condition but, instead of calculating that point, we keep the previous value of q_i .

Figure 4.17 shows the result of this simulation that took 21 changes in q_1 and 22 changes in q_2 . At $t = 354.24$, the algorithm reaches a stable situation where the changes in both variables are scheduled at $t = \infty$. This is the basic idea that defines the Backward QSS (BQSS) method. For each state variable x_i , we use two quantization functions, one from above and the other from below x_i . Then,

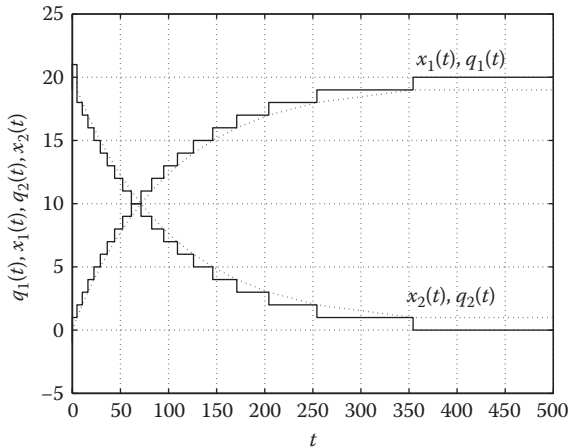


FIGURE 4.17 BQSS simulation.

q_i takes its value equal to one or the other function according to the sign of the derivative $\dot{x}_i$.

In the case analyzed, this idea worked very well. The algorithm solved the stiff system of Equation 4.19 using 43 steps only, which is comparable to the performance of a classical numerical stiff ODE solver. Evidently, the method is only first-order accurate, and consequently, the approximation error is large for decently large values of the quanta.

The number of steps performed coincides, as can be expected, with the trajectory amplitudes divided by the quantum. This is also true for the QSS method, except that QSS provokes fast oscillations (Figures 4.15 and 4.16) that increase the number of steps. The BQSS method, by looking at the future value of the states and enforcing the state derivatives to be zero in the presence of sign changes, avoids the appearance of those fast oscillations. For this simple reason, BQSS works fine with stiff systems like that of Equation 4.19.

The BQSS method [21] can be formally defined as follows. Given the system of Equation 4.11, the BQSS method approximates it by:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)) + \Delta \mathbf{f}, \quad (4.20)$$

where the components of $\mathbf{q}$ are chosen from the set:

$$q_j(t) \in \{\underline{q}_j(t), \bar{q}_j(t)\} \quad (4.21)$$

with:

$$\underline{q}_j(t) = \begin{cases} \underline{q}_j(t^-) - \Delta Q_j & \text{if } x_j(t) - \underline{q}_j(t^-) \leq 0 \\ \underline{q}_j(t^-) + \Delta Q_j & \text{if } x_j(t) - \underline{q}_j(t^-) \geq \epsilon_j + \Delta Q_j \\ \underline{q}_j(t^-) & \text{otherwise} \end{cases} \quad (4.22)$$

$$\bar{q}_j(t) = \begin{cases} \bar{q}_j(t^-) + \Delta Q_j & \text{if } \bar{q}_j(t^-) - x_j(t) \leq 0 \\ \bar{q}_j(t^-) - \Delta Q_j & \text{if } \bar{q}_j(t^-) - x_j(t) \geq \epsilon_j + \Delta Q_j \\ \bar{q}_j(t^-) & \text{otherwise} \end{cases} \quad (4.23)$$

In other words, $q_j(t)$ is chosen from a set of two values denoting a lower and an upper bound. These bounds by themselves are being calculated from their own previous values. We furthermore choose $q_j(t)$ such that $x_j(t)$ approaches $q_j(t)$:

$$f_j(\mathbf{q}(t), \mathbf{u}(t)) \cdot (q_j(t) - x_j(t)) > 0, \quad (4.24)$$

and if either none or both of the two possible values $q_j(t)$ and $\bar{q}_j(t)$ satisfy the condition Equation 4.24, then there must exist a vector $\hat{\mathbf{q}}^{(j)}(t)$, such that $f_j(\hat{\mathbf{q}}^{(j)}(t), \mathbf{u}(t)) = 0$:

$$\exists \hat{\mathbf{q}}^{(j)}(t) \mid f_j(\hat{\mathbf{q}}^{(j)}(t), \mathbf{u}(t)) = 0, \quad (4.25)$$

where each component of the vector $\hat{\mathbf{q}}^{(j)}(t)$ satisfies:

$$\mid x_i(t) - \hat{q}_i^{(j)}(t) \mid < \Delta Q_i + \varepsilon_i. \quad (4.26)$$

Finally, we choose the increments:

$$\Delta f_j = \begin{cases} 0 & \text{if } f_j(\mathbf{q}(t), \mathbf{u}(t)) \cdot (q_j - x_j) > 0 \\ -f_j(\mathbf{q}(t), \mathbf{u}(t)) & \text{otherwise} \end{cases}; \quad (4.27)$$

that is, the increments are either zero if a unique consistent evolution has been found, or alternatively, the increments are chosen such that the corresponding derivatives are set equal to zero.

Just like in QSS, ΔQ_j is called the quantum and ε_j is the hysteresis width. Contrary to QSS, it is convenient here to choose smaller values of $\varepsilon_j < \Delta Q_j$. The reason is that for such values of ε_j , the two hysteresis loops from above and below do not intersect. Also contrary to QSS, the oscillation frequency of BQSS does not grow linearly with decreasing values of ε_j . In the current implementation, BQSS sets $\varepsilon_j = 0.01 \cdot \Delta Q_j$.

BQSS has similar theoretical properties to those of QSS. As in the explicit methods, the difference between x_i and q_i remains bounded by the quantum. However, there is an additional perturbation term $\Delta \mathbf{f}$ in Equation 4.20 that enforces the derivatives to be equal to zero when the state approaches a point where the sign of the derivative changes. Thus, the formula of Equation 4.16 changes in BQSS, resulting in a larger global error bound.

Compared with the classic numerical stiff ODE solvers, BQSS has the advantage of not requiring any iteration. At each step, there are only two choices for q_i . Yet, since BQSS is only a first-order accurate, the method cannot compete with higher-order methods in the simulation of systems with stringent accuracy requirements.

We are currently working on deriving stiffly stable QSS methods of higher order. Unfortunately, extending the idea of BQSS to a second-order accurate method is nontrivial. The reason is that a BQSS2 method must use information about the second derivative, and that information is not available when we enforce a derivative to be zero without finding the value of the state for which that condition occurs. Thus, a second-order method requires finding the point for which the derivatives are zero, and this once again leads to an implicit formulation. Although not yet fully implemented, we designed a second-order linearly implicit QSS method (LIQSS2) [20]. This new method works in a mostly explicit way, except for those steps when it

cannot find a correct value for q . In that case, it solves a linear system of equations (commonly limited to a subset of states) to find the point at which the second derivative becomes zero.

The PowerDEVS implementations of BQSS, LIQSS1, and LIQSS2 have been compared in terms of accuracy and computational effort with conventional ODE solvers in several benchmark tests [20,21]. As can be expected, BQSS and LIQSS can effectively reduce computational efforts only for low accuracy requirements, as these methods perform low order approximations only.

4.3.3 MARGINALLY STABLE SYSTEMS AND CENTERED QSS

The harmonic oscillator of Equation 4.1 belongs to the category of *marginally stable systems*. Its analytical solution does not gain or lose energy as time advances. If we simulate this system, we would like to ensure that the numerical solution preserves the energy as well. The simulation of the harmonic oscillator with Forward Euler (FE), for example, produces an oscillation of increasing amplitude (i.e., a numerically unstable solution). In contrast, the usage of Backward Euler (BE) produces an oscillation of decreasing amplitude (i.e., an asymptotically stable solution). On the other hand, if we advance a half step with BE and a half step with FE (i.e., simulate using the trapezoidal rule), we obtain a marginally stable result for any step size.

Besides the trapezoidal rule, there exist several other classic numerical ODE solvers that ensure (at least for linear systems) that the simulation of marginally stable systems produces marginally stable numerical solutions [4,9]. Yet, just like the trapezoidal rule, all of these algorithms are of implicit type.

What happens if we use QSS? Figure 4.18 shows some simulation results. QSS essentially behaves in the same fashion as FE; that is, it produces growing oscillations.* This fact was first studied in Kofman and Zeigler [18], where it was shown that the error grows linearly with time.

How about BQSS? Figure 4.19 displays some curves. BQSS behaves similar to BE (i.e., it produces oscillations that die out after a while). Neither of the two algorithms preserves the energy. It is interesting to discuss, whether QSS and BQSS can be combined to obtain a new simulation method that preserves the energy, analogously to the way in which FE and BE are combined to generate the trapezoidal rule. If we use the mean of the two q_i values as they are produced by QSS and BQSS for the computation of the state derivatives $\dot{x}_i$, we obtain a new method that we shall coin *Centered QSS* or CQSS [5].

Figure 4.20 shows the behavior of the new quantization function. From Figure 4.20, it becomes clear, why the name “centered” QSS was chosen. Whereas the quantized states q_i always lag behind the true states x_i in QSS, they always precede the true states in BQSS. CQSS offers a compromise between these two extremes. The CQSS

* The simulation results of Figure 4.2 do not exhibit growing oscillations, because the simulation was performed using QSS without hysteresis. Thereby q_i lags behind x_i during positive slopes and leads x_i during negative slopes. As the analytical solution is symmetric, so will be the numerical solution.

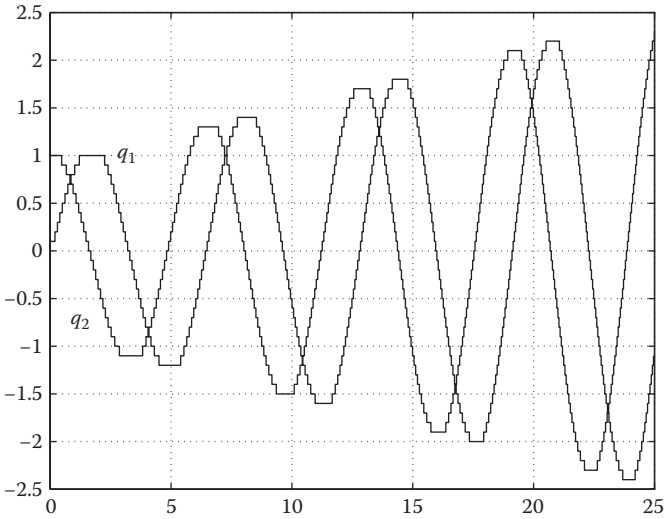


FIGURE 4.18 QSS simulation of a harmonic oscillator, $\Delta Q = 0.1$.

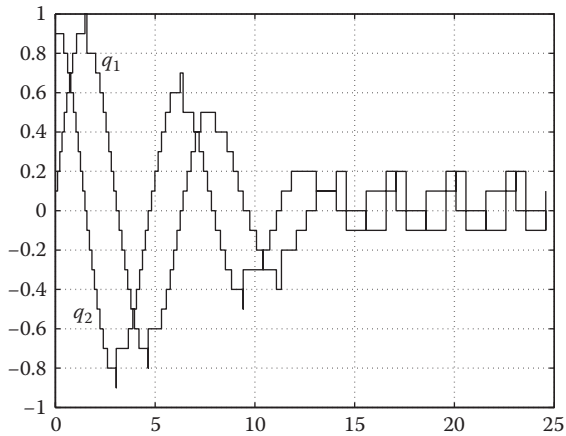


FIGURE 4.19 BQSS simulation of a harmonic oscillator, $\Delta Q = 0.1$.

method is otherwise defined just like QSS. The only difference is that the values of q_i are centered in the interval.

Figure 4.21 shows the simulation results obtained by CQSS. Evidently, CQSS is able to simulate the harmonic oscillator problem of Equation 4.1 while preserving the energy of the system. This result is not an accident. It was meanwhile proven that CQSS is indeed a geometric ODE solver [3], and therefore, can be expected to exhibit a behavior similar to that of the trapezoidal rule.* Unfortunately, and contrary

* The proof, so far, was given for a CQSS simulator without hysteresis only. We are still working on the case of a CQSS simulator with hysteresis.

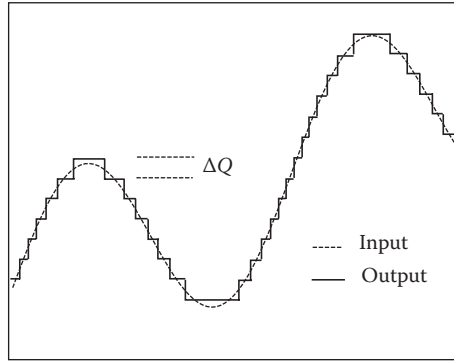


FIGURE 4.20 Centered quantization function.

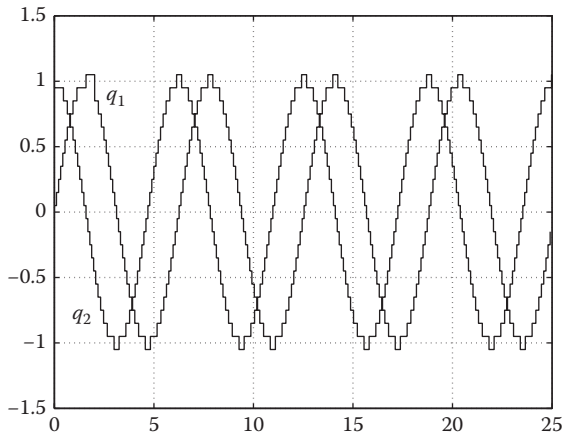


FIGURE 4.21 CQSS simulation of a harmonic oscillator, $\Delta Q = 0.1$.

to the trapezoidal rule, CQSS is still only a first-order accurate. However, CQSS has the advantage of not requiring any iterations.

4.4 QSS METHODS AND DIGITAL CONTROL

Digital simulation is not the only application where differential equations must be discretized. Another important field is that of digital control. A typical feedback control scheme is depicted in Figure 4.22. Here, a continuous-time plant is connected to a continuous-time controller designed for the purpose of controlling the plant. The system of equations can be written as:

$$\begin{aligned}\dot{\mathbf{x}}_p(t) &= \mathbf{f}_p(\mathbf{x}_p, \mathbf{u}_p) \\ \mathbf{y}_p(t) &= \mathbf{g}_p(\mathbf{x}_p),\end{aligned}\tag{4.28}$$

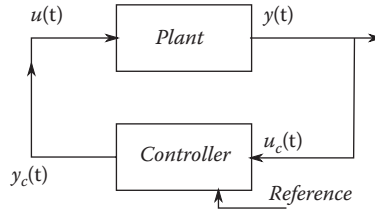


FIGURE 4.22 Continuous-time feedback control system.

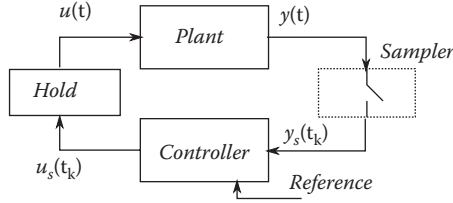


FIGURE 4.23 Sampled data control system.

for the plant, and

$$\begin{aligned}\dot{\mathbf{x}}_c(t) &= \mathbf{f}_c(\mathbf{x}_c, \mathbf{u}_c, \mathbf{u}_{\text{ref}}) \\ \mathbf{y}_c(t) &= \mathbf{g}_c(\mathbf{x}_c, \mathbf{u}_c),\end{aligned}\tag{4.29}$$

for the controller, and ideally, the feedback connection is expressed by the laws $\mathbf{u}_c(t) = \mathbf{y}_p(t)$, $\mathbf{u}_p(t) = \mathbf{y}_c(t)$.

Since building a real system that accomplishes Equation 4.29 may be technically very difficult (if not impossible) to realize, continuous-time controllers are usually replaced by digital devices. This idea defines the so-called *sampled data control systems*, depicted in Figure 4.23. Here, the controller observes a discrete-time law that can be obtained in two ways: (i) The design can be done directly in discrete time starting from a discrete-time equivalent model of the plant (*direct design*), and (ii) the discrete controller represents an approximation of a previously designed continuous-time controller (*digital redesign* [19] or *emulation*).

In many applications, particularly in the presence of nonlinear plants, the latter alternative is usually preferred. Emulation is then performed using a discrete-time approximation of Equation 4.29 obtained using a classical numerical ODE solver. Typically Forward Euler is being used, but also versions of Backward Euler and the Trapezoidal Rule (Tustin's method [8]) can be encountered, especially in the case of linear control laws.

Some drawbacks in this type of digital emulation of continuous-time controllers are the following:

- The stability of the closed-loop scheme depends on the sampling period.
- It is very difficult and sometimes even impossible to ensure stability in nonlinear plants.

- It is difficult to analyze, at design time, the side effects of nonideal components, such as the quantization introduced by D/A and A/D conversions, delays, and so on.
- When the plant and the controller are connected using a communication network, there are fundamental limitations in both bandwidth and stability related to the amount of information that must be transmitted across the network.

How can DEVS and the QSS methods help in this context? First, the QSS methods offer an alternative discretization approach. The controller of Equation 4.29 can be replaced by its QSS approximation (i.e., a DEVS model):

$$\begin{aligned}\dot{\mathbf{x}}_c(t) &= \mathbf{f}_c(\mathbf{q}_c, \mathbf{u}_c, \mathbf{u}_{\text{ref}}) \\ \mathbf{y}_c(t) &= \mathbf{g}_c(\mathbf{q}_c, \mathbf{u}_c)\end{aligned}\quad (4.30)$$

Second, taking into account the asynchronous nature of the QSS approximation, we can replace the regular sampling by an *event driven* sampling, where samples are taken only when the difference between the continuous signal and the sampled one differ by a prespecified quantity, the *quantum*. This sampling strategy, referred to as *asynchronous sampling* or *level-crossing sampling*, has an input-output behavior equivalent to that of a hysteretic quantizer (see Figure 4.6).

The resulting scheme, depicted in Figure 4.24, is called Quantized State Control (QSC) [14]. In this fashion, the control system lacks a centralized synchronous time discretization scheme. Instead, the control system is dealing with state, input, and output quantizations. Thus just as in the case of the QSS methods, the effects of the discretization can be studied by interpreting it as a bounded perturbation around the original continuous-time control system.

The QSC emulation of a continuous-time controller ensures the following properties:

- If a continuous LTI closed-loop system is stable, its QSC emulation is *practically* stable; that is, the trajectories finish inside a bounded region that can be computed a priori [14].

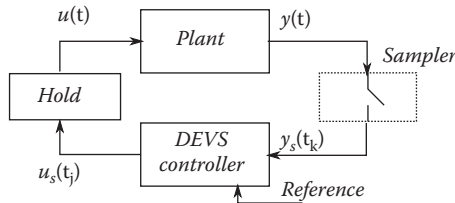


FIGURE 4.24 Quantized state control system.

- The above property holds also for nonlinear and time-varying systems. In these cases, a Lyapunov analysis allows computing the largest quantum that ensures convergence to a prespecified bounded region [12].
- Most nonideal characteristics of components, except for delays, can be easily incorporated into the model for analyzing their effects.
- Communication constraints are drastically reduced, as the scheme works by transmitting a single bit per sample (because the difference between consecutive samples is always $\pm \Delta Q$) [15].

The aforementioned real-time version of PowerDEVS allows easy implementation of QSC strategies on normal PCs. All that needs to be done is to draw the block diagram representation of the continuous-time controller, to select an appropriate quantum for each integrator, and to add an interrupt detector block for the controller input and a synchronizer block for the controller output.

The most demanding task related to QSC implementation is that of finding or designing appropriate asynchronous samplers. Yet, there exist some classes of sensors, such as position encoders, which are naturally asynchronous. Figure 4.25 shows an example of the QSC implementation of a PI controller for angular velocity control of a DC motor in PowerDEVS.

The motor moves an encoder that causes interrupts in the PC parallel port each time the motor rotates by a small angle. Whenever a hardware interrupt is detected, the block *IRQ7 Detector* provokes an output event. The *speed estimator* block counts the number of events it receives per time unit, computing in that way an estimation of the motor speed. That estimation is compared with the reference speed, and the error is being integrated, using a QSS approximation, to compute a control action with a PI strategy. Finally, the control action is coded with a Pulse Width Modulation (PWM) strategy, and the resulting value, “on” or “off,” is sent to a bit of the parallel port. That bit commands a switch (transistor) that turns the motor feeding voltage on and off. Figure 4.26 shows the speed as measured by the speed estimator block.

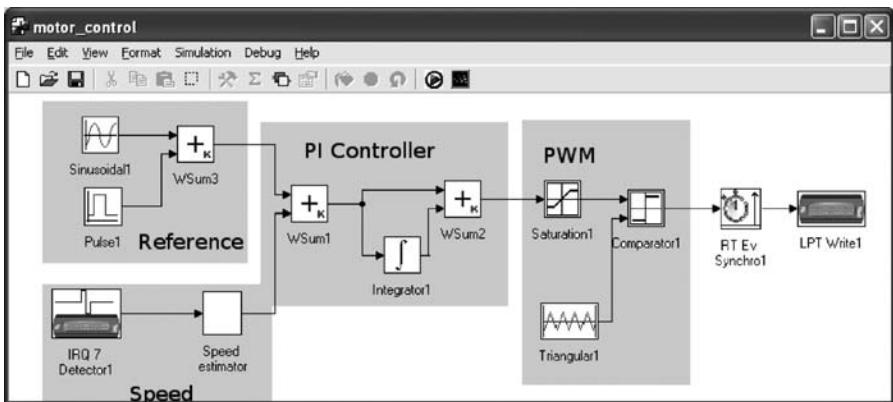


FIGURE 4.25 Quantized state control of a DC motor in PowerDEVS.

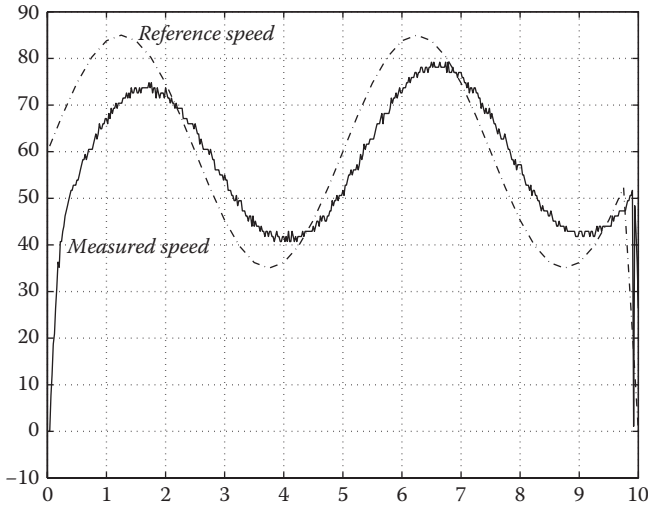


FIGURE 4.26 Reference and measured speed of a DC motor with QSC control.

4.5 CONCLUSIONS

In this chapter, we have shown that continuous-time systems described by sets of ODEs can be conveniently converted to equivalent discrete-event descriptions under the DEVS formalism. It was shown that the numerical properties (stability and accuracy) of these approximations can be analyzed as rigorously as is the case with the classical numerical ODE solvers. However, the resulting QSS simulators exhibit features that make them superior to the classical numerical ODE solvers in several respects. It was shown that QSS simulators can deal with discontinuities in model descriptions effectively and efficiently, and that QSS simulators can be designed that can deal successfully with stiff and marginally stable systems. The chapter ended with an outlook to an important application of quantized state methodology; namely, the systematic design of digital controllers for continuous plants using a new approach to digital redesign involving a QSC architecture.

ACKNOWLEDGMENT

The research described in this chapter was partially supported by the Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) of Argentina under grant PIP 2009-2011 0138.

REFERENCES

1. Beltrame, T. Design and Development of a Dymola/Modelica Library for Discrete Event-Oriented Systems Using DEVS Methodology. Master's Thesis, ETH Zurich, Zurich, Switzerland, 2006.

2. Bergero, F. and Kofman, E. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 2010. in press.
3. Bortoloto, M., Kofman, E., and Migoni, G. Métodos de Integración por Cuantificación en Sistemas Hamiltonianos. *Proceedings of AADECA 2008*, AADECA, Buenos Aires, Argentina, 2008.
4. Cellier, F. E., and Kofman, E. *Continuous System Simulation*. New York: Springer-Verlag, 2006.
5. Cellier, F. E., Kofman, E., Migoni, G., and Bortolotto, M. Quantized State System Simulation. *Proceedings of the 2008 Summer Computer Simulation Conference*, SCS, San Diego, CA, 2008.
6. D'Abreu, M., and Wainer, G. M/CD++: Modeling Continuous Systems Using Modelica and DEVS. *Proceedings of MASCOTS 2005*, 229–38, IEEE, New York, 2005.
7. Esquembre, F. Easy Java Simulations: A Software Tool to Create Scientific Simulations in Java. *Computer Physics Communications* 156, no. 1 (2004): 199–204.
8. Franklin, G. F., Powell, J. D., and Workman, M. *Digital Control of Dynamic Systems*, 3rd ed. Menlo Park, CA: Addison-Wesley, 1998.
9. Hairer, E., Lubich, C., and Wanner, G. *Geometric Numerical Integration Structure-Preserving Algorithms for Ordinary Differential Equations*. New York: Springer-Verlag, 2002.
10. Kofman, E. A Second-Order Approximation for DEVS Simulation of Continuous Systems. *Simulation* 78, no. 2 (2002): 76–89.
11. Kofman, E. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research* 36, no. 2 (2006): 101–8.
12. Kofman, E. Discrete Event Control of Time Varying Plants. *Latin American Applied Research* 35, no. 2 (2005): 135–41.
13. Kofman, E. Discrete Event Simulation of Hybrid Systems. *SIAM J. Scientific Computing* 25, no. 5 (2004): 1771–97.
14. Kofman, E. Quantized-State Control. A Method for Discrete Event Control of Continuous Systems. *Latin American Applied Research* 33, no. 4 (2003): 399–406.
15. Kofman, E., and Braslavsky, J. Level Crossing Sampling in Feedback Stabilization under Data-Rate Constraints. *Proceedings of CDC'06, IEEE Conference on Decision and Control*, 4423–28, San Diego, CA. New York: IEEE, 2006.
16. Kofman, E., and Junco, S. Quantized State Systems: A DEVS Approach for Continuous System Simulation. *Transactions of SCS* 18, no. 3 (2001): 123–32.
17. Kofman, E., Lee, J. S., and Zeigler, B. P. DEVS Representation of Differential Equation Systems: Review of Recent Advances. *Proceedings of ESS'01, 13th European Simulation Symposium*, 591–95, SCS, San Diego, CA, 2001.
18. Kofman, E., and Zeigler, B. DEVS Simulation of Marginally Stable Systems. *Proceedings of IMACS'05*, IMACS, New Brunswick, NJ, 2005.
19. Kuo, B. C. *Digital Control Systems*, 2nd ed. Fort Worth, TX: Saunders College Publishing, 1992.
20. Migoni, G. and Kofman, E. Linearly Implicit Discrete Event Methods for Stiff ODEs. *Latin American Applied Research*, 39 no.3 (2009): 245–254.
21. Migoni, G., Kofman, E., and Cellier, F. Integración por Cuantificación de Sistemas Stiff. *Revista Iberoamericana de Automática e Informática Industrial* 4, no. 3 (2007): 97–106.
22. Nutaro, J., and Zeigler, B. On the Stability and Performance of Discrete Event Methods for Simulating Continuous Systems. *J. Computational Physics* 227, no. 1 (2007): 797–819.

23. Quesnel, G., Duboz, R., Ramat, E., and Traoré, M. VLE: A Multi-modeling and Simulation Environment. *Proceedings of the 2007 Summer Computer Simulation Conference*, 367–74, SCS, San Diego, CA, 2007.
24. Zeigler, B. P. and Lee, J. S. Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment. *Enabling Technology for Simulation Science(II)*, SPIE AeroSense 98, Bellingham, WA.
25. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd ed. New York: Academic Press, 2000.
26. The MathWorks Inc., Simulink User's Guide, Natick, Massachusetts: The MathWorks Inc., 2006.

Section II

Methodology and Tools

5 Heterogeneous Model Composability

Hessam S. Sarjoughian and Gary R. Mayer

CONTENTS

5.1	Introduction	111
5.2	Multilayer Modeling	113
5.2.1	Example: Supply-Chain Modeling	114
5.2.2	Example: Agent–Environment Modeling	116
5.3	A Taxonomy of Multi-Formalism Model Composability Approaches.....	118
5.3.1	Modeling Formalisms.....	118
5.3.2	Mono-Formalism Model Composability	121
5.3.3	Super-Formalism Model Composability	121
5.3.4	Meta-Formalism Model Composability	122
5.3.5	Poly-Formalism Model Composability	123
5.3.6	Model Composition With Adapters	124
5.4	Model Specification and Execution Algorithm Separation	125
5.4.1	Distributed Execution of Composable Models	126
5.4.2	Simulation Data Visualization.....	127
5.5	Application Domain Modeling.....	128
5.5.1	Modeling Interactions between Discrete Process and Predictive Control Models	129
5.5.2	Agent–Environment Interaction	131
5.6	Conclusions	134
	References.....	135

5.1 INTRODUCTION

Many contemporary systems are integrated from subsystems that have simple to complex structures and behaviors. Given the diversity of parts and resulting complexity of the systems, criticality of operation, and enormous financial consequences, simulation is being used more and more as the primary science and technology to aid analysis, design, implementation, and testing. Indeed, simulation can be used across each phase of system development as proposed in the Simulation-Based Acquisition [1]. For example, a suite of system-level simulation models may be developed to help analyze requirements and evaluate potential architectural solutions far in advance of defining detailed design specifications. Yet, another suite of models may be used to

help develop detailed design specifications that can be implemented. The purposes for these systems vary significantly, as each is aimed at satisfying particular goals.

To understand and imitate the intricate dynamics of the above systems, it is often necessary to use different kinds of models to describe their subsystems. Heterogeneous model types offer greater capability in terms of describing distinct aspects and interactions of systems, as opposed to homogenous model types. However, combining different model types pose a variety of challenges, depending on the system being modeled (e.g., Refs [2–7]). It is also possible to use software design ideas and programming concepts to integrate models (e.g., Refs [8,9]).

Decomposition and composition of models is challenging when models are heterogeneous in terms of their formal specifications. More specifically, a modeling formalism can be defined to consist of a model specification and execution algorithm; thus making a strong separation between the concepts of model composability and simulation interoperability [10]. The use of modeling formalism in handling model composability is distinct from the use of a software-centric concept, design, and implementation to integrate simulation models (e.g., Ref. [8]). The model specification and execution algorithm together form a mathematical basis to describe the structure and behavior of systems. Systems theory, logical processes, and object-orientation worldviews are well-known approaches for describing dynamic systems [11–16]. Ultimately, different modeling paradigms are suitable for different kinds of needs. For example, a discrete-event modeling approach may be used to describe manufacturing processes and their interactions as a collection of model components. Alternatively, given historical data, a continuous model can be developed to show how inventory holding varies in relation to factory throughput. Optimization models may be developed to understand logistics and the financial impact of satisfying customer demand.

Supply-chain and agent–environment processes are often modeled at varying levels of resolution and from multiple aspects. Depending on a system’s different behaviors, system simulation models may be developed using a monolithic modeling paradigm in one extreme and a combination of distinct modeling paradigms at the other extreme. For example, a manufacturing process may be modeled using a single modeling approach, for example, Discrete Event System Specification (DEVS) [16] or Petri Net [17]. However, to effectively design, assess, and operate a manufacturing supply-chain in an optimal fashion, it is also necessary to use optimization models [18]. Similarly, an agent–environment system could be modeled using Cellular Automata [19] or DEVS. For example, this kind of system may be described using the Geographic Information System (GIS) [20] concepts and implemented using a tool such as Geographic Resources Analysis Support System (GRASS) [21]. Important benefits (e.g., efficiency and large data set visualization tools) can be gained for simulating very large landscape models with support for simple agent modeling.

In the remainder, basic concepts describing approaches to the heterogeneous model composition is developed. Awareness of these methods aids in the choice of a modeling approach based upon factors such as the domain under study, the extent of the disparity between subsystem models, and the available model development frameworks. With this goal in mind, in Section 5.2 the basic concepts and methods important for separately modeling different aspects of a system are reviewed. In

Section 5.3, the proposed model composability classification based on multiple, distinct modeling formalisms is developed. In Section 5.4, the role of composing heterogeneous models in relation to the interoperation of different execution algorithms and data visualization is described. In Section 5.5, the use of the poly-formalism model composability is demonstrated for two different example domains. In Section 5.6, a summary of the chapter with focus on the impact of the multi-formalism model composability taxonomy is presented.

5.2 MULTILAYER MODELING

Many real-world engineered and natural systems can be abstracted to consist of two layers (or subsystems): one is a *plant* and the other is a *controller* (see Figure 5.1). At a high level of abstraction, the plant and controller models of a system can be considered computations that exchange data (i.e., states) and control (i.e., commands) under some well-defined constraints. The plant/controller pair can form a symbiotic relationship, where one is concerned with operations and the other is concerned with control of the operations. The plant dynamics can be described in a variety of forms—for example, discrete, continuous, or some combination thereof. The controller can be based on optimization and fuzzy control. For example, a plant can have stochastic discrete-event operations taking place at the present time and the controller can be an optimization-based feedback control. The plant’s operations occur from one time instant to the next; that is, the dynamics of the plant can be modeled as deterministic (or stochastic), continuous, and discrete equations. The controller operations can be based on the current or future state of the plant as well as the present and expected inputs that can affect controller’s decision making.

The plant and controller are considered a two-layer system—plant and controller are at lower and higher levels of abstractions, respectively. The plant is responsible for carrying out low-level operations, some of which are requested from the controller. It carries out some operations given some inputs and generates some outputs. The controller responsibility is carrying out higher level operations given details provided from the plant. Its other responsibility is to constrain the operations of the

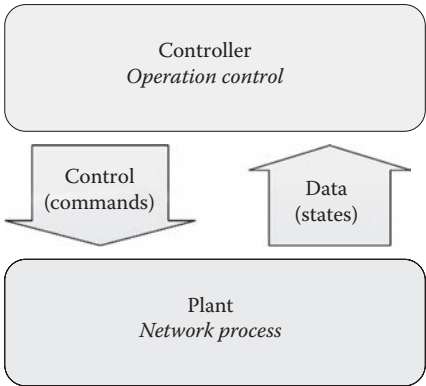


FIGURE 5.1 A two-level plant and control model.

plant to those that are deemed desirable or acceptable for some finite time period in the future.

The controller itself can be viewed as the plant and its operations sanctioned by another controller. This leads to multilayer plant/controller specifications. This separation offers fundamental benefits in terms of developing separate and hybrid models and simulations. It provides conceptual separation of knowledge. Plant and controller models play distinct roles where one supplies data and the other provides control. It also provides support for multiple levels of control for a plant. For instance, a tactical controller provides short-term commands to the plant, and a strategic controller provides long-term commands to the tactical controller. Another advantage is that alternative modeling choices may be used for the plant or controller. For example, the plant may be described in terms of discrete and continuous models and the controller may be described in terms of linear optimization or event-based control models.

The remainder of this section focuses on two classes of problems: engineered (supply-chain) and natural (agent–environment) systems. In the case of a semiconductor manufacturing supply chain, the plant is the *manufacturing process* and the control is the *operational/logistics* controller (see [Figure 5.1](#)). In the agent–environment system, the role of controller and plant is dependent upon the domain under study. An agent-centric focus will devise the agent as the controller, and the opposite is true of an environmentally focused domain. Alternatively, in system models where both agent and environment are robust models of their respective systems, the agent and environment may swap controller–plant roles during the simulation.

5.2.1 EXAMPLE: SUPPLY-CHAIN MODELING

In the domain of semiconductor manufacturing supply-chain systems, models of discrete processes, control policies, and decision plans are important in handling stochastic dynamics of individual parts of manufacturing processes under tactical control and strategic planning, given short- and long-term supply and customer demand [18]. These kinds of aggregated models play a central role in the understanding of not only physical operations of processes, but also how they are managed using tactical control and strategic planning. Such complex industries can gain both short-term and long-term technical competitiveness as well as financial advantages. Another crucial benefit of synthesizing different kinds of models is the ability to better handle mismatches (between competing objectives arising from operational vs. decision aspects, for example).

A canonical manufacturing process can be modeled as a collection of inventory, factory, and transportation nodes. The inventory (and transportation) nodes store (transport) products such as raw material, semifinished goods, and finished goods. The factory node processes the products it receives from inventories and sends the processed products to inventory or transportation nodes after some time period. Together, these nodes characterize the state of the supply-chain process at any one time instantly. Each factory, inventory, and transportation node can have stochastic yield and duration which, in turn, results in a chain of nodes that can exhibit complex dynamics.

A well-known and common approach for controlling a supply-chain is to use Linear Programming (LP) [22] for optimization. An optimization model describes a set of formulas and relationships (constraints c_1, c_2 , etc.). A solver can be used to optimize the optimization model's variables of interests (e.g., inventory holdings) given some objective functions. Some examples of decisions can be how much should be held in the Finished Goods inventory, how many of which products should a factory produce, and what transportation mode to use for shipping products from an inventory to the customer (see Figure 5.2). The responsibility of the controller is to optimize expected versus actual demand and supply over some future time horizon given key data about manufacturing nodes (e.g., work-in-progress in a Finish node) [23]. Alternatively, it may be useful to use a combination of optimization constraints and control-theoretic feedback/feedforward filters that are known as Model Predictive Control (MPC) [24–26].

Integrating these models requires accounting for differences between data types that are used in, for example, DEVS and LP models. The structure of input and output data for DEVS models is complex as compared with the data for LP models. Data for DEVS models developed in DEVSJAVA [27] is specified in terms of messages, each defined in terms of port and value pairs—the port is a string and the data can be an arbitrary expression that usually evaluates to a simple value (e.g., double, integer, and string). The structure of input and output data for LP models is a simple data type that may be formed into vectors (e.g., array of integers or strings). Therefore, for the

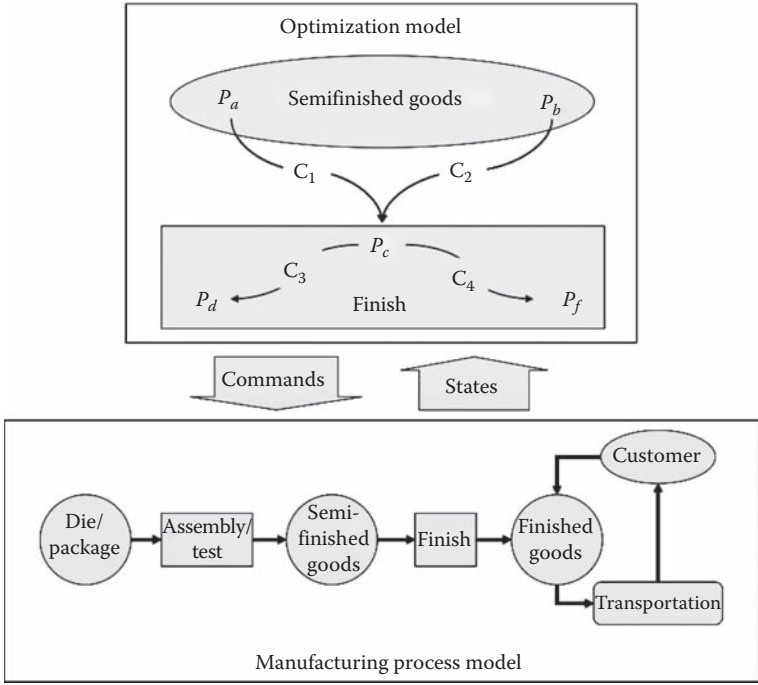


FIGURE 5.2 Conceptual model of a semiconductor manufacturing process with an optimization control.

manufacturing process and optimization models to interact, their data types need to be syntactically mapped from one to the other [23,25]. For example, the Finish process produces a set of finished products Lots at output port Data_Out. The number of tested products is provided as input to the LP model.

Aside from input and output exchanges and mappings, it is also necessary for the behaviors of the manufacturing process and optimization models to be well-defined—data exchanges need to occur at appropriate times and frequency. For example, the manufacturing process model advances from day to day whereas the optimization model is solved at the start of each day. In this scenario, a command determined by the optimization model requests n products from the Semifinished Goods to be released into Finish; that is, Finish receives a release command on port Control_In with value n . Details of the DEVS, LP, and MPC models partially shown Figure 5.2 can be found in [25,26,28–31].

5.2.2 EXAMPLE: AGENT–ENVIRONMENT MODELING

Agent–environment models have a broad applicability across multiple disciplines (e.g., social science [32] and defense [33]). The “agent” facet of this domain is an entity whose logic may range from simple to complex [32,34]. In many cases, the agent has the capability to obtain knowledge about the environment, make decisions based upon goals, and effect change in the environment in order to meet its goals. In more simplistic cases, the agent has only rudimentary behavior and is there to force changes within the environment, which may have dynamics driven by functional relations (e.g., ordinary differential equations, partial differential equations, and difference equations). Alternatively, the environment is there simply as a vehicle for the agent dynamics. In more complex cases, both the agent and the environment are capable of complex dynamics without the other, and share a bidirectional interaction. This bidirectional interaction may create additional complexity in that the changes affected by one on the other, may reflect back. The models chosen to represent these systems will dictate in part how expressive each model, and the models’ interaction, can be.

Discrete models can easily represent both facets of an agent–environment system model for many applications within a domain. An agent model, depending on its activities, may be discrete-event or discrete-time. Either may represent the time step process of evaluating the environment, making decisions, handling failure, and affecting change in the environment. Cellular automata (CA) are useful as model representations of the environment. First, CA provide a common abstraction of an environment’s spatial properties into a regular tessellation of spatial entities with an inherent spatial relation. Second, this same abstraction provides an easy mechanism from which the location of an agent in its environment may be discussed and used to facilitate the dynamics of the system [35]. Figure 5.3 shows an example of a rules-based agent and a cellular automaton environment system model.

The development environment and integration of these models is heavily dependent upon the domain, answers sought from the simulated models, and the modeler’s experience. Thus, some development environments are better suited than others. For example, a model of a moderate number of agents and an environment tessellated to

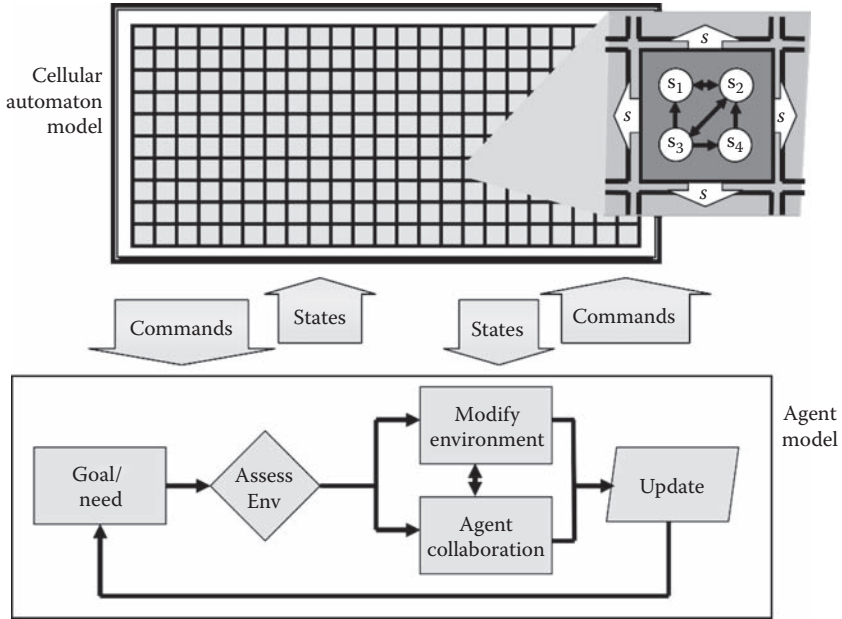


FIGURE 5.3 Conceptual model of an agent–environment model.

only a few hundred cells can easily be handled using a DEVS/Cell-DEVS approach (e.g., Refs [36,37]). However, if the environment is abstracted to multiple millions of cells, a GIS development environment may be the better choice (e.g., Ref. [38]). This is because the GIS tools are designed around handling large datasets with geospatial relationships, many scientists who study large environmental regions are already familiar with the tools, and most GIS development environments come with a visualization tool that allows a modeler to get a visual overview of simulation results from large datasets. Unlike DEVS, the GIS development environments are not derived from a system-theoretical formalism. However, a CA system specification that accounts for input and output from a CA can be represented within these environments in order to facilitate a formal interaction between the agent and environment models. One such CA specification is the Composable Cellular Automata (CCA) [35].

When a DEVS agent model and a CCA environment model are made to interact, there are a number of aspects that must be considered. First, the timing of the models may require synchronization. For example, an agent model may operate on an annual cycle in simulation time while the environment model updates once per decade. Or, if the agent model is discrete-event, an output event from the agent model must be managed to coincide with the environment model's discrete-time segments. Additionally, there is a spatial resolution abstraction. An example of this is an agent operating in an area of 100 m² while the environmental model is tessellated to regions of 25 m². Lastly, some of the interactions between the agent and the environment may require data from both models and, furthermore, require representation in both. For example, a farming village leaves a footprint on the environment that is dependent upon the number of villagers. This footprint limits where farmers may

plant crops and impacts foliage growth and erosion in the surrounding areas. Both models provide data to this interaction and require knowledge of different aspects of the impact caused by the village.

5.3 A TAXONOMY OF MULTI-FORMALISM MODEL COMPOSABILITY APPROACHES

Considering the multilayer modeling concept, it is possible to use a single modeling formalism to describe a plant and its controller. However, if the system under consideration has parts with dynamics that are intrinsically different, it is crucial to use multiple modeling formalisms. The above semiconductor manufacturing supply-chain system is an example where the plant (manufacturing process) is suitable to be modeled as the discrete-event model and the controller (logistic control) as an optimization model. A key difference between these models is that the discrete-event model describes how nodes of a manufacturing process network affect one another and the optimization model searches for optimal operation of the overall manufacturing process. While it may have been possible to use a single formalism—discrete-event, for example—the loss of specificity within the domain, in terms of domain expertise application and model abstraction, may be too great. Thus, a multi-formalism application, and an accompanying composability approach, must be considered.

5.3.1 MODELING FORMALISMS

A modeling formalism can be defined to consist of two parts: *model specification* and *execution algorithm* [10]. The former is a mathematical theory describing the structure and behavior that can be described with it. The latter specifies an algorithm that can correctly execute any model that is described in accordance with the model specification. A model A that can be specified in a modeling formalism Ψ is denoted as $M_{A, \{\Psi\}}$; that is, model A adheres to the model specification Ψ and can be executed using its execution algorithm. A model composed of a finite number of disparate models (e.g., $A, B, \dots, K$) specified in a finite number of distinct modeling formalisms (e.g., $\Psi, \Phi, \dots, \Omega$) is denoted as $M_{\cup A, B, \dots, K, \{\Psi, \Phi, \dots, \Omega\}}$.

Approaches to model composability are classified into *mono-*, *super-*, *meta-*, and *poly-formalism* [39]. These approaches provide different degrees of capabilities toward model composition. The first two are grounded in the concept that, principally, a single formalism is well suited for modeling all of the different parts of a system. In contrast, the latter two are aimed at the cases where disparate modeling formalisms are necessary to describe the parts of a complex system. Despite the differences among model composability approaches, every approach must ensure that the interactions among composed model components are structurally and behaviorally consistent (i.e., not only do data and control successfully pass between the models, but semantically the interaction makes sense within the domain and within the confines of each model's respective formalism and implementation).

Every model composability approach must ensure that the interactions among composed models are correct. The exchange of data and control, causality of input and output interactions, and synchronization of models' executions with respect to

time must be well defined. This not only requires that individual model specifications are executed correctly, but also their compositions (i.e., the execution of multiple execution algorithms are well defined given their distinct model specifications).

To help describe the four model composability approaches, model specifications are shown as rectangle, rounded rectangle, oval, and hexagon icons (see [Figures 5.4 through 5.7](#)). Process and control models are shown as rounded rectangles and ovals, respectively. A composite model that consists of two or more models (whether specified in one or multiple modeling formalisms) is shown as a rectangle. For example, Figure 5.4 shows that the manufacturing and control models as well as their composition are described in Ψ . A Knowledge Interchange

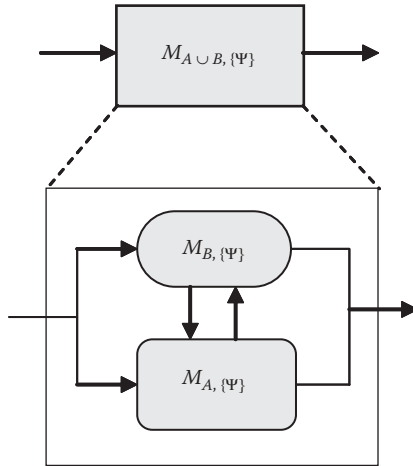


FIGURE 5.4 Mono-formalism model composability.

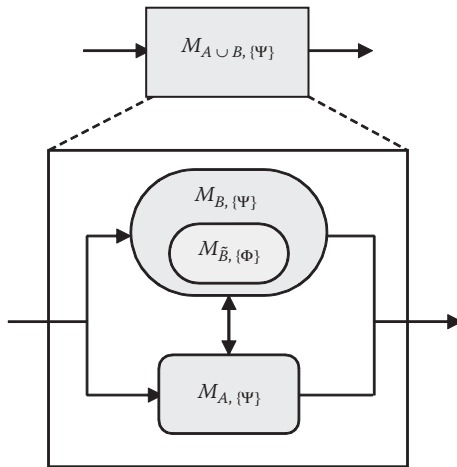


FIGURE 5.5 Super-formalism model composability.

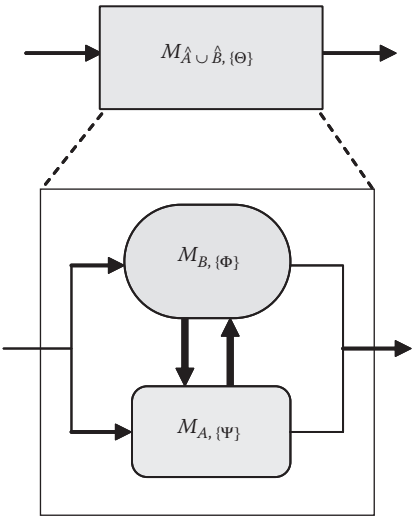


FIGURE 5.6 Meta-formalism model composability.

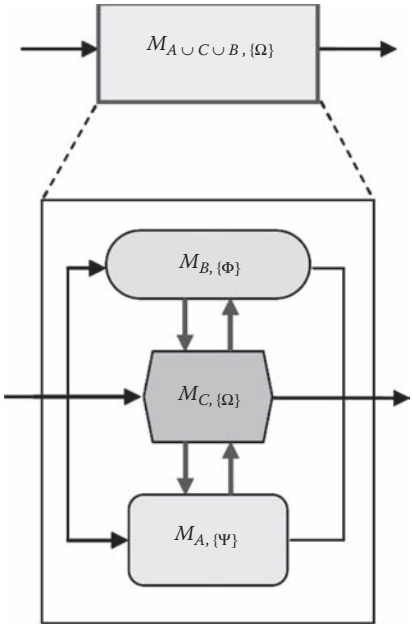


FIGURE 5.7 Poly-formalism model composability.

Broker [40] model that composes multiple models specified in multiple modeling formalisms is shown as a hexagon (see Figure 5.7).

A modeling formalism can have several implementations (i.e., software realizations) based on the choice of programming languages and implementation techniques—for example, a mathematical model can be designed and implemented using

object-oriented modeling concepts and a programming language. Software realizations of a modeling formalism can also be affected by the intended or expected modeling and simulation applications (e.g., parallel/distributed simulation for large-scale application domains). In addition, entity relations (Entity Relation [ER] models) and the XML family of models are also referred to as models. These are not considered here since (i) such models are primarily aimed at describing structure of data instead of a system's structure and the behavior and (ii) they can be subsumed by component-based and object-oriented modeling approaches.

Before proceeding further, it is useful to note that model specifications are also described and implemented using (high-level) programming languages. Therefore, a model specification can be referred to as a software model specification or a mathematical model specification. An example of the mathematical specification for Parallel DEVS atomic and coupled specifications can be found in Zeigler, Praehofer, and Kim [16]. It is also helpful to note that different terms are used for execution algorithms. An algorithm that is void of software design choices and implementations is sometimes referred to as an abstract simulator or solver.

5.3.2 MONO-FORMALISM MODEL COMPOSABILITY

It is common to use a single modeling formalism to specify one aspect of a system [41]. There exist numerous mono-modeling formalisms such as differential equations and cellular automata. Use of a mono-modeling approach provides key advantages—many suitable methods have been developed that support decomposition (or hierarchical composition) of a model into (from) parts. Modeling formalisms enable modelers to specify structure and behavior of dynamical systems given well-defined syntax and semantics. For example, discrete-event modeling is often used to specify discrete processes. A mono-modeling formalism can also be used to specify different aspects of a system—detailed process flow dynamics with simplified event-based control. With a monolithic modeling formalism, models for different parts of a system adhere to a single structure and behavior specification. That is, $M_{A \cup B, \{\Psi\}}$ may be used to model the discrete-event process model A , event-based control model B , and their interactions using modeling formalism Ψ . For the semiconductor manufacturing supply-chain systems, the manufacturing process and control models are identified as A and B (see Figure 5.4). These models can be specified in the DEVS formalism [16] (shown as Ψ in Figure 5.4). The basic advantage of a mono-modeling formalism is that the manufacturing and control models as well as their interactions are modeled within one theory. This approach eliminates integration of disparate model types and avoids having to integrate different simulation tools.

5.3.3 SUPER-FORMALISM MODEL COMPOSABILITY

Use of mono-formalism model composability, however, may not be suitable if the models that are to be composed are intrinsically different—these models are best described in different modeling paradigms. This is because a modeling formalism is suitable for describing only some parts of a complex, heterogeneous system while all the remaining parts must either be abstracted away or formulated within other

modeling formalisms. To overcome this difficulty, super-formalism model composability may be used as shown in Figure 5.5.

In this approach, a modeling formalism ($M_{\tilde{B},\{\Phi\}}$) is encapsulated within a super formalism $M_{A \cup B, \{\Psi\}}$. Models $\tilde{B}$ and B have distinct syntax and semantics, and the former must be encapsulated inside the latter. This allows the specification of the interactions between models A and B to be described using the modeling formalism Ψ . This approach, however, requires the modeling formalisms $\tilde{B}$ and B to have a well-defined relationship with respect to one another. The syntax and semantics of the super-formalism model composition must encapsulate disparate model types, but also support the specification of their interactions. Model encapsulation (e.g., enclosing $\tilde{B}$ inside B) requires the encapsulated model to be well defined. In the semiconductor manufacturing supply-chain example, a logistic controller can be a linear optimization model $\tilde{B}$ in formalism Φ (i.e., LP) and B can be a discrete-event model in formalism Ψ (i.e., DEVS). In the case of the discrete part supply-chain systems, we must ensure both (i) the input/output structure and behavior for the encapsulated optimization and discrete-part models and (ii) the interactions between the optimization and process models can be fully specified within the Ψ modeling formalism. This enables composition of the plant and the controller models to be specified with the DEVS formalism, for example.

The super-formalism model composition can be used to specify different kinds of models at one or multiple levels of abstraction (input/output, state-based, and hierarchical) [16]. Composing models that are specified at multiple levels of abstraction can be said to be a *strong* form of super-formalism. For example, with DEV&DESS [42], state-based discrete-time and continuous models may be composed. It is also possible for a state-based model to be composed with an input/output model, a *weak* form of super-formalism [43]. The Ptolemy project [44] formalizes input/output interactions among actors (model components) under the control of directors for mixed signal systems using Interface Automata, a variant of the automata model [45].

5.3.4 META-FORMALISM MODEL COMPOSABILITY

A model can also be composed from models that are described according to two or more modeling formalisms. In the meta-formalism model composability approach, different types of model specifications are transformed or abstracted to another modeling formalism. This approach to model composability must be able to account for the differences between disparate model specifications that can be composed.

As depicted in Figure 5.6, $M_{\hat{A} \cup \hat{B}, \{\Theta\}}$ is a composition of models $\hat{A}$ and $\hat{B}$ with their interactions specified in formalism Θ . Here models A and B specified in Ψ and Φ are mapped to $\hat{A}$ and $\hat{B}$. This requires $\Psi \rightarrow \Theta$ and $\Phi \rightarrow \Theta$. The transformation $\rightarrow$ defines structure and behavior of formalism Ψ and Φ to that of the formalism Θ ; that is, the interactions between models A and B are specified in terms of $\hat{A}$ and $\hat{B}$.

A standardized modeling approach representative of Figure 5.6 is the High Level Architecture (HLA) [46–48]. It provides a suite of generalized services to which different simulation models are mapped. This approach is strongly aimed at handling interoperability needs with some limited capability for model composability as supported by an Object Model Template [49]. The interaction of different types of model

specifications is supported with the publish/subscribe technique and data management service while handling of timed interactive behavior of execution algorithms is the responsibility of the time management service.

Another approach is to use meta modeling and model transformation [50,51]. It allows the determination of whether or not two models described in different modeling formalisms can be transformed completely or partially to their counterparts in a meta-modeling formalism. In some cases, a model described in Ψ can be transformed into a model Φ (without any loss); whereas in other cases, some constraints must be defined and placed on models described in Ψ to be represented in Φ . For example, a discrete-time model specified in Discrete Time System Specification and a discrete-event model specified in Petri Nets can be completely transformed into models described in DEVS [16].

Given the semiconductor manufacturing supply-chain system, its manufacturing process and logistic control can be specified in the DEVS and LP modeling formalisms (denoted as Ψ and Φ in Figure 5.6). The composition of the meta models of A and B is denoted as $M_{\hat{A} \cup \hat{B}, \{\Theta\}}$. The composite (or federated) DEVS and LP models may be specified in High Level Architecture (HLA; denoted as Θ in Figure 5.6). Using the object-oriented concepts and methods, a set of federates (which are primarily simulation models with I/O specification) can be composed to create a federation.

5.3.5 POLY-FORMALISM MODEL COMPOSABILITY

The remaining approach, called poly-formalism modeling, composes disparate models using a model referred to as Knowledge Interchange Broker (KIB) [40]. The KIB handles the differences between disparate modeling formalisms. Specifically, the KIB formalizes the interactions between two disparate composed models as a separate model in terms of input/output mappings, synchronization, concurrency, and timing properties [10]. The basic concept and formulation of the KIB was developed in the context of a simple intelligent transportation system [10]. As shown in Figure 5.7, $M_{A \cup C \cup B, \{\Omega\}}$ is a composition of $M_{A, \{\Psi\}}$ and $M_{B, \{\Phi\}}$ using $M_{C, \{\Omega\}}$. The interactions between models A and B are specified in Ω , the formalism for KIB.

One of the aims of the poly-formalism model composition is to keep the composed models independent. The key benefit is describing the composed models as rich as possible with their modeling formalisms and, in addition, modeling separately (and thus flexibly) interactions between two disparate models. Reuse is supported at the level of modeling formalisms and domain knowledge can be partitioned for an application problem based on the separation of the composed models and their interactions. The KIB approach has been developed in a realistic semiconductor manufacturing supply-chain system; that is, detailed models described in the DEVS, LP, and MPC modeling formalisms are developed and composed with $KIB_{DEVS/LP}$ [10,23,28] and $KIB_{DEVS/MPC}$ [10,29,30].

With poly-formalism model composability, there is no need to transform two or more models to be composed into other models, all of which conform to a single formalism. Furthermore, poly-formalism model composability is distinct from the strong form of super-formalism model composability as KIB supports direct formulation of the interactions (data and control exchanges) among different model types

(e.g., discrete-event and linear optimization), and also supports different kinds of data transformation and control schemes that are described external to the models that are composed. Using the poly-formalism model composition, common forms of data transformation (i.e., aggregation and disaggregation) are inherently supported. Furthermore, it supports alternative forms of common control schemes (i.e., sequential, synchronous, and asynchronous). Specification of the time-based data transformations and controls need to be defined based on the specifics of an application domain for which the simulation modeling is desired.

5.3.6 MODEL COMPOSITION WITH ADAPTERS

An important practical distinction between the meta- and the poly-formalism model composability approaches is that the latter supports composition of syntax and semantics of known disparate modeling formalisms. Consequently, with poly-formalism model composition, arbitrary data mappings and transformations with timing and synchronization can be specified [10,52,53]. For example, the differences between DEVS and LP/MPC syntax and semantics are supported with general mappings and domain-specific data transformations. This is in contrast to the meta-formalism model composability approach where differences between different types of model can be partially accounted for in the meta-formalism model composability; thus requiring all remaining differences to be handled on a case-by-case basis. This is because the model of the interactions between composed models needs to be augmented with additional constructs that cannot be handled via meta-formalism or interoperability.

A common approach to overcoming differences between different types of models is illustrated in Figure 5.8. Adapters can be used to facilitate model interactions in the absence of super-, meta-, or poly-formalism model composability approaches. Here *adapter* $\Phi \rightarrow \Psi$ is needed to transform the data that are specified in Φ but that also needs to be used with models specified in Ψ . For each modeling effort, this approach relies on uni- and bidirectional data and control in terms of software design concepts and implementation techniques. This approach requires developing data mappings separately from data transformations. The control of execution between execution algorithms is handled as part of the modeling effort instead of making use of precise sequential, synchronous, or asynchronous execution schemes.

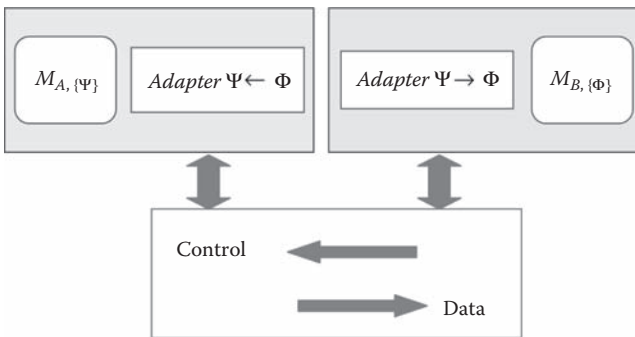


FIGURE 5.8 Model composition with adapters.

5.4 MODEL SPECIFICATION AND EXECUTION ALGORITHM SEPARATION

In the previous section, the composability modeling approaches were described in terms of model specifications. However, as noted earlier, a formalism is defined as a pair. The separation of model specification and execution algorithm enables model specification composability and execution algorithm interoperability differently depending on the composability approach [25,28,30,35,40,53]. As shown in Figure 5.9, the former is concerned with the composition of syntax and semantics of different formalisms, whereas the latter is concerned with the execution protocols and their interoperation.

In the mono-formalism model composability approach, one execution protocol is used for a set of models that are all defined using a single model specification approach. In the super-formalism model composability, execution algorithms of the enclosing models and those that are encapsulated are interoperated under the super-formalism’s execution algorithm. That is, the execution algorithm of the model that is encapsulated in the super-formalism is viewed as an atomic operation within the super-formalism execution algorithm.

The meta-formalism model composability approach, in contrast to super-formalism model composability, is primarily concerned with interoperability. The execution algorithms of the disparate model specifications are cast into model constructs and operations (i.e., a set of services as in HLA) that are intended for interoperating different execution algorithms. In this approach, model composability is not strongly supported and the model specification and execution algorithm are weakly separated from one another.

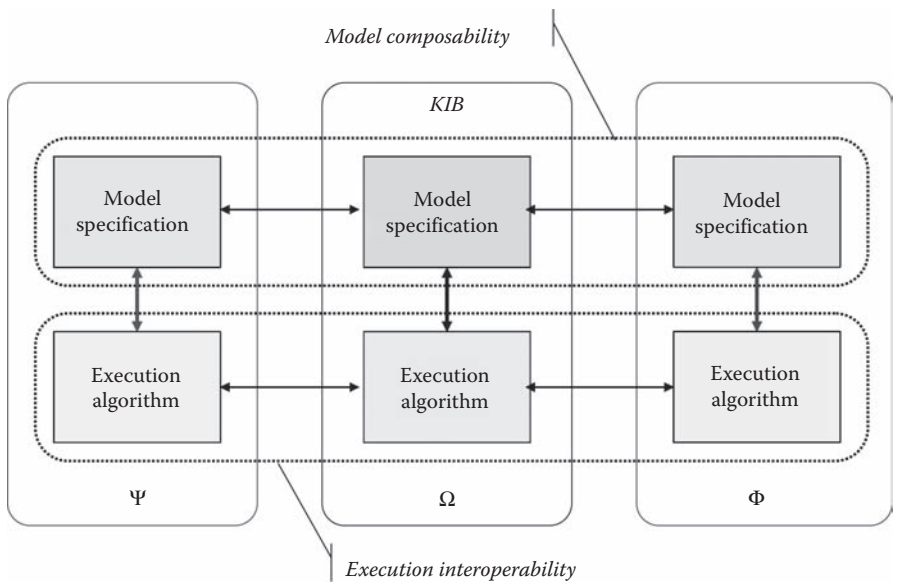


FIGURE 5.9 Separation between model specification and execution algorithm.

The poly-formalism model composability approach, unlike the other approaches, is concerned with both model composability and simulation interoperability. As shown in Figure 5.7, not only are modeling specifications of Ψ and Φ composed using the model specification of Ω , but the execution algorithm of Ω is responsible for interoperability between the execution algorithms of Ψ and Φ . In the domain of semiconductor manufacturing supply-chain systems, Ψ can be DEVS, Φ can be LP or MPC, and Ω can be $\text{KIB}_{\text{DEVS/LP}}$ or $\text{KIB}_{\text{DEVS/MPC}}$ [23,30]. In the domain of intelligent transportation systems [54], Ψ can be DEVS, Φ can be RAP (Reactive Action Packages) [34], and Ω can be $\text{KIB}_{\text{DEVS/RAP}}$ [10].

This separation between *model composability* from *simulation interoperability* is key given different application domains. For example, the DEVS/LP or DEVS/MPC model composability approach [23,30] is based on the manufacturing process (plant) and decision (control) models interaction with one another using a synchronous control scheme consistent with actual operation of a semiconductor manufacturing supply-chain system. In contrast, the DEVS/RAP model composability approach [10] is based on asynchronous interaction between plant and control models. The differences between control schemes defined for interactions in the KIBs may be necessary not only because of the choice of a poly-formalism model composition, but also the specifics of the application domain.

With poly-formalism model composability shown in Figure 5.7, where only two modeling formalisms are composed, if another type of model (e.g., RAP) is introduced that is not well suited to be specified in Ψ (e.g., DEVS) or Φ (e.g., LP), this third type of model needs to be cast into either Ψ or Φ . This suggests that when more than two modeling formalisms need to be composed, multiple or hierarchical KIBs may be needed [25]. Alternatively, a combination of super-, meta-, and poly-formalism model composability approaches may be used instead.

Given the DEVS, LP, and MPC modeling formalisms, the models described in them need to be augmented with adapters written to handle the disparities between them. While inputs and outputs of DEVS model specification are complex messages, the inputs and outputs of LP model specifications are integer and string arrays. This requires customizing DEVS and LP models to handle data received for every model. This is in contrast to a modeling formalism that supports all models that can be described in the DEVS and LP modeling formalisms. The aggregation and disaggregation of data and alternative control schemes can be supported at the level of model specifications instead of using adapters or interoperability services. For example, data and control interactions between the manufacturing process (plant) and the optimization control (controller) models are handled using modeling constructs that are given in the KIB. With the poly-formalism model composability formalism, reuse is supported at the level of modeling formalisms.

5.4.1 DISTRIBUTED EXECUTION OF COMPOSABLE MODELS

All of the above composability approaches lend themselves to distributed execution. For example, the HLA federates can be geographically distributed and executed [55]. HLA, therefore, can be used to interoperate between different execution algorithms (i.e., models synthesized using any of the mono-, super-, meta-, and poly-formalism

model composability approaches can be executed in parallel). Alternative approaches such as Common Object Request Broker Architecture (CORBA) [56] and Service Oriented Architecture (SOA) [57] may also be used for executing heterogeneous composed models

It is also possible to use lightweight technologies to interoperate between distinct execution algorithms (e.g., DEVS-Suite* [58] with Opl-Studio [59], MATLAB®/Simulink® [60], and GRASS). Other considerations (e.g., scale of individual models and data exchanges, frequency of model interactions, and length of experiments) that can affect execution of combined models must also be taken into account. This is because the separation of model specification and execution algorithm can adversely affect the overall execution performance and scalability.

5.4.2 SIMULATION DATA VISUALIZATION

Data visualization is an effective tool for analyzing large data sets. Agent–environment models are good examples of models that are likely to use and to generate large data sets. For example, environmental models may span over two million cells in a cellular automaton and a few hundred agents may add another one thousand data points. These numbers refer to each simulation cycle. If an agent–environment model is simulated for 200 cycles in an effort to study long-term environmental effects, the simulation has produced over 400 million data points. Evaluating the results of an agent–environment simulation like this will benefit from the ability to quickly assess key areas of interest within the resultant simulation data set. Thus, a task for model composability becomes providing an efficient method to collect and display the data—a model architecture-related endeavor. Mono- and super-formalism model composability formalism techniques benefit from the fact that the underlying models share the same or similar formalisms. As a result, the software architecture of the models and development environment are more likely to share similar characteristics, which enables a single technique for extracting and displaying the data. On the other hand, the disparate nature of the modeling formalism within the meta- and poly-formalism model compositions often leads to very different model architectures. A mapping will likely be required to combine the data from the two disparate models. However, these will only capture the data being exchanged between composed models. This does offer an architectural advantage in that the data has already been separated from the composed models. The disadvantage is that model-specific data may be required (i.e., data that remains entirely within one or both composed models). Furthermore, because of the disparate nature of the models and the data that they contain, it may not make sense or be feasible to display the data from the two composed models in the same view. Thus, the simulation data visualization from meta- and poly-formalism model compositions may benefit from software design patterns such as the Model–View–Controller pattern [61] that separate the data from view and control.

* DEVS-Suite is a new generation of the DEVJSJAVA simulator. It offers support for the design of experiments, component-based animation, AND time-based data trajectory visualization for Parallel DEVS models.

5.5 APPLICATION DOMAIN MODELING

A general purpose modeling formalism supports describing and executing a model. It is not intended to account for domain-specific modeling. Therefore, until now, model composability has been considered independent of the application domains to which it may be applied. However, given the central role domain knowledge plays in developing composable models, it is important to consider it for model composability. A domain-neutral modeling formalism still remains key for modeling specific systems (application domains). As the complexity of composite models is due in part to the application domain that is being modeled, the use of domain-neutral formalism provides a proven foundation. This foundation enables the modeler to employ model dynamics that can be examined and tested irrespective of the domain complexities [62]. For example, discrete-event modeling can be used in the domains of manufacturing, information systems, and event-based control. The interactions to be modeled among disparate models need not only capture data transformations and executions between composed models, but also be appropriate to the system being modeled since general purpose modeling formalisms are void of domain knowledge. For example, data transformation between a discrete-event model and an LP optimization model of a manufacturing process has to handle a list of products that are specialized to hold Finished Goods having some defect distribution [23,25]. The frequency between the process and controller models can also depend on the domain; for example, interactions can be sequential or synchronous.

Domain-specific modeling is not only central to the mono- and the super-formalism model composability approaches, but also to the meta- and the poly-formalism model composability approaches. Separating domain-neutral and domain-specific model interactions is also important for handling general purpose modeling concepts while extending them with specific needs of a domain. Given the separation between model specification and execution algorithm of a modeling formalism, the model specification can be extended to support domain knowledge. In the case of the mono- and super-formalism model composability, which are based on the concept of components, general purpose model components can be specialized for a given domain. For example, DEVS model components are specialized to the entities of manufacturing process network.

In the case of the meta-formalism model composability, and HLA in particular, domain specific federates are modeled using specialization supported by object-oriented concepts and methods. In addition, data engineering (XML family of data representations and ontologies) may also be used for handling static (i.e., nonbehavioral) domain-specific knowledge [9]. For poly-formalism model composability, due to the strong separation between modeling formalisms that are composed, existing approaches to domain-specific modeling may be used. The existence of the KIB offers a basis to explicitly account for the interaction of domain knowledge.

Figure 5.10 shows a general approach to implementing poly-formalism model composition. Model A and Model B are the two models with disparate formalisms that are being composed. As discussed previously, each model's formalism is chosen to best represent the subsystem being modeled. The KIB formalism (Ω from Figure 5.9) and *realization* (i.e., implementation) of the composing model, within the

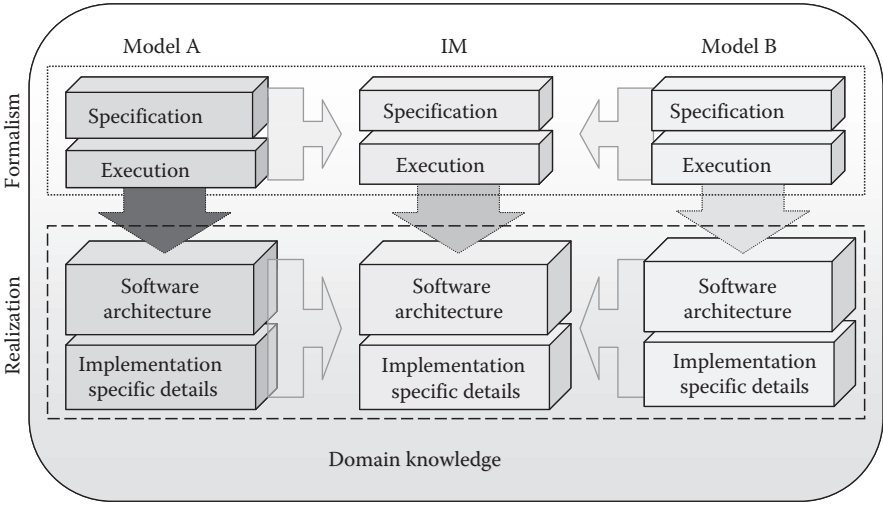


FIGURE 5.10 Domain-driven poly-formalism model composition.

context of the domain, are referred to collectively as an *Interaction Model* (IM). The formalism of the IM is chosen based upon the formalisms of the two composed models, A and B, and the definition of their interaction within the domain. For example, if Model A is a discrete-time model, and Model B is a discrete-event model, then the IM may be chosen to be a discrete-event model in order to manage events that arise from Model B. Note that the IM need not be specified by either Model A's nor Model B's formalisms; it could be an entirely different formalism altogether. As Figure 5.10 shows, all three models' realizations are derived from the respective formalisms. Realization refers to a models' software architecture (including development framework), and implementation specific details. The latter is highly domain influenced. Disparities between the composed models' realizations drive the implementation of the IM. Thus, the dynamics within the IM may be focused on such tasks as aggregating or disaggregating values, or transforming objects to primitive value sets.

Given the above composability approaches and the kinds of modeling formalisms used offer different degrees of support for describing domain knowledge. That is, the degree of support for domain-specific modeling is dependent both on the implementations of the modeling formalism for each composed model, and the model composition approach used.

5.5.1 MODELING INTERACTIONS BETWEEN DISCRETE PROCESS AND PREDICTIVE CONTROL MODELS

The benefits of poly-formalism model composability have been shown in the domain of semiconductor manufacturing supply-chain systems described in Section 5.2.1 [25,30,52]. The DEVS/MPC testbed [25] supported configuring, simulating, and analyzing the discrete-event process, optimization control, and the KIB models both individually and collectively. To assess the benefit of an MPC that relies on a linear,

discrete-time model for its decision making, it is key to have a nonlinear, stochastic, and discrete-event model of an actual manufacturing process [25]. The MPC's feedforward and feedback controllers can be accurately parameterized at a realistic level of detail. Similarly, to accurately model the dynamics of the manufacturing processes, it is important for lots in the inventory nodes to be released to the manufacturing nodes under optimized control commands. Given the complex dynamics of interacting process and control models, real-world scenarios can be modeled [28]. For example, for the manufacturing process model, 3-level and 5-level throughput-load models for the manufacturing node were considered. The 5-level model more accurately represents the change in factory throughput as the number of lots to be processed increases. Similarly, the MPC model has a tunable control parameter that can handle stochasticity and uncertainty in customer demand.

The interactions between the DEVS and MPC models were modeled with the $KIB_{DEVS/MPC}$ [25]. The KIB consists of a number of models such as the Semifinished that defines the interactions between the manufacturing process network and the predictive controller (see Figure 5.11). The partial manufacturing process has process nodes M_{20} and M_{30} and inventory node I_{20} . The Semifinished KIB component defines specific mappings between the manufacturing process network and the predictive controller. The KIB snippet shows the predictive controller needs to have the inventory data called Beginning On Hand (BOH) in the manufacturing process model. Similarly, the Semifinished Goods needs a release command called Release in the predictive controller model in order to limit the number of goods that should be released to Finish process M_{30} . The Semifinished KIB model disaggregates data from the predictive controller (i.e., u_2), an element of the control vector U , is the Release variable for the Semifinished Goods inventory I_{20} and aggregates data for the predictive controller (i.e., the BOH inventory variable y_2 is aggregated with other variables to form data vector Y for the predictive controller). The models included in the KIB follow a time-based, sequential synchronization for all data mappings and control exchanges.

A variety of experiments may be devised and simulated for the semiconductor manufacturing supply-chain system shown in Figure 5.2 [29]. For example, a customer demand profile that has a pulse signal that increases by 500 units (53% variation compared with the average customer demand) from day 201 to day 400. The manufacturing process is to be optimally controlled such that the customer demand is satisfied for a desired Die/Package inventory target as shown in Figure 5.12. For the 3-level throughput-load, the Die/Package inventory has substantial transient dynamics because of the significant change in the upstream Assembly/Test factory. Comparing the 5-level throughput-load response with the 3-level throughput-load, the benefits of the smoother transition between throughput values is obvious. The level of details provided with the DEVS process and MPC control models as well as their KIB interaction model supports design and operation of realistic supply-chain systems that would be difficult or impractical using mono-, super-, and meta-formalism modeling composability approaches. With the interaction model, the frequency and timing of the data and control exchanges between the process and control models can be modeled and implemented [25]. For example, given the explicit model of the process and control models, comprehensive analysis and design of data and control

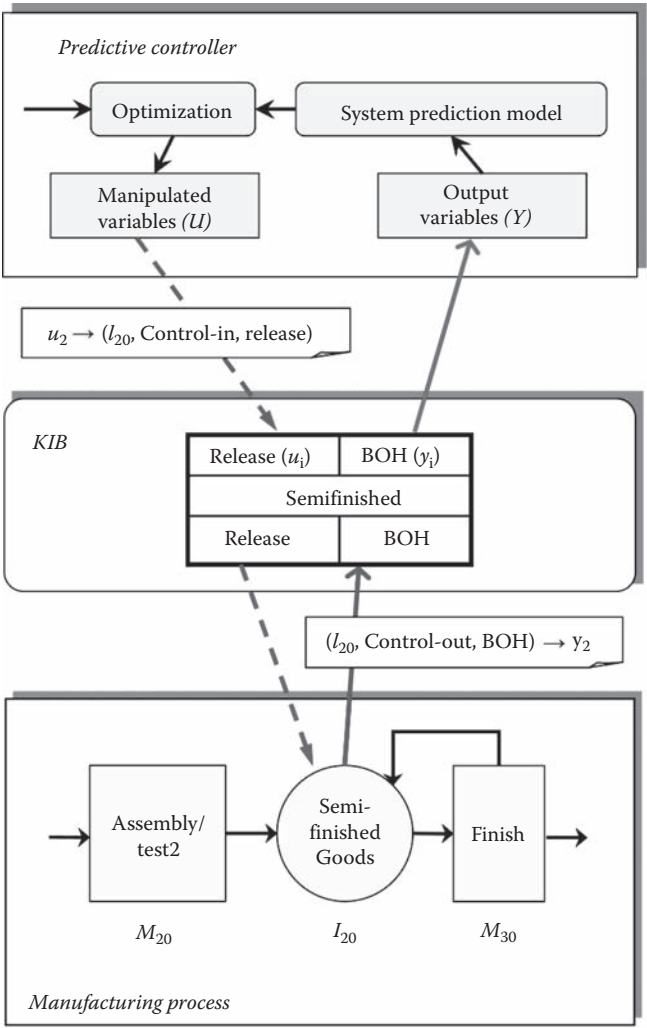


FIGURE 5.11 Composition of DEVS and MPC models with a KIB model.

regimes for shifts (8 or 12 hours), daily, or weekly are accounted for in a straightforward fashion [52]. With the 5-level throughput-load model, it can be determined how to best set controller parameters and thus achieve substantial reductions in both the absolute and sum of squares deviation from the target for both the die/package and semifinished goods inventories.

5.5.2 AGENT-ENVIRONMENT INTERACTION

Figure 5.13 is the output of a simulation that shows the impacts of two Bronze Age farming villages and their associated agricultural activity. Underlying the simplistic looking figure is a number of interactions between the agent and environment

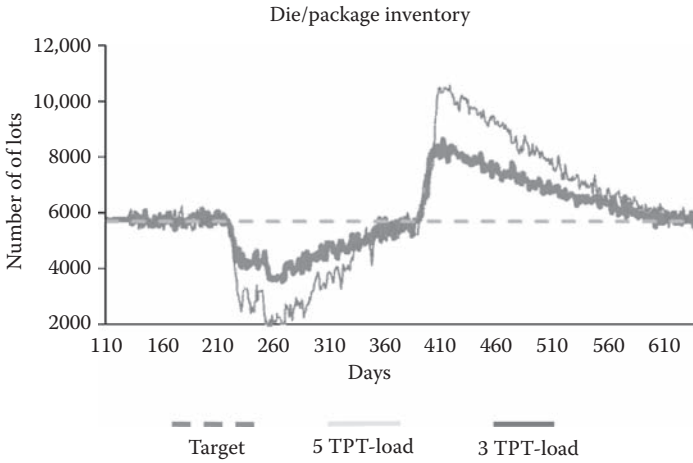


FIGURE 5.12 Semiconductor supply-chain simulation.

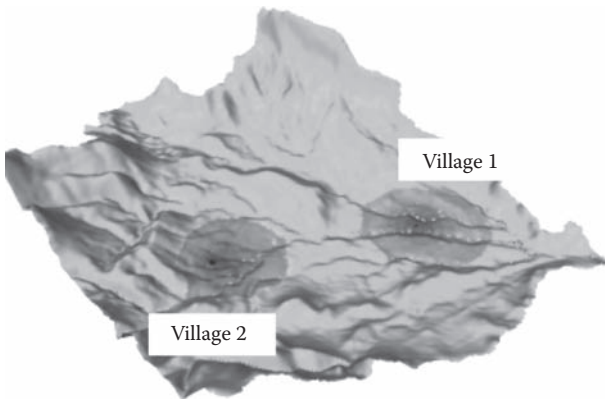


FIGURE 5.13 Agent-environment simulation.

models. In this scenario, village sites are determined a priori. Agents within a village select the best land to farm using a set of weighted criteria such as the soil quality, the soil depth, and a distance cost that accounts for how far the land is from the agent's village and how difficult it is to traverse that distance. This last attribute, the distance cost, may account for things like slope of the terrain and foliage that impedes travel. This is why the darkened, circular areas in Figure 5.13 tend to follow the terrain vice move up steep slopes.

These soil attributes are very detailed, dynamic environmental aspects that the agent model would likely not retain within its simplified abstraction of its environment; else it would be taxed with routinely checking for updated values. Furthermore, the agent domain is that of farmers during the Bronze Age. It is useful to maintain a simple abstraction of the agent's knowledge as it pertains to the Bronze Age data. This abstraction is broken if detailed environmental data is sent to the agent that we,

as humans, have only recently come to understand. Therefore, it may be beneficial to map specific environmental variable values to more abstract representation. For example, soil quality within the environment model may be represented by explicit values of nitrogen, phosphorus, and potassium, while the agent utilizes more simplistic, aggregate values of “poor,” “fair,” and “good.” Once an agent has analyzed its environment and chosen where to farm, it must then modify the environment to actually do such.

The farmed site requires unique representation in both models. The agent model needs to know what is farmed so that the agents can maintain control over their managed resources and get food during harvest. They also need to be able to manage competition between agents wanting to farm the same land. The environment model needs to know what land has been farmed so that this impact can be accounted for in dynamical processes such as soil erosion calculations and foliage growth. Each model requires knowledge of the farming actions, but in its own domain language such that the two models do not become closely coupled. For example, the farming action may be represented in the environment model as a change in land cover and soil quality. This abstraction provides the details required for the environment model but doesn’t retain knowledge of which agent controls what piece of land. Similarly, the managing agent’s data has no meaning within the environment model. Furthermore, the agent’s harvest dependent upon dynamic values of soil depth, soil quality, and rainfall would be duplicated and difficult to manage if maintained in the agent model while still maintaining a separation of knowledge between the agent and the environment models.

As another example, consider the data mapping that may occur between an agent model representing early farmers and an environment model that represents an erosion process. The erosion process, E , may be a simple water transport function with input variables of rainfall, r , which specifies how much water is available to move soil, and soil depth, d , which enumerates how much soil is available to move. The output of this function is then scaled by the product of two erosion factors—land cover, c , and soil compaction, k . These scalar values would each range from 0.0 to 1.0, where a value of 0.0 indicates all erosion is prevented. Let these sets of possible values be denoted C and K , respectively. The cumulative erosion effect would be represented by $E(r, d) \times c \times k$. For simplistic, rain-fed farming methods, the agents would not have the capability to affect rainfall or soil depth to any great extent. However, their actions could modify both the land cover and the soil compaction values. But, within the agent model, concepts such as land cover are more likely to be qualitative values. For example, land may be barren of any foliage, it may contain some scrub, or it may be fully wooded. In order to farm the area, the agent would remove land cover to a point at which crops could more easily be grown *en masse*. If the agents’ set of land cover descriptors is the set L , where $L = (\text{“barren,” “some grasses,” ... , “heavily wooded”})$, then the IM would contain the mapping function set, M , for $L \xrightarrow{\text{onto}} L^*$, and $\forall m(l) \in M, m(l) = l^*$, where $L^* \subseteq C$, $l \in L$, and $l^* \in L^*$. The erosion for the areas in which the agents farmed would then be $E(r, d) \times l^* \times k$. A similar mapping could be created to account for soil compaction in and around village settlements where there is a lot of concentrated foot traffic. A benefit to this approach is that if the agents were made more sophisticated in the future, say to

aerate (or decompact) the soil, then only the mapping function(s) would change. The change in agent behavior would not necessitate a change in the environment model structure or the dynamics.

Considering interaction from the environment's perspective, a fire or flooding may result from natural causes such as lightening or heavy rain. As the natural disaster moves, there may be impacts to the agents' farms and villages. Once again, knowledge from both models is required to fully express these disaster events within the agent–environment system model. The interaction model for the agent–environment system model provides a clean and precise way to manage the two models' interactions.

The environment in Figure 5.13 is represented by approximately 1.6 million cells. There are approximately 70 people represented by agents within the agent model. The interaction model captures the agent impacts on the environment and maps them to the environmental data. This data from the interaction model can then be contrasted with environmental erosion maps to determine what impacts the agents have had on the environment (e.g., an increase or decrease in erosion) and what effect the environment has had on the agents. For example, determining if the lack of food is because of environmental factors caused by the agents' own actions or something out of their control (e.g., a slight decrease in rainfall had a more significant impact on the area than anticipated).

The interaction model maintains an independence between the two models in addition to the interaction data, control, and timing management that it provides. This independence manifests as a lack of knowledge of structural, behavioral, and data properties of the other composed model. The first benefit of this approach is that the two models may be developed separately by domain experts of the particular fields of study. The second benefit is that a change to one model is unlikely to have an impact on the other composed model. This is not necessarily true of the IM though. As the IM is a model of the interaction between the composed models; if something about the interaction changes, then the interaction model must change too.

5.6 CONCLUSIONS

A model composability taxonomy is presented based on multi-formalism modeling approaches. These approaches differ significantly from one another in supporting heterogeneous model composition. The classification can help in selecting a suitable model composability approach with modeling formalisms that can best represent the structure and behavior of system and its subsystems. How disparate modeling formalisms affect model composability is described using simplified examples from the domains of the semiconductor supply-chain and agent–environment systems. The discussion on separating model specifications and execution algorithms reveals the impact of model composability choices. In particular, it demonstrates support for varying degrees of model compositions and consequently, limitations and complexity of specifying disparate composite models, level of support for domain-specific modeling, and degree of support for distributed execution.

REFERENCES

1. SBA, *Simulation Based Acquisition: A New Approach*. Report of the Military Research Fellows DSMC. Belvoir, VA: Defense Systems Management College, 1998.
2. Kaspitis, S., and H. C. Ng. Composable Simulations, *Proceedings of the 2000 Winter Simulation Conference*, 1577–84, Orlando, FL, 2000.
3. Page, E. H., and J. M. Opper. Observations on the Complexity of Composable Simulation, *Proceedings of the Winter Simulation Conference*, 553–60, Phoenix, AZ, 1999.
4. Sarjoughian, H. S., and F. E. Cellier. *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*. New York: Springer Verlag, 2001.
5. Barros, F., and H. S. Sarjoughian. *Component-Based Modeling and Simulation*, Guest Editorial, *Transactions of The Society for Modeling and Simulation International* 80 (2004): 319–20.
6. Fujimoto, R., D. Lunceford, E. Page, and A. Uhrmacher. *Grand Challenges for Modeling and Simulation*. Schloss Dagstuhl, 2002.
7. Davis, P. K., and R. H. Anderson. Improving the Composability of DoD Models and Simulations, *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology* 1, no. 1 (2004): 5–17.
8. Turnitsa, C. D. Extending the Levels of Conceptual Interoperability Model, *Summer Computer Simulation Conference*, Philadelphia, PA, 2005.
9. Tolk, A., and S. Diallo. Model-Based Data Engineering for Web Services, *IEEE Internet Computing* 5, no. 4 (2005): 65–70.
10. Sarjoughian, H. S., and D. Huang. A Multi-Formalism Modeling Composition Framework: Agent and Discrete-Event Models, *The 9th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Montreal, Quebec, Canada. New York: IEEE, 2005.
11. Cellier, F. E. *Continuous System Modeling*, xxvii, 755. New York: Springer Verlag, 1991.
12. Fishwick, P. A. An Integrated Approach to System Modelling using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies, *ACM Transactions on Modeling & Computer Simulation* 4, no. 2, (1992): 307–30.
13. Fishwick, P. A. *Simulation Model Design and Execution: Building Digital Worlds*. Upper Saddle River, NJ: Prentice Hall, 1995.
14. Miller, V. T., and P. A. Fishwick. Hybrid Heterogeneous Hierarchical Models for System Simulations, *International Journal in Computer Simulation* 5, no. 3 (1995): 209–27.
15. Sarjoughian, H. S., and B. P. Zeigler. DEVS and HLA: Complementary Paradigms for Modeling and Simulation? *Simulation Transactions* 17, no. 4 (2000): 187–97.
16. Zeigler, B. P., H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd ed. New York: Academic Press, 2000.
17. Wang, J. Timed Petri Nets: Theory and Application, *The International Series on Discrete Event Dynamic Systems*. New York: Springer, 1998.
18. Kempf, K. G. Control-Oriented Approaches to Supply Chain Management in Semiconductor Manufacturing, *Proceedings of American Automatic Control*, 4563–76, Boston, MA, 2004.
19. Ilachinski, A. *Cellular Automata: A Discrete Universe*. Hackensack, NJ: World Scientific, 2001.
20. Tomlin, C. D. *Geographic Information Systems and Cartographic Modeling*. Englewood, NJ: Prentice Hall, 1990.
21. GRASS, *Geographic Resources Analysis Support System*, 2005 [cited April 2005].

22. Rardin, R. L. *Optimization in Operations Research*. Upper Saddle River, NJ: Prentice Hall, 1998.
23. Godding, G. W., H. S. Sarjoughian, and K. G. Kempf. Multi-Formalism Modeling Approach For Semiconductor Supply/Demand Networks, *Proceedings of the Winter Simulation Conference*, 232–39, 2004.
24. Qin, S. J., and T. A. Badgwell. A Survey of Industrial Model Predictive Control Technology, *Control Engineering Practice* 11, no. 7 (2003): 733–64.
25. Huang, D., H. S. Sarjoughian, W. Wang, G. W. Godding, D. E. Rivera, K. G. Kempf, and H. D. Mittelmann. Simulation of Semiconductor Manufacturing Supply-Chain Systems with DEVS, MPC, and KIB, *IEEE Transactions on Semiconductor Manufacturing* 22, no. 1 (2009): 165–74.
26. Wang, W., and D. E. Rivera. A Model Predictive Control Algorithm for Tactical Decision-Making in Semiconductor Manufacturing Supply Chain Management, *IEEE Transactions on Control Systems Technology* 16, no. 5 (2008): 841–55.
27. DEVJAVA, *DEVJAVA Simulator*, 2001 [cited 2007, April 19]; Available from: <http://www.acims.arizona.edu/SOFTWARE>
28. Godding, G. W. *A Multi-Modeling Approach Using Simulation and Optimization for Supply-Chain Network Systems*, PhD Dissertation, Computer Science and Engineering. Tempe, AZ: Arizona State University, 202, 2008.
29. Huang, D. *Composable Modeling and Distributed Simulation Framework for Discrete Supply-Chain Systems with Predictive Control*, PhD Dissertation, Computer Science and Engineering. Tempe, AZ: Arizona State University, 191, 2008.
30. Sarjoughian, H. S., D. Huang, W. Wang, D. E. Rivera, K. G. Kempf, G. W. Godding, and H. D. Mittelmann. Hybrid Discrete Event Simulation with Model Predictive Control for Semiconductor Supply-Chain Manufacturing, *Proceedings of the Winter Simulation Conference*, 256–66, Orlando, FL, 2005.
31. Singh, R. K., H. S. Sarjoughian, and G. W. Godding. Design of Scalable Simulation Models for Semiconductor Manufacturing Processes, *Proceedings of the Summer Computer Simulation Conference*, San Jose, CA, 2004.
32. Mayer, G. R., H. S. Sarjoughian, E. K. Allen, S. Falconer, and C. M. Barton. Simulation Modeling for Human Community and Agricultural Landuse, *Agent Directed Simulation Symposium at Spring Simulation Multi-Conference*, 65–72, Huntsville, AL, 2006.
33. Hall, S. Learning in a Complex Adaptive System for ISR Resource Management, *Spring Simulation Conference*, 5–12, San Diego, CA, 2005.
34. Firby, R. J., and W. Fitzgerald. *The RAP System Language Manual, Version 2.0*. Evanston, IL: Neodesic Corporation, 2000.
35. Mayer, G. R., and H. S. Sarjoughian. Composable Cellular Automata, *Simulation Transactions*, 85, no. 11–12 (2009): 735–749.
36. Ntaimo, L., B. P. Zeigler, M. Vasconcelos, and B. Khargharia. Forest Fire Spread and Suppression in DEVS, *Simulation Transactions* 80, no. 10 (2004): 479–500.
37. Wainer, G. Applying Cell-DEVS Methodology for Modeling the Environment, *Simulation Transactions* 82, no. 10 (2006): 635–60.
38. Mayer, G. R., and H. S. Sarjoughian. Complexities of Simulating a Hybrid Agent-Landscape Model Using Multi-Formalism Composability, *Spring Simulation Conference*, 161–68, Norfolk, VA, 2007.
39. Sarjoughian, H. S. Model Composability, *Winter Simulation Conference*, 149–58, Monterey, CA, 2006.
40. Sarjoughian, H. S., and J. Plummer. *Design and Implementation of a Bridge between RAP and DEVS*, Internal Report, Computer Science and Engineering, 1–26. Tempe, AZ: Arizona State University, 2002.

41. Wymore, A. W. *Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricategory Theory of System Design*. Boca Raton, FL: CRC Press, 1993.
42. Zeigler, B. P. DEV & DESS in DEVS: Characteristic Behaviors of Hybrid Models, *DEVS Integrative Modeling & Simulation Symposium*, 125–32, Huntsville, AL, 2006.
43. Sarjoughian, H. S., and B. P. Zeigler. *Inductive Modeling: A Framework Marrying System Theory and Non-Monotonic Reasoning*, Hybrid II, Lecture Notes in CS, Vol. 999. Edited by P. Antsaklis and A. Nerode, 417–35, 1995.
44. Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity—The Ptolemy Approach, *Proceedings of the IEEE* 91, no. 2 (2003): 127–44.
45. de Alfaro, L., and T. A. Henzinger. Interface Automata, *Proceedings of the 8th European Software Engineering Conference*, Vienna, Austria, 2001.
46. Dahmann, J., M. Salisbury, C. Turrel, P. Barry, and P. Blemberg. HLA and Beyond: Interoperability Challenges, *Simulation Interoperability Workshop*, Orlando, FL. New York: IEEE, 1999.
47. *HLA Framework and Rules, Version IEEE 15162000*. New York: IEEE, 2000.
48. *HLA Interface Specification, Version IEEE 1516.1-2000*. New York: IEEE, 2000.
49. *HLA Object Model Template, Version IEEE 1516.2-2000*. New York: IEEE, 2000.
50. Jaramillo, J. L., H. Vangheluwe, and M. A. Alfonseca. Using Meta-Modelling and Graph Grammars to Create Modelling Environments, *First International Conference on Graph Transformation*, 36–50, Barcelona, Spain. Electronic Notes in Theoretical Computer Science, 2002.
51. Mosterman, P. J., and H. Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction, *Simulation Transactions* 80 (2004): 433–50.
52. Godding, G. W., H. S. Sarjoughian, and K. G. Kempf. Application of Combined Discrete-Event Simulation and Optimization Models in Semiconductor Enterprise Manufacturing Systems, *Proceedings of Winter Simulation Conference*, 1729–36, Washington, DC, 2007.
53. Mayer, G. R. *Composing Hybrid Discrete Event System and Cellular Automata Models*, 133, PhD Dissertation, Department of Computer Science and Engineering. Tempe, AZ: Arizona State University, 2009.
54. Sarjoughian, H. S., B. P. Zeigler, and S. B. Hall. A Layered Modeling and Simulation Architecture for Agent-based System Development, *IEEE Proceedings*, 201–13. Edited by A. M. Uhrmacher, P. A. Fishwick, and B. P. Zeigler, 2001.
55. Fujimoto, R. M. Time Management in the High-Level Architecture, *Simulation* 71, no. 6 (1998): 388–400.
56. Orfali, R., D. Harkey, J. Edwards, and D. Harkey. *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons, 1995.
57. Erl, T. *Service-Oriented Architecture Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice Hall, 2006.
58. DEVS-Suite, *DEVS-Suite Simulator*, 2008 [cited 2009, April 30]; Available from: <http://sourceforge.net/projects/devs-suitesim/>
59. ILOG, *OPL Studio*. 2005: <http://www.ilog.com/products/oplstudio/> [cited 2006].
60. The MathWorks™, *MATLAB®/Simulink®*. 2005: <http://www.mathworks.com> [cited 2007].
61. Reenskaug, T. M. H. *MVC XEROX PARC 1978–79, 1979* [cited 2009, March]; Available from: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>
62. Sarjoughian, H. S., G. R. Mayer, C. M. Barton, and I. I. Ullah. *Managing Model Composition Complexity: Human-Environment Simulation Models*, Department of Computer Science and Engineering, 1–25. Tempe, AZ: Arizona State University, 2009.

6 Effective and Efficient Modeling and Simulation with DEVS Variants

*Adeline M. Uhrmacher, Jan
Himmelspach, and Roland Ewald*

CONTENTS

6.1	Introduction	140
6.2	Modeling Formalisms Based on DEVS.....	141
6.2.1	Example: A Virus Attack	141
6.2.2	DYNPDEVS	142
6.2.2.1	Virus Attack in DYNPDEVS	145
6.2.3	p-Devs.....	146
6.2.3.1	Virus Attack in p-Devs.....	149
6.2.4	ml-Devs.....	151
6.2.4.1	Virus Attack in ml-Devs.....	153
6.2.5	epi-Devs	153
6.2.6	Model Components.....	154
6.3	Algorithms and Data Structures for Simulating DEVS Models.....	155
6.3.1	Simulation Algorithms	155
6.3.1.1	Abstract Threaded.....	155
6.3.1.2	Abstract Sequential	157
6.3.1.3	Hierarchical Sequential.....	158
6.3.1.4	Flat Sequential	159
6.3.1.5	Parallel Sequential	160
6.3.1.6	Reuse.....	162
6.3.1.7	Results.....	162
6.3.2	Partitioning and Load Balancing.....	163
6.3.2.1	Partitioning	164
6.3.2.2	Hierarchical Partitioning	165
6.3.2.3	Results	167
6.3.2.4	Load Balancing.....	167
6.3.2.5	A DEVS-Specific Load Balancing Algorithm	168
6.3.3	Event Queues	169
6.3.3.1	Calendar Queue	169
6.3.3.2	Results.....	169
6.3.4	Summary	171

6.4 Summary and Outlook 172

Acknowledgment 172

References..... 172

6.1 INTRODUCTION

Since the 1970s, when Discrete Event System Specification (DEVS) was introduced by [1], the number of DEVS variants is increasing steadily. This is rather typical for modeling formalisms, for example, we also find a huge number of Petri Nets variants, all aimed at increasing the expressiveness of the original formalism. Referring to DEVS, three types of variant prevail: those that extend its discrete nature by introducing continuous aspects [2,3], those that exploit the DEVS formalism for real-time applications by coupling DEVS to external sources [4,5], and those that try to overcome the static structure of the formalism. Whereas the state of a DEVS model changes over time, the components and couplings of coupled models as well as the state space and functions of atomic models do not change in many DEVS formalisms. This hampers the description of systems that are characterized by variable structures.

The lack of a suitable support of this type of dynamics has already been identified in the mid-1980s [6]. Thereafter, a variety of approaches to address this deficiency has emerged [7–12,27]. Partially it has been shown that these variants can be simulated by the original DEVS formalism, for example, the DYNDEVS formalism and the DEVS formalism are bisimilar [9]. The value of the new variants lies in the ease of modeling.

Variable models structure the temporal dimension of models. Composing models add structure referring to the organizational dimension. Both are aimed at a more efficient modeling. The modular design of DEVS models—each communicates with its environment via input and output ports—allows the construction of complex models by composition. However, to truly support the design and exploitation of model components, further significant efforts are required. Model components shall facilitate the reuse of models by others and the composition of models on demand. Similar to reusing software components, this is not easily achieved in praxis [13]. Component-based modeling shares the problems of component-based software design. In addition, it has to deal with the problem of valid model reuse in varying contexts.

The efficient modeling with DEVS, based on certain formalisms or on model composition, is one important aspect of using DEVS in practice. Another is the efficient simulation of the specified DEVS models. The operational semantics of DEVS formalisms are usually described by abstract simulators. Their functioning can be realized by various algorithms, and those in turn can be supported by various additional subalgorithms. To have more than one algorithm is highly advisable. Supporting several simulators can help testing the soundness of new simulation algorithms. As the efficiency of an algorithm depends also on the model, to support the efficient execution of different models requires being able to select from different algorithms. Several algorithms have been proposed in the literature or can be found in implementations [14–18,68]. Distributed simulation requires partitioning and load balancing techniques to complement the basic algorithms. Several partitioning methods have been proposed over the years, aiming at balancing the load evenly and minimizing interprocessor communication [19–21]. In addition to the actual algorithms, a data

structure widely used in the discrete-event world is required for the computation of DEVS models as well, but has to support additional methods: event lists organize the upcoming (internal) events [22]. Efficiently executing a DEVS model implies that all involved algorithms are working together “optimally,” which is of particular importance for models with strong memory requirements or large computational cost.

The plethora of alternatives sketched above provides the possibilities for a lot of reuse on the technical level, and should thus be realized on the base of a single system. This reduces the overall work to be done, can reduce the number of faults, and allows for efficient modeling and simulation (M&S). Before outlining the technical realization of this approach, we first discuss some variants of DEVS and thereby focus on formalisms for variable structure models.

6.2 MODELING FORMALISMS BASED ON DEVS

Our extensions of DEVS are based on Parallel Discrete Event System Specification (PDEVS); [23] rather than the original DEVS formalism [24]. PDEVS supports a parallel execution of internal and external events that occur at the same simulation time and avoids the need for specifying a select function at the level of the coupled model. As does DEVS, PDEVS assumes a static model structure. However, with the number of flexible and adaptable technical systems, for example, in the area of multiagent systems [25], and thriving application areas like systems biology [26], the need for the specification and analysis of these systems has been growing in the last decade. Variable structure models add structure to the temporal dimension in describing systems. As the relation of bisimulation has been shown to hold between the original DEVS formalism and variable structure variants like dynamic structure system network (DSSN; [27]) and dynamic DEVS (DYNDEVS; [9]), the question arises, what benefit to expect from such formalisms. It appears that it is less important *whether* a formalism is able to express certain phenomena—but *how easily* this can be done is crucial for a discussion [28].

6.2.1 EXAMPLE: A VIRUS ATTACK

To illuminate the central concepts of the different formalisms, we introduce a biological example, the lambda phage. The lambda phage has been studied extensively in systems biology, particularly as a case study for discrete, stochastic modeling [29]. In the context of this work we will only sketch fragments of the overall model.

The lambda phage is a virus infecting *Escherichia coli* bacteria (*E. coli*). The virus injects its DNA into the cell. Depending on the concentration of certain key molecules (e.g., CI or Cro), the cell will enter either the lytic cycle or the lysogenic cycle. In the lytic cycle, the phage DNA will be directly transcribed by the cell’s gene expression machinery, so that a phage replication takes place until the resources of the cell are depleted. Thus, the cell will be destroyed and the produced phages will be released to the environment. In the lysogenic cycle, the phage DNA will be integrated into the cell’s DNA. Each time the cell replicates, the phage DNA being part of the cell’s DNA will be replicated as well. If at some point the environmental conditions change, the cell will enter the lytic cycle. Figure 6.1 illustrates the relation between both cycles.

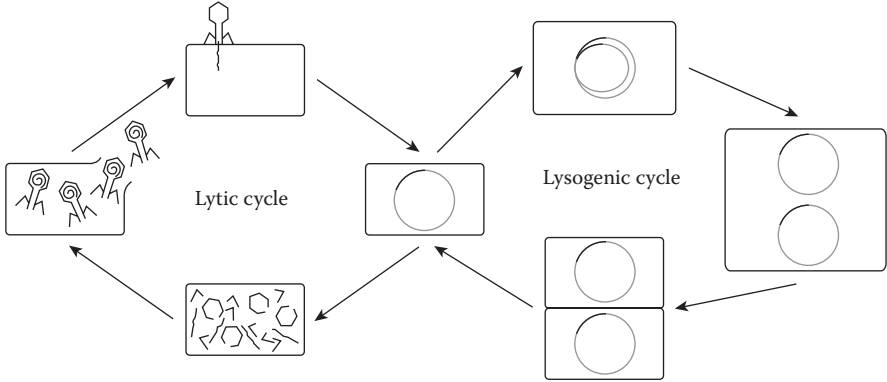


FIGURE 6.1 The lytic and lysogenic cycle of the lambda phage

6.2.2 DYNPDEVS

The starting point for our discussion of variable structure models is DYNDEVS [9]. In DYNDEVS, a model's description entails the possibility of changing its own state and behavior pattern. The underlying idea of DYNDEVS is to interpret models as a set of models successively generating themselves by model transitions. Thereby, sequences of models are produced. Model and network transitions are introduced that map the current state of a model into a set of models that the model belongs to. In the following, a revised and parallel version of DYNDEVS, DYNPDEVS is described, as it has been implemented in JAMES II [10,30].

Each element m_i of the set M represents an incarnation of the model and describes a phase of the evolving dynamic system that is modeled. Unlike other approaches that emphasize the distinction between the controlling and controlled unit [27], for example, the formalism supports models that adapt their own interaction structure and their own behavior. The transition function ρ_α is responsible for this. In addition, a special port type is introduced for propagating the information that is necessary to realize variable structures at coupled model levels, via the network transition function ρ_N . The newly introduced function ρ_λ is feeding this port, and structural changes at the network level are induced bottom-up. More about this in the section about the DYNPDEVS network (see Definitions 6.2.1 and 6.2.2).

The DYNPDEVS defines inputs, outputs, and states (i.e., X , Y , and S) as structured sets. They are structured according to a set of variable names, which, in the case of input and output, denote ports by which inputs are received and outputs are sent. The model transition ρ_α does not interfere with other transitions; it preserves the values of variables that are common to the states of two successive model incarnations and assigns “default initial” values to the “new” variables, as described in [9].

Input and output ports are part of the model incarnations and can thus be exchanged via the model transition. In addition, a special type of port has been introduced. The role of ρ_λ is to fill the port for structural changes Y_{sc} , which shall occur at the level of the network model. This information will be accessed by the network transition function ρ_N of the parent coupled model (see Definition 6.2.2). As other models might induce structural changes that have an effect on the model incarnation,

an input port X_{sc} has been added for this kind of request. Its input is considered when applying ρ_α to determine the new model incarnation.

Definition 6.2.1

An atomic DYNPDEVS model is the structure $\langle m_{init}, M, X_{sc}, Y_{sc}, X, Y \rangle$ with $m_{init} \in M$ being the initial model; X, Y , the structured sets of inputs and outputs; X_{sc}, Y_{sc} the structured sets for communicating structural changes; and M the least set with the following structure:

$$\langle S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \rho_\alpha, \rho_\lambda, \lambda, ta \rangle$$

where

S	structured set of states
$s_0 \in S$	initial state
$\delta_{int} : S \rightarrow S$	internal transition function
$\delta_{ext} : Q \times X^b \rightarrow S$	external transition function, with
$Q = \{(s, e) : s \in S, 0 \leq e < ta(s)\}$	state set including elapsed time
$\delta_{con} : S \times X^b \rightarrow S$	confluent transition function
$\rho_\alpha : S \times X_{sc} \rightarrow M$	model transition function
$\rho_\lambda : S \rightarrow Y_{sc}$	implied structural network changes
$\lambda : S \rightarrow Y$	output function
$ta : S \rightarrow \mathbb{R}^{\geq 0} \cup \{\infty\}$	time advance function

and for which the following reachability property holds:

$$\forall n \in M : n = m_{init} \vee \exists m_0 = m_{init}, \dots, m_i = n : \exists x_k \in X_{sc} : \rho_\alpha(s^{m_k}, x_k) = s^{m_{k+1}}$$

with $i > 0; k = 0, \dots, i-1; s^{m_k}, s^{m_{k+1}} \in S^{m_{k+1}}; m_0, \dots, m_i \in M$.

As usual for DEVS variants, the operational semantics is described by the abstract simulator. Each simulator has a link to the current incarnation of the model m (line 4, Listing 6.1). In addition, it keeps track of the time by storing the time of the last event, $tole$, and the time of the next event, $tonie$ (1.21, 1.20). After executing internal, external, and confluent events of the current model incarnation, it is checked whether a structural request concerning the network structure has to be processed by invoking $m.\rho_\lambda$ (1.15). The structural output is sent to the parent (1.16). After having received structural input, the model transition function ρ_α is invoked to generate the new incarnation if necessary.

LISTING 6.1

Pseudo Code of DYNPDEVS Simulator

```

1 when receive *, x or sc message
2   if message is * or x message
3     if message is * message
4        $y := m.\lambda(s)$ 

```

```

5      send y message to parent
6      wait for receive x message
7      if isEmpty (x message)
8          m.s := m.δint(m.s)
9      else
10         m.s := m.δcon(m.s, x)
11         end if
12     else
13         m.s := m.δext(m.s, x, t-tole)
14         end if
15         yr := m.ρλ(m.s)
16     send sc(yr) message
17 end if
18 wait for sc message
19 m := m.ρα(m.s, sc)
20 tonie := t + ta(m.s)
21 tole := t
22 send done (tonie) message
23 end when

```

To support hierarchical and modular modeling, a network structure is introduced in DYNPDEVS. Here, a structural change means a change of interaction and composition structure.

Definition 6.2.2

A dynamic network, a DYNNPDEVS, is the structure $\langle n_{init}, N, X_{sc}, Y_{sc}, X, Y \rangle$ with $n_{init} \in N$ being the start configuration; X_{sc}, Y_{sc} the ports to communicate structural changes; X and Y , the input and output ports; and N , the least set with the following structure: $\langle C, \{I_i\}, \rho_N, \rho_\lambda \rangle$,

where

C	set of components that are of type DYNPDEVS
$\{I_i\}$	I_i is set of influencers for $i \in C$
$\rho_N : S^n \times X_{sc} \rightarrow N$	network transition function
$\rho_\lambda : S^n \rightarrow Y_{sc}$	structural output function

with $S^n = \times_{d \in C} \oplus_{d \in C} Y_{sc}^d$ and for which the following reachability property holds:

$$\forall n \in N : n = n_{init} \vee \exists n_0 = n_{init}, \dots, n_i = n : \exists x_k \in X_{sc} : \rho_N(S^{n_k}, x_k) = S^{n_{k+1}}$$

with $i > 0; k = 0, \dots, i-1; S^{n_k} \in S^{n_k}; S^{n_{k+1}} \in S^{n_{k+1}}; n_0, \dots, n_i \in N$.

LISTING 6.2**DYNPDEVS Coordinator**

```

1 when receive *, x or sc message
2   if message is * or x message
3     if message is * message
4       send * to  $d \in \text{IMM} = \{c \in n.C \mid \text{tonie}(c) = t\}$ 
5       wait for  $y_d$  messages from all children
6       send  $y_{\text{toParent}}$  message according to the
         influencer information  $I_i$ 
7       wait for  $x_{\text{fromParent}}$  message from parent
8     end if
9     forward x messages to
10       $\cup y_d \setminus y_{\text{toParent}} \cup x_{\text{fromParent}}$  to
11       $d \in \text{IMM} \cup \text{INF}$  according to  $I_i$ 
12      wait for  $sc_d$  message from all children
13       $yr := n.p_n(ysc_1, \dots, ysc_m)$ 
14      send  $sc(yr)$  to parent
15    end if
16    receive sc message from parent
17    send sc message to children
18    wait for done messages from any activated children
19     $n := n.p_n(ysc_1, \dots, ysc_m, sc)$ 
20     $\text{tonie} := \text{minimum}(\text{tonie}_n)$ 
21     $\text{tole} := t$ 
22    send done(tonie) message
23 end when

```

An incarnation of the current network, n , is associated with each coordinator. The network transition is based on the information provided by the output ports $ysc_1, \dots, ysc_n$ of its components. Structural change requests may need to be propagated bottom-up and top-down in the hierarchy [10]. Requests that are not directed to the network itself are sent to the coordinator's parent, so they can be processed by an upper level. The role of the function p_n is to take the different requests from its components, which are available via the structural change ports Y_{sc} of its components and the input that has reached the coordinator via X_{sc} , and to invoke the network transition based on this information (1.19, Listing 6.2). This produces a new network incarnation. Afterward, the time of the next event is sent to the parent via the done message (1.22).

6.2.2.1 Virus Attack in DYNPDEVS

Let us illustrate (see also Figure 6.2) the functionalities of DYNPDEVS by the virus example. As soon as the cell receives as input the phage's DNA, it will enter the lytic or lysogenic cycle, which depends on the concentration of Cro and CI. The decision will be made as part of the external transition function δ_{ext} depending on the local situation, which involves the concentration of the key players.

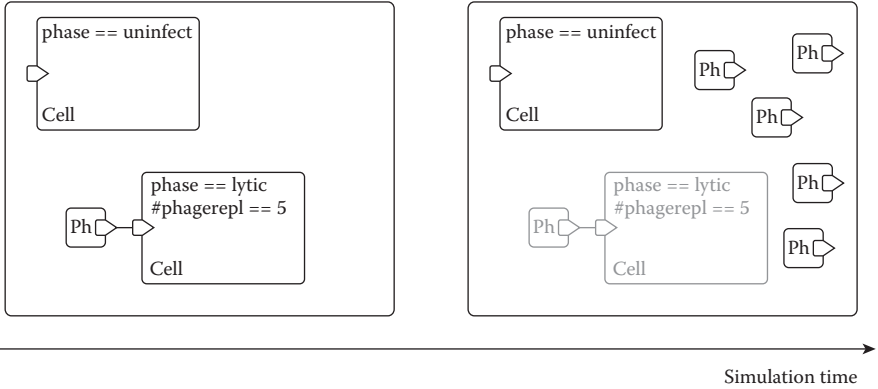


FIGURE 6.2 The virus example in the DYNPDEVS formalism.

The ρ_α will be invoked to enter the lytic cycle. Thus, a new incarnation that models the lytic cycle will replace the incarnation of the uninfected cell. In the lytic cycle, the internal transition function is invoked repeatedly and counts the number of new phages until the resources of the cell are depleted. At that point, another structural change is initiated. Via ρ_λ , the cell will generate as many new phage model copies as indicated by its internal state, and will remove itself and the lambda phage that infected it. This structural change will take effect at the level of the network, where the network transition function ρ_N is invoked to generate a new network incarnation that no longer contains the cell model, but all generated phage models instead.

Entering the lysogenic cycle implies that the lambda DNA is integrated into the cell's DNA. This will lead to certain changes in behavior, for example, inserting lambda DNA into the DNA of *E. coli* helps the bacterium to create useful amino acids for the digestive system. Thus, the incarnation of the cell that represents the uninfected cell will be replaced by an incarnation that models the lysogenic cycle. For this structural change, ρ_α will be invoked. If the cell replicates itself later, it will replicate the lysogenic cell incarnation by invoking the ρ_λ function specific to the lysogenic cycle, which results in the invocation of the network transition function. The newly generated cells will all be in the lysogenic cycle. If certain environmental factors change later on, those lysogenic cells might enter the lytic cycle triggering the processes described above.

6.2.3 ρ -DEVS

In DYNPDEVS, variable structures in M&S refer to the change of behavior pattern, the change of interaction structure, and the change of composition—but not to a dynamic interface. This is not surprising, as the distinction between the system and environment, and maintaining this distinction, has been traditionally emphasized in systems theory [31]. A system seems more likely to change its composition, its interaction structure, or its behavior pattern, rather than its interface to its environment [9]. However, some systems are characterized by just that: a plasticity of their

interface with which they signalize significant changes to the external world. Thus, dynamic interfaces have received some attention recently [11,32,33].

The ρ -DEVS (see definitions 6.2.3 and 6.2.4) is based on DYNPDEVS, but unlike DYNPDEVS with its static interface, ρ -DEVS supports changing input and output ports. Without the static boundary, the set of incarnations is now merged into one unity by the internal continuity of processing. The difference between the two formalisms DYNDEVS and ρ -DEVS is rather small at the level of atomic models. The input and output ports are becoming part of the incarnations $m_i \in M$.

Definition 6.2.3

An atomic ρ -DEVS model is the structure $\langle m_{init}, M, X_{sc}, Y_{sc} \rangle$ with $m_{init} \in M$ being the initial model, X_{sc} and Y_{sc} the ports to communicate structural changes, and M the least set with the following structure: $\langle X, Y, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \rho_\alpha, \rho_\lambda, \lambda, ta \rangle$, where

X, Y	structured sets of inputs and outputs
S	structured set of states
$s_0 \in S$	initial state
$\delta_{int} : S \rightarrow S$	internal transition function
$\delta_{ext} : Q \times X^b \rightarrow S$	external transition function, with
$Q = \{(s, e) : s \in S, 0 \leq e < ta(s)\}$	state set including elapsed time
$\delta_{con} : S \times X^b \rightarrow S$	confluent transition function
$\rho_\alpha : S \times X_{sc} \rightarrow M$	model transition function
$\rho_\lambda : S \rightarrow Y_{sc}$	implied structural network changes
$\lambda : S \rightarrow Y$	output function
$ta : S \rightarrow \mathbb{R}^{\geq 0} \cup \{\infty\}$	time advance function

and for which the following reachability property holds:

$$\forall n \in M : n = m_{init} \vee \exists m_0 = m_{init}, \dots, m_i = n : \exists x_k \in X_{sc} : \rho_\alpha(s^{m_k}, x_k) = s^{m_{k+1}}$$

with $i > 0; k = 0, \dots, i-1; s^{m_k} \in S^{m_k}; s^{m_{k+1}} \in S^{m_{k+1}}; m_0, \dots, m_i \in M$.

The simulator for the ρ -DEVS atomic model equals the abstract simulator of DYNPDEVS (see Listing 6.1), the only difference is that the information about ports has to be sent to the coordinator (see Listing 6.4), thus the `done` message contains the time of the next event and the set of ports of the current incarnation.

LISTING 6.3

Pseudo Code of ρ -DEVS Simulator

```

1 when receive *, x or sc message
2   // See Listing 6.1
3   send done (XYports(m), tonie) message
4 end when
```

To support hierarchical and modular modeling, a network structure is introduced in ρ -DEVS as well. A structural change means a change of interaction and composition structure, just as in DYNPDEVS. In addition, ports can change and *multicouplings* are introduced (see Definition 6.2.5).

Definition 6.2.4

A reflective, higher order network, a ρ -NDEVS, is the structure $\langle n_{init}, N, X_{sc}, Y_{sc} \rangle$ with $n_{init} \in N$ being the start configuration, X_{sc} and Y_{sc} the ports to communicate structural changes, and N the least set with the following structure: $\langle X, Y, C, MC, \rho_N, \rho_\lambda \rangle$ where

X	set of structured inputs
Y	set of structured outputs
C	set of components that are of type ρ -DEVS
MC	set of multicouplings
$\rho_N : S^n \times X_{sc} \rightarrow N$	network transition function
$\rho_\lambda : S^n \rightarrow Y_{sc}$	structural output function

with $S^n = \times_{d \in C} \oplus_{d \in C} Y_{sc}^d$ and for which the following reachability property holds:

$$\forall n \in N : n = n_{init} \vee \exists n_0 = n_{init}, \dots, n_i = n : \exists x_k \in X_{sc} : \rho_N(S^{n_k}, x_k) = S^{n_{k+1}}$$

with $i > 0; k = 0, \dots, i-1; S^{n_k} \in S^{n_k}; S^{n_{k+1}} \in S^{n_{k+1}}; n_0, \dots, n_i \in N$.

In addition, similar to the DYNPDEVS definition, ρ -NDEVS has to satisfy the following constraint: The application of ρ_N preserves the state and the structure of models that belong to the “old” and the “new” composition of the network. C is the set of components. Components that are newly created need to be initialized. The initial state of a component is given by the model m_{init} being in its initial state s_0 [9]. The outputs of the components over Y_{sc}^d form the quasi state, on which the structural output function and the network transition are based. The structural output function defines, given a component’s structural outputs, what shall be made available to the coupled model further up. The network transition takes this information into account, as well as the structural input information from X_{sc} , to determine the next network incarnation. A difference to DYNPDEVS is the introduction of multicouplings. The idea behind multicouplings is to make use of the information of the components’ available ports and allow a dynamic coupling between models.

Couplings are defined as 1:1, 1:n, n:1, or n:m relationships between sets of components. Taking part in these couplings is based on the availability of ports. In the following, we give the extensional definition of multicouplings.

Definition 6.2.5

A multicoupling $mc \in MC$ is defined as a tuple

$$mc(\text{port}_s, \text{port}_t) = \langle \{(C_{src} \text{ port}_s) \mid C_{src} \in C\}, \{(C_{tar} \text{ port}_t) \mid C_{tar} \in C\}, \text{select} \rangle$$

with $\text{select} : 2^c \rightarrow 2^c$.

The function `select` determines how the values are distributed from output to input ports. In regular DEVS, if more than one input port is linked to an output port, each output will be cloned and sent to all connected input ports. This standard strategy is meaningful if information shall be broadcasted. It is not a good strategy for propagating consumable resources (e.g., molecules or work pieces). A random selection strategy can be utilized for the latter [11].

Again the coordinator for ρ -DEVS looks similar to the one for DYNPDEVS. The only changes are that now the information about multicouplings will be evaluated (and not the set of influencers I_i ; 1.6, 1.11 Listing 6.4) and the coordinator has to be informed about possibly changed ports of components (1.22 Listing 6.4), as this information is crucial for evaluating the multicouplings correctly. The port information is updated for all components as soon as the done message has been received.

LISTING 6.4

ρ -DEVS Coordinator

```

1  when receive *, x or sc message
2  if message is * or x message
3    if message is * message
4      send * to  $d \in \text{IMM} = \{c \in n.C \mid \text{tonie}(c) = t\}$ 
5      wait for  $y_d$  messages from all children
6      send  $y_{\text{toParent}}$  message according to MC
7      wait for  $x_{\text{fromParent}}$  message from parent
8    end if
9    forward x messages to
10      $\cup y_d \setminus y_{\text{toParent}} \cup x_{\text{fromParent}}$  to
11      $d \in \text{IMM} \cup \text{INF}$  according to MC
12     wait for  $sc_d$  message from all children
13      $yr := n.p_k(y_{sc_1}, \dots, y_{sc_m})$ 
14     send  $sc(yr)$  to parent
15   end if
16   receive sc message from parent
17   send sc message to children
18   wait for done messages from any activated
    children and update port information
19    $n := n.p_N(y_{sc_1}, \dots, y_{sc_m}, sc)$ 
20    $\text{tonie} := t + \text{minimum}(\text{tonie}_n)$ 
21    $\text{tole} := t$ 
22   send  $\text{done}(XYports(n), \text{tonie})$  message
23 end when
```

6.2.3.1 Virus Attack in ρ -Devs

Introducing variable ports and the multicouplings facilitates the modeling even further. Cells carrying a lambda prophage, which means the cell entered the lysogenic

cycle and the lambda phage DNA has been integrated, are immune for further infections, which can nicely be captured by removing the ports where the lambda phage typically binds to. Thus, the lambda phage can only bind to those cells that are still open for an infection.

In addition, the DYNPDEVS model requires a lambda phage to be explicitly coupled to a cell for infecting it. Due to the extensional definition of multicouplings in this case, the lambda phage would induce a corresponding update of the multicouplings via ρ_λ , for example, to signal that it would like to be coupled to a cell and its inPhage port.

An intensional representation of multicouplings at network level, for example, $((\text{outPhage}, \text{inPhage}), n : 1, \text{Select}) \in \text{MC}_{\text{intensional}}$ would imply that $\forall c_1, c_2 \in C \wedge c_1 \neq c_2 \wedge \text{name}(c_1.x) = \text{inPhage} \wedge \text{name}(c_2.y) = \text{outPhage}$ there is a $\text{mc} \in \text{MC}_{\text{extensional}}$ of the form $\langle C_{\text{src}}, \{(c_1 \text{inPhage})\}, \text{Select} \rangle$ and $(c_2 \text{outPhage}) \in C_{\text{src}}$. This allows a more comfortable modeling.

Multicouplings would be updated automatically by merely adding ports at the network level. The model with the changed ports would not need to know anything about its environment. The network model would contain more information (i.e., on the intensions of its couplings) that can then be applied automatically.

Figure 6.3 shows that in ρ -DEVS a newly released lambda phage generates its port to dock to a cell after a certain delay. The multicoupling then takes care that it is coupled to an uninfected cell, enabling an infection.

These two variants of DEVS illustrate nicely the potential for reuse, since both formalisms have more in common than what distinguishes them. Even the original PDEVS formalism is still very visible in the conception of both. The more we add features from other formalisms, for example, the idea of multicouplings and variable ports has been adopted from PI-CALCULUS [34], the more the similarity with the original formalism vanishes. This will become obvious in another extension also; that is, ML-DEVS [35], which we only shortly sketch in the following.

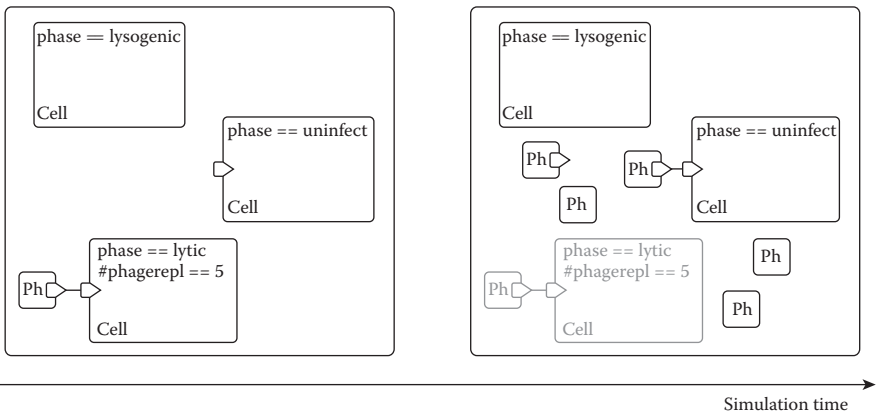


FIGURE 6.3 The virus example in the ρ -DEVS formalism.

6.2.4 ML-DEVS

The idea behind dynamic ports and multicouplings in ρ -DEVS has been to use the knowledge of the coupled level, but still to associate the initiating dynamics with the atomic model. ML-DEVS moves even one step further. Although a hierarchical composition of models is accomplished by coupling models, the behavior of coupled models is completely determined by their constituents and how they are coupled with each other. The formalism's inherent reductionism sometimes burdens modeling. This can be the case when certain events within a system depend on the overall state of the system, for example, chemical reactions may depend on pressure, temperature, and the concentrations of the involved species. This macrolevel perspective has its own state and rules of behavior and is complemented by the microlevel view where many individuals are described in detail and form a population. Obviously, macro and microlevels are not independent. The macrolevel influences the behavior of the individuals, the so-called downward causation, and the individuals' behavior also has an impact on macrovariables, the so-called upward causation. There is no direct way in DEVS to conveniently combine such macro and microlevels of a system and to describe their interdependencies; that is, a property of the whole system that influences the state of a submodel, or how submodels determine the behavior at the macrolevel.

As a workaround, coupled models are often equipped with submodels that serve to manage the high-level properties of the whole coupled model: this would be the *BulkSolution* model in the Tryptophan model from [36], which handles the number of existing enzymes and additional dynamics, or this would be the cell in our lambda phage model. Obviously, there are several problems with this approach:

- Downward and upward causation are realized by exchanging events asynchronously. Communication by asynchronous events is entirely appropriate between independent subsystems that react to these external perturbations according to their internal rules of behavior. However, whether downward causation and upward causation between the macro- and microlevel of a system is best described by exchanging events appears doubtful.
- Hierarchical composition of DEVS models and the metaphors of coupled and atomic model are misleading. Although the coupling of models might be associated with a compositional hierarchy of systems [37], coupled models serve merely as a container for their submodels. They neither have a state nor a dynamic of their own.
- If micro- and macrolevels shall be supported, atomic models have to function as macromodels and interact with all micromodels. For realizing upward and downward causation, their ports have to be coupled with the ports of all micromodels. Additionally, protocols for exchanging information from the microlevel to the macrolevel and vice versa have to be realized. This has to be done for each macro- and all micromodels, which burdens M&S significantly.

Since there is no satisfactory way to circumvent the resulting difficulties with standard DEVS, it might be worth considering another possibility: to integrate a

high-level model within the coupled DEVS model itself. This is not a new idea. Faced with the problem of introducing variable structures (changing composition and interaction patterns) into the DEVS formalism, the DSSN equips the coupled model with a behavior of its own [27]. The introduced executive of a coupled model is a hybrid between the atomic and coupled model, and can access the structure of the entire model. In RUISEM [38], an executive is introduced that is able to change the couplings between components on demand, and thus to realize a Jibshop model efficiently in DEVS. Also the ideas of multicouplings in ρ -DEVS already address the desire to enrich the network model with some dynamics of its own. However, in all these approaches downward and upward causation are only supported rudimentary.

Thus, a new formalism was introduced (i.e., ML-DEVS); [35]. In ML-DEVS the coupled model is equipped with a state and a behavior of its own, such that the macrolevel does not appear as a separate unit (an executive) of the coupled model. Secondly, we have to explicitly define how the macrolevel affects the microlevel and vice versa. Both tasks are closely interrelated.

Obviously, one means to propagate information from the macro- to the microlevel is to exchange events between models. However, this is rather tedious (e.g., in case the dynamics of a micromodel has to take the global state into consideration). Therefore, we will adopt the idea of value couplings. The information at the macrolevel is mapped to specific port names. Each micromodel may access macrovariables by defining input ports with corresponding names.

In the opposite direction, the macrolevel needs access to crucial information at the microlevel. For this purpose, we equip micromodels with the ability to change their ports and to thereby signalize crucial state changes to the outside world. Upward causation is supported, as the macromodel has an overview of the number of micromodels being in a particular state (i.e., exhibiting a particular set of ports) and to take this into account when updating the state at macrolevel.

Another question is, how are models “activated”? We assume that at the macrolevel as well as at the microlevel, models are still triggered by the flow of time and the arrival of events. In addition, the macrolevel can directly activate its components by sending them events, thereby, it becomes possible to synchronously let several micromodels interact, which is of particular interest when modeling chemical reactions in biological systems [33]. Additionally, the dynamics at the macrolevel can be activated by the dynamics at the microlevel, for example, if the number of components being in a certain state (signalized by their ports) surpasses a certain threshold. Therefore, a form of invariant is defined at the macrolevel, whose violation initiates a transition at the macrolevel. This is inspired by the ideas of hybrid state automata [39], where the discrete state changes are triggered at the moment the continuous dynamics lead to threshold crossing.

The approach makes use of different concepts. It adopts the idea of ρ -DEVS, as the possibility of variable ports is central to propagate knowledge bottom-up without accessing the models explicitly. However, the “reflective” strategy of DYNDVS and ρ -DEVS, which introduce structural changes bottom-up, is no longer pursued, as in ML-DEVS the macrolevel has a state and behavior of its own, an ability that can be exploited for variable structure models. Introducing new components and couplings top-down bears more similarity to DSSN [27] and the modeling approach in EMSY

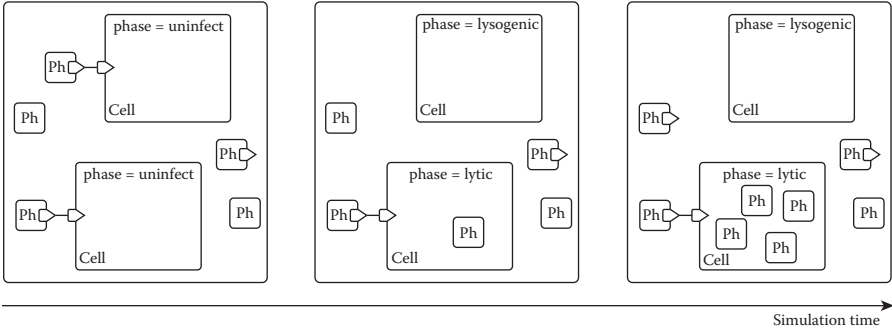


FIGURE 6.4 The virus example in the ML-DEVS formalism.

[40], the similarity with the latter is no coincidence as EMSY as well as ML-DEVS have been motivated by biological systems applications.

6.2.4.1 Virus Attack in ML-Devs

In ML-DEVS the model now looks differently. The cell can be modeled as a coupled ML-DEVS model. Inside the cell, global property-like concentrations of key players and reactions to update those are described. In the lytic cycle, lambda phage models are generated as components of the cell. Already before being released, the generated lambda phages are treated as “first class objects,” which also describe their dynamics within the cell in a more adequate manner. All other features that ML-DEVS shares with DYNPDEVS and ρ -DEVS are exploited as well; that is, different incarnations, variable ports, and multicouplings (see Figure 6.4).

6.2.5 EPI-DEVS

Whereas variable structure models imply major changes in the structure of the DEVS formalism, another kind of extension requires comparatively little changes. EPI-DEVS (external process interface-DEVS) has been developed to allow the integration of external processes into the DEVS models [41] and thus EPI-DEVS is a “Software-in-the-loop DEVS.” Additional ports are introduced, and the domain and range of the original DEVS transition functions are adapted. There are different variants: unpaced and paced EPI-DEVS, and variants that also support variable structure models. Paced EPI-DEVS is a best-effort variant, running in “real time”; unpaced EPI-DEVS can use explicit time models to synchronize the externally running processes and the simulation.

Similarly, as the different variable structure variants, the developments of new formalisms are motivated by the requirements of applications. The motivation of EPI-DEVS has been the simulation-based evaluation of software agents, but it works for all external software. Testing activities support quality assurance by gathering information about the nature of the software being studied. Same as the functionality of real-time systems and embedded systems, the functionality of agent systems cannot be evaluated based on an a priori fixed input specification, but only as a course of reactions to the evolution of an environment. As agents are aimed at working in dynamic environments, simulation seems a natural approach toward analyzing the

behavior of an agent system in interaction with its environment. Its interaction with the environment has to be observed over a period of time. The usage of a virtual environment in contrast to the real environment typically reduces costs and efforts and allows testing system behavior in “rare event situations.” Virtual environments are easier to observe and to control, and probe effects are easier to manage. To use simulation throughout the entire software development process, we introduced different variants of EPI-DEVS. In [5], an environment has been modeled for the AUTOMINDER software, developed to assist elderlies in their environment [42], in order to evaluate the behavior of the software when confronted with different types of elderlies. An EPI-DEVS model forms the interface between the simulator and AUTOMINDER. Here the simulation and the software interact asynchronously, therefore a paced simulator is used to execute the EPI-DEVS models, and no explicit time model is needed.

6.2.6 MODEL COMPONENTS

A component should be a replaceable part of a system and be usable in an unforeseen context for different purposes [43]. Thus, to become a component, a model needs to announce its functionality by a well-defined interface [44]. The main challenge for compositional approaches is summarized in the term compositionality, which requires that the meaning of a composition can be derived solely from the semantic descriptions of the parts together with the rules of combination [45]. Parts have compositional properties if the semantics of a composition may be derived from the semantics of the parts [43]. Therefore, interfaces should contain all information that is necessary for composition, but not more, as the latter could hamper the analysis [46]. However, although modular-hierarchical modeling approaches like DEVS, but also Modelica [47], and Ptolemy [48] let models exhibit ports to indicate that certain kinds of events may be received or sent, they do not treat interface descriptions as first class entities. In our approach we decided that interfaces should exist in their own right, so that they can be published and analyzed independently [49–51].

The first step for introducing interfaces as separate units of definition was to increase the flexibility for defining and relating the units for data exchange (events). Compared to basic set theory, type systems [52] support modular definition units [51]. By separating interface definitions from their model implementations, compositions can be analyzed solely based on interface definitions. Interfaces are represented as XML documents, and as such can be stored in databases. The compatibility to type definitions can be analyzed based on XSD [51]. As XSD is compatible to standards like SAWSDL, semantic annotations are facilitated. The refinement between interfaces and implementations, the completeness of compositions, and the correctness of components can be checked. Correct components fulfill all requirements to be deployed and ensure that a proper simulation model can be derived [51]. “Proper” refers here to the first three to four levels of model composability [53]. It covers the syntactic level and part of the semantic level, but forms a suitable basis to climb further on.

The discussed DEVS variants and the component-based approach toward modeling are aimed at an effective modeling in DEVS, by facilitating describing the system of interest and the construction of models. In addition, efficient simulators for DEVS models are needed.

6.3 ALGORITHMS AND DATA STRUCTURES FOR SIMULATING DEVS MODELS

An efficient execution of models based on one of the DEVS formalisms is mandatory for a widespread usage of those. It imposes several challenges, comparable to any other discrete-event modeling paradigm:

- The simulation algorithm as such
- Event queues
- Auxiliary functionality, supporting the creation and maintenance of efficient runs (partitioning, load balancing)

In the following, we describe several alternative algorithms for the computation of DEVS models, followed by partitioning and load-balancing methods, as well as some adapted event queue implementations.

6.3.1 SIMULATION ALGORITHMS

DEVS modeling formalisms usually ship with an “abstract simulator,” which defines the execution semantics of the modeling formalism. Traditionally there is an extra algorithm for each model formalism entity (i.e., for coupled and atomic models). These algorithms communicate by using messages, and depending on the formalism may block each other but at least need to be synchronized.

6.3.1.1 Abstract Threaded

The abstract threaded simulator is an implementation of the “abstract simulator,” which is usually defined for each DEVS variant. The algorithm given here is the one for the PDEVS formalism. A direct implementation of this algorithm requires a thread for each coordinator and simulator, which is usually a hard limit to the size of models (see [Figure 6.5](#))—at least as long as there is no technique like a “virtual” thread pool.

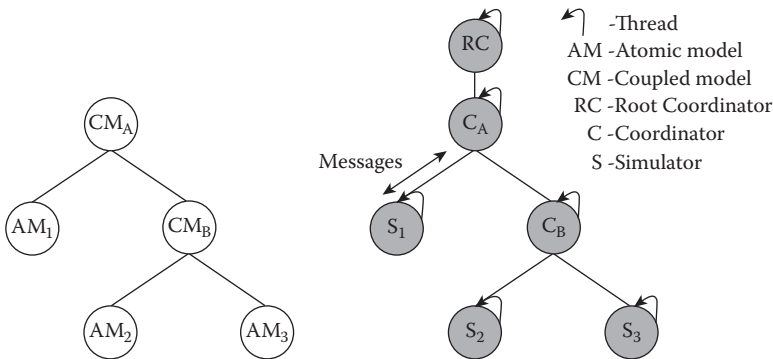


FIGURE 6.5 Mapping of the abstract threaded simulator tree.

LISTING 6.5**Pseudo Code of the Abstract Threaded PDEVS Coordinator**

```

1 when receive * or x message
2   if message isA * message then
3     for each processor in imminents do
4       send * message to processor
5     for each imminent in imminents do
6       wait for y message
7       send y message to parent
8       wait for x message
9     end if
10    for each influencee do
11      send x message to influencee
12    for each processor in union
      (influencees, imminents)
13      wait for done message
14      send done message to parent
15  end when

```

LISTING 6.6**Pseudo Code of the Abstract Threaded PDEVS Simulator**

```

1 when receive * or x message
2   if message isA * message then
3     execute model.lambda
4     send y message to parent
5     wait for x message
6     if empty (x) then
7       execute model.deltaInt
8     else
9       execute model.deltaCon
10    end if
11  else
12    execute model.deltaExt
13  end if
14  execute model.timeAdvance
15  send done message to parent
16 end when

```

The pseudo codes of the coordinators and the simulators are given in Listing 6.5 and 6.6, respectively. Both algorithms provide support of the standard PDEVS communication protocol (i.e., they wait for incoming messages) execute the corresponding actions, and finally send back the answers.

6.3.1.2 Abstract Sequential

The abstract sequential algorithm is pretty close to the abstract threaded simulator. There is still an explicit coordinator (Listing 6.7), respectively, an explicit simulator (Listing 6.8) per model entity. This simulator has been used by Fillipi for his JDEVS realization, where it simulates DEVS models (2008). We implemented an adapted version for PDEVS. The process of computing a simulation pulse is split up into two phases:

1. Creation of outputs (by executing the λ -function) $\rightarrow$ `getOutputs`
2. Computing the state transitions $\rightarrow$ `doRemainder`

This simulator has only one thread of control, thus there is no real parallelism (which is sad in regards to multicore CPUs), but the number of threads is not a limiting factor anymore, which it can be for the abstract threaded variant. However, we still need an extra instance of the corresponding simulation algorithm per model entity (see [Figure 6.6](#)).

If the `getOutputs` method of a coordinator is called, it will call the `getOutputs` method of all imminent children. If the `doRemainder` method is called, it “sends” the passed messages to the influenced children. Both methods are called for a “star message,” only the latter is executed for an “external message.”

LISTING 6.7

Pseudo Code of the Abstract Sequential PDEVS Coordinator

```

1 Function Messages getOutputs ()
2   for each child in imminents do
3     allmsgs=union (allmsgs, child.
4       getOutputs ())
5     influencees=propagateIC (allmsgs)
6     allmsgs=getEC (allmsgs)
7   return allmsgs
8 end Function
9 Function double doRemainder (msgs)
10  influencees=union (influencees,
11    propagateIC (msgs))
12  for each child in union(influencees,
13    imminents) do
14    setTonic (child, child.doRemainder())
15  end Function
```

LISTING 6.8

Pseudo Code of the Abstract Sequential PDEVS Simulator

```

1 Function Messages getOutputs ()
2   execute model.lambda
3   return getMsgs (model)
4 end Function
```

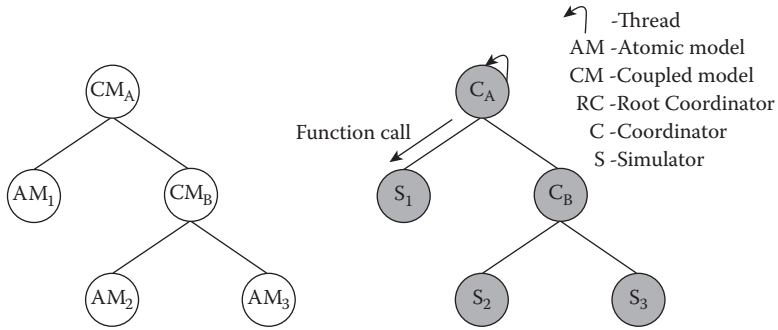


FIGURE 6.6 Mapping of the abstract sequential simulator variant.

```

5 Function double doRemainder (msgs)
6   execute model.stateTransition
7   execute model.timeAdvance
8   return getTonic (model)
9 end Function

```

The simulator, comparable to the coordinator, is based on the two different methods as well. If it is an imminent simulator, the `getOutputs` method will be called to compute λ . If it is imminent or influenced, the `doRemainder` method takes care of executing the right state transition method.

6.3.1.3 Hierarchical Sequential

In the hierarchical sequential simulator, we no longer have a simulation algorithm per model entity, but we have just one recursive algorithm (see Figure 6.7). This algorithm is created following the “phases” idea of the abstract sequential algorithm—the outputs are computed in the first recursion (Listing 6.9), the state transitions are computed in the second recursion (Listing 6.10).

LISTING 6.9

Pseudo Code of a Hierarchical PDEVS Simulator: Part I

```

1 procedure partI (aCoupledModel)
2   for each model in imminent
     (aCoupledModel) do
3     if model isA AtomicModel then
4       execute model.lambda
5     else
6       partI (model)
7     end if
8     influencees := union (influencees,
        copyValues (aCoupledModel, model,
        internal))

```

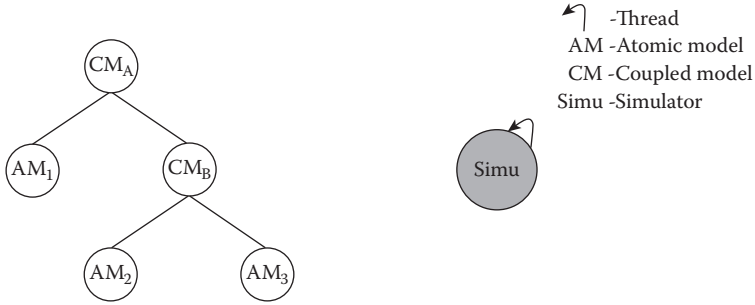


FIGURE 6.7 Mapping of the hierarchical sequential simulator

```

9   copyValues (aCoupledModel, model,
               extOut)
10  end for

```

LISTING 6.10

Pseudo Code of a Hierarchical PDEVS Simulator: Part II

```

1  procedure partII (aCoupledModel)
2  influences := union (influences,
                       copyValues (aCoupledModel,
                                   aCoupledModel, extIn))
3  for each model in union (influences,
                           imminents) do
4    if model isA AtomicModel then
5      execute model.stateTransition
6      execute model.timeAdvance
7    else
8      partII (model)
9    end if
10 end for

```

6.3.1.4 Flat Sequential

The flat sequential simulator virtually flattens a hierarchical DEVS model, so that it can be computed with a single simulation algorithm in a very efficient manner (Figure 6.8). However, the hierarchical structure is kept on the model's side, so that any instrumentation can be kept, and thus simulation results can be observed independently from the simulation algorithm used. As can be seen in Listing 6.11, the flat sequential simulator needs only two loops—the first one is responsible for the first pulse of the classic PDEVS protocol (i.e., output generation and the propagation of output value to parental nodes). The second loop copies values from the parental nodes to the children and executes pending state transitions and the `timeAdvance` method.

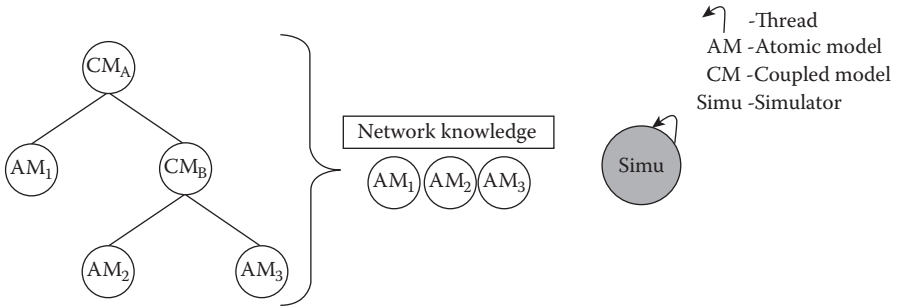


FIGURE 6.8 Mapping of the flat sequential simulator.

LISTING 6.11**Pseudo Code of a Flatsequential PDEVS Simulator**

```

1 for each model in imminents do
2   execute model.lambda
3   influences := union (influences, copyOutputs
                        (model))
4 end for
5 for each model in union (influences,
                          imminents) do
6   execute model.stateTransition
7   execute model.timeAdvance
8 end for

```

6.3.1.5 Parallel Sequential

The parallel sequential simulator [18] combines the advantages of the simulation algorithms presented so far. On the one side it allows the existence of simulator instances for single entities of the model, and on the other side it allows having instances of the flat sequential algorithm for (subsets of) the children of a coupled model.

The pseudo code of virtual and subcoordinators can be found in the Listings 6.12 and 6.13, respectively. Figure 6.9 sketches the basic idea.

LISTING 6.12**Pseudo Code of the Virtual Coordinator**

```

1 when receive * or x message
2   if message is * message
3     for each subCoordinator do
4       send * message to subCoordinator
5     for each subCoordinator do

```

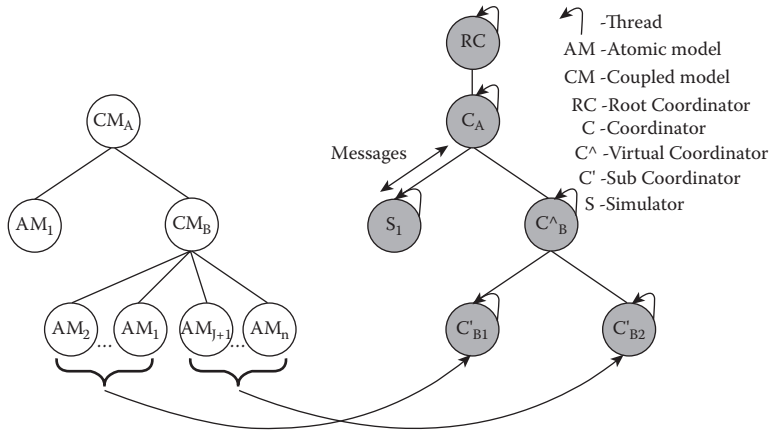


FIGURE 6.9 Mapping between the PDEVS model tree and the partial simulator tree.

```

6  wait for composite message
7  send y message to parent
8  wait for x message
9  end if
10 create composite messages for
    subCoordinators
11 for each subCoordinator do
12   send composite message to
    subCoordinator
13 for each subCoordinator do
14   wait for done message
15   send done message to parent
16 end

```

LISTING 6.13

Pseudo Code of the Subcoordinator

```

1 when receive * or composite message
2   if message is * message then
3     for each child in imminent do
4       if isLocal(child) then
5         compute Outputs
6       else
7         send * message to child
8       end if
9     for each non local child do
10      wait for y message

```

```

11     send composite message to parent
12     wait for composite message
13 end if
14 for each child in influenced do
15     if isLocal(child) then
16         compute state transition
17         compute timeAdvance
18     else
19         send x message to child
20     end if
21 for each non local child do
22     wait for done
23     send done message to parent
24 end

```

6.3.1.6 Reuse

Reuse of algorithms is still a hot topic, and will most likely remain one in the future. In [54], we described a template for the creation of a simulation algorithm family for DEVS formalisms. We identified the possibility of splitting up the simulation code of a DEVS coordinator or simulator into three phases, which we labeled with “pre,” “do,” and “post.” Most of the code belongs to the “do” part, which is usually responsible for the state transition. “Pre” and “post” take care of additional jobs, for example, computation of time advance, pacing the execution, charging ports in EPI-DEVS, a.s.o. In these cases, the “do” part might be reusable across formalism boundaries.

6.3.1.7 Results

In [17,18], we compared the algorithms listed above. In Figure 6.10 we give an updated excerpt of these results. For each data point in the left chart we used the worst, and for each data point in the right chart we used the best performing combination of simulation algorithm, event queue, and problem size; thus, a different queue might be responsible for every worst/best case data point. Comparing the worst to the best case setup reveals that the abstract threaded for a model size of 1600 atomic models performs better in the best case than the hierarchical sequential in the worst case. Comparing the performance of the hierarchical simulation algorithm in both figures (problem size 150) reveals that using the right event queue can lead to a speed up of more than 700. In the best case the abstract threaded needs for a model comprising 1600 atomic models already as much time as the flat sequential for a model of 22,500 atomic models (each atomic model has to go through the same number of internal events). Our results show that the flat sequential algorithm outperforms the others for the model used for testing. In addition, the results show that the performance significantly depends on the event queue implementation used, and thus on using the wrong event queue, even a pretty efficient algorithm can have a bad overall performance.

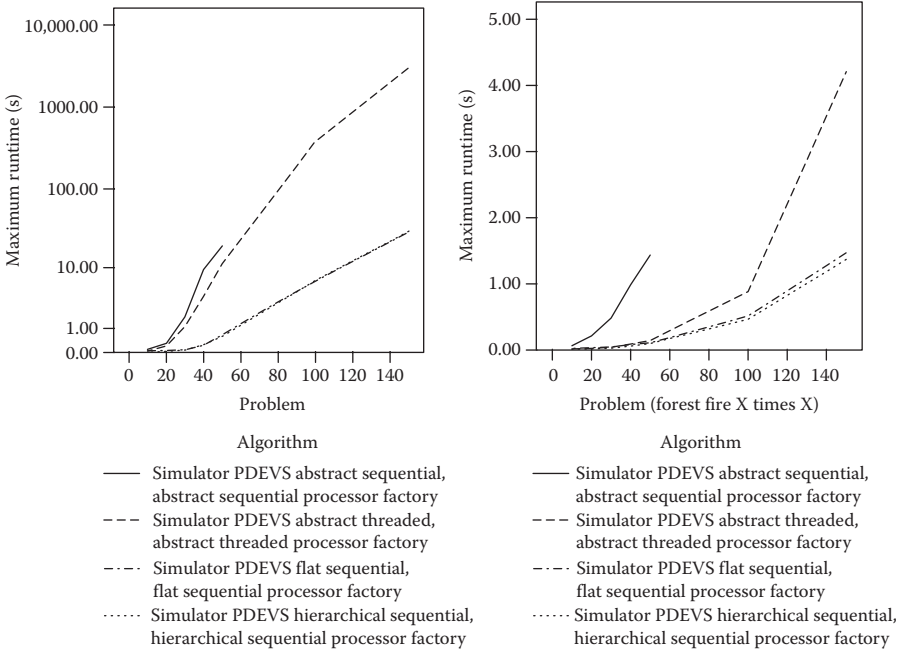


FIGURE 6.10 Excerpt of the results from a comparison of sequential simulation algorithms.

6.3.2 PARTITIONING AND LOAD BALANCING

Distributed simulation requires that the model at hand can be split up into several parts, which will then be assigned to the available computing resources. Such an assignment is a prerequisite for efficient distributed execution: if it is not chosen carefully, the slowest processor might be responsible for simulating the most computationally expensive parts of the model, or two model entities that heavily interact with each other are placed on processors with a slow network connection in between. In both cases, the performance of the distributed simulation would be hampered dramatically, which could even lead to execution times exceeding those of sequential runs.

The process of automatically finding a good assignment between model entities and computing resources, which avoids the two aforementioned pitfalls, is called *partitioning*. The partitioning problem can be formalized with graphs. Let $G_M = (V_M, E_M)$ be a graph that represents a model. Each vertex $v \in V_M$ corresponds to a model entity and each edge $e = \{v_1, v_2\} \in E_M$ denotes that two model entities might interact with each other. Similarly, the available resources can be represented by an infrastructure graph $G_I = (V_I, E_I)$. Here, each vertex corresponds to an available processor, while edges express network connections. Introducing weights for edges and vertices allows reflecting heterogeneity in both the model and infrastructure: Each model graph vertex can be associated with a real number that quantifies the computational load that will be imposed on the processor, each model graph edge can be associated with a value that expresses the degree of interaction. For the infrastructure graph, vertex weights can be interpreted as the computational resources of a processor, and edge

weights would define the quality of the network connection between two processors. Now, we can define the partitioning problem as finding a mapping $\text{part}: V_M \rightarrow V_I$ with certain properties. A fundamental property is the *balance* of the load; that is, the fairness of the assignment when considering each processor's computational capabilities and the computational load assigned to it by the mapping. Given that w is the weight function and $M_{v_i} = \{v_m \mid v_m \in V_M \wedge \text{part}(v_m) = v_i\}$ with $v_i \in V_I$ is the set of model entities assigned to the processor represented by v_i , the balance can be formulated as:

$$\forall v_i \in V_I : \frac{w(v_i)}{\sum_{v \in V_I} w(v)} \approx \frac{\sum_{m \in M_{v_i}} w(m)}{\sum_{v \in V_M} w(v)}.$$

Another important property is the *cut size*, which estimates the amount of inter-processor communication and which should be minimized:

$$\sum_{e=\{v_1, v_2\} \in E_M \wedge \text{part}(v_1) \neq \text{part}(v_2)} w(e).$$

Depending on the actual problem, other properties—such as, for example, efficiently exploiting the best network connections for the edges with most weight in the cut—might be of interest as well. Fulfilling one of these properties may conflict with fulfilling another. For example, the cut size would be zero when all model entities are placed on one processor, but this would also result in a large imbalance. Where exactly a good trade-off between two properties is (e.g., allowing 5% imbalance for a 10% smaller cut size), highly depends on the given model and the infrastructure: The higher the computational load per entity, the more important is a balanced partition, but simulation of large, fine-grained models might require an optimization toward cut size instead.

In theoretical computer science, the graph partitioning problem is merely defined on the model graph and the balance of a partition is presumed, so that only the cut size needs to be minimized [55]. This problem is known to be NP-hard, and it is still NP-complete when deciding on the existence of a partition into two disjunct vertex subsets with a cut smaller than a given value [56]. Consequently, most practical partitioning algorithms strongly rely on heuristics or are targeted at certain graph classes.

Finally, one has to distinguish between *static* and *dynamic* partitioning. Static partitioning is the assignment of model entities to processors *before* the simulation starts, dynamic partitioning is repeated repartitioning *at runtime*. As the latter is also known as *load balancing*, we make this distinction by referring to partitioning and load balancing in the following sections.

6.3.2.1 Partitioning

Besides classical partitioning algorithms, such as the Kernighan–Lin (KL) method [57]; (see [55] for a good survey), there are also many partitioning algorithms that are tuned to a specific problem class. For example, [58] use an algebraic partitioning method that identifies symmetries in graphs. They apply their method to a simulation

of computer hardware, which usually exhibits various symmetries. In [59], multi-level partitioning algorithms for power law graphs are presented. Power law graphs are graphs for which the number of nodes with a certain degree (i.e., with a certain number of neighbors) is exponentially distributed. Important examples of power law graphs are networks of air traffic or protein interaction. A multilevel partitioning algorithm consists of three basic phases: coarsening, partitioning, and uncoarsening. The basic idea is to simplify the input graph by merging nodes that are likely to be assigned to the same processor. The resulting graph is usually much smaller, so that computationally expensive methods—which ensure a good quality of the partition—can be applied. In the uncoarsening phase, the original graph is rebuilt from the coarse graph and the partition, which results in the original graph and a (potentially adapted) partition of it. Both coarsening and uncoarsening are usually reiterated. A well-known and powerful library of multilevel graph partitioning algorithms is the METIS package [60], which is written in C. It can be easily integrated into any simulation system. Apart from a DEVS partitioning algorithm for optimistic simulation [19], the generic model partitioning method with lookahead k , GMP- k , can be used to partition hierarchical models [61]. It traverses the DEVS model tree top-down and calculates an estimated cost for all models that are encountered. These are distributed over the number of desired partition subsets. Better partitions are found by expanding the most costly vertex; that is, removing it and replacing it with all leaves of the vertex' subtree with maximal depth k . The leaves will then be redistributed over all subsets. This mechanism is particularly interesting because it estimates the computational cost of each node in the model graph [20]. This illustrates that the *actual* partitioning problem encountered in practice is not just to find a mapping between the weighted model and infrastructure graph, but also to retrieve the graph structures and weights in the first place. The hierarchical partitioning algorithm described in the following exploits the tree structure of the DEVS models in a similar manner, but avoids fragmentation to minimize cut size and allows the definition of partitioning constraints [21]. Two kinds of constraints are supported: the assignment of a model entity to a processor; that is, to define that $\text{part}(e) = p$ with $e \in V_M$ and $p \in V_I$. This is useful when executing a parallel-distributed EPI-DEVS simulation (see Section 6.2.5), since the model entities interfacing the external process, which is not part of the DEVS model, can be placed on the same machine manually. The other kind of constraint allows forcing two model entities onto the same processor; that is, it defines that $\text{part}(e_1) = \text{part}(e_2)$ for $e_1, e_2 \in V_M$. Such constraints can augment the partitioning quality by allowing the user to integrate additional knowledge, particularly if this knowledge cannot be captured easily by cost estimation functions. For example, the user could force the partitioning algorithm to put two model entities on the same machine, when it is known that these will interact heavily. The partitioning algorithm allows an arbitrary number of both kinds of constraints and can be applied to any model graph that is a tree; it is not restricted to DEVS variants.

6.3.2.2 Hierarchical Partitioning

Basically, the algorithm strives to split up the model tree into p large subtrees of equal size, given that $p = |V_I|$ is the number of available processors. This minimizes

the cut size, as only $p-1$ edges would be in the cut if the algorithm reaches its goal. The main algorithm is split up into three phases: a top-down phase, a partitioning phase, and a bottom-up phase. This is similar to the basic strategy in Ref. [61], and for the same reasons: as the DEVS model trees to be partitioned could be large and therefore expensive to traverse, it is advisable to restrict the analysis to the most important parts of the tree.

Before the top-down phase starts, the model tree has to be created in a preliminary phase. Each vertex is analyzed and its computational cost is calculated. Here, the cost functions from [20] could be employed. The top-down phase then steps down the model tree level by level, where one level is the set of all vertices with the same distance to the root. It stops when a suitable *partitioning level* is found; that is, there are sufficiently many vertices on the level to be distributed over p processors and there are no conflicts that need to be resolved. A conflict arises when at least two constraints in the subtree of any vertex in the current level prohibit it from being assigned to a single machine.

The partitioning phase applies various heuristics to split up all vertices in the partitioning level (see [21] for details). Each vertex represents the subtree of which it is the root, assigning the vertex to a processor therefore assigns the whole subtree to this processor. The partitioning phase has to take special care of all constraints that are defined on upper and lower levels of the model tree, these are incorporated into the partitioning decision. The main heuristic for the partitioning decision is the distance of the vertices on the partitioning level among each other. This is a simple measurement that can be calculated easily in the top-down phase, and which helps to identify large subtrees (see Figure 6.11).

Finally, the algorithm has to partition all vertices above the partitioning level. This is done in the bottom-up phase, which assigns all vertices to the processor to which its child vertices have been assigned. If the child vertices have been assigned to more than one processor, a voting scheme is applied. In case an upper level contains vertices without children (i.e., leaves of the model tree that are closer to the tree root than the partitioning level), the partitioning phase is reinvoked on this level. Depending on the number of leaves, these will be either assigned to the same processor as a neighbor subtree, or a new processor may be chosen. This can be implemented by adapting the voting scheme.

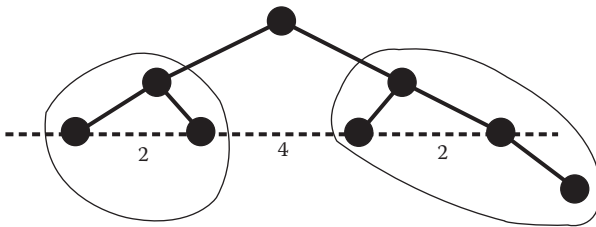


FIGURE 6.11 Calculation of distances for subtree identification. Given the current partitioning level (gray dashed line), distances allows identification of the largest subtrees of this simple tree (gray).

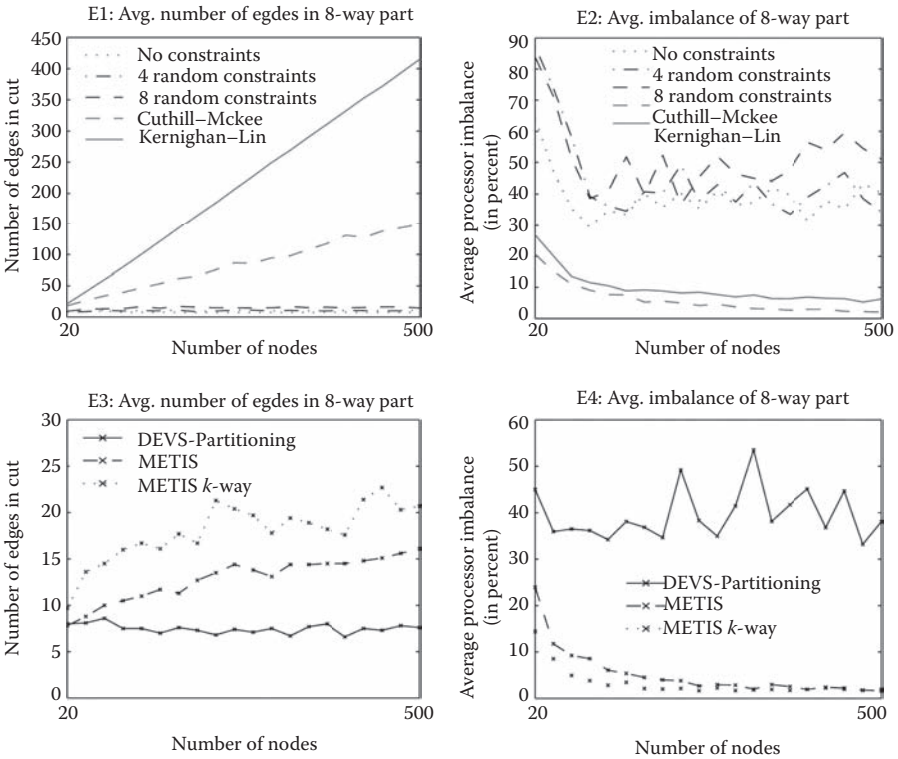


FIGURE 6.12 Results of the nonfragmenting DEVS partitioning with constraints.

6.3.2.3 Results

Comparing the algorithm to other general partitioning schemes like METIS or the KL algorithm yields the results as depicted in Figure 6.12. Please note that the other approaches are generally applicable, while the presented algorithm only works on trees. It is specifically tuned to the given problem and the comparison is therefore biased. Still, it is interesting to see the trade-off between minimizing the cut and minimizing the imbalance (E1/3 vs. E2/4, see Figure 6.12), and that the addition of constraints does not harm the overall performance, provided that the model tree is sufficiently large.

Future work should compare the performance of the partitioning algorithm to other DEVS-specific algorithms, as presented in [19,20,61].

6.3.2.4 Load Balancing

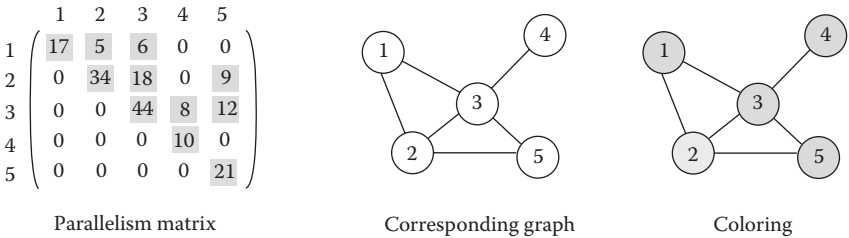
Load balancing (i.e., dynamic repartitioning of a distributed model at runtime) has received considerable attention over the past decade. It introduces various additional aspects to the already challenging partitioning problem. Load balancing algorithms in general can be defined by an *information policy* (i.e., defining which processor and network load statistics need to be aggregated and propagated), a

transfer policy that defines which model entities are eligible for replacement, and finally a *placement policy* ([62], p. 1330). The migration of model entities has to be implemented efficiently and without invalidating the simulation results. Finally, the invocation frequency of the load-balancing scheme has to reflect the dynamism in the model. If a model’s interaction and load patterns are static, there is no need for load balancing. On the other hand, executing models with dynamic structures (e.g., based on the formalisms described in Section 6.2) almost inevitably leads to imbalanced load. The obstacles mentioned above usually lead to highly specific approaches, for example, tuned to a certain synchronization protocol [63]. Likewise, it is also possible to devise a load balancing algorithm that is tailored to DEVS. The following algorithm again makes use of the hierarchical nature of DEVS models and translates the actual repartitioning to the graph coloring problem [64].

6.3.2.5 A DEVS-Specific Load Balancing Algorithm

The basic idea behind the load balancing algorithm is to balance the load of *simultaneous* computations. A coupled PDEVs model might never be overloaded, as long as its submodels do not execute their state transition functions in parallel (i.e., if they do not process events at the same points in simulation time). To identify the submodels that are frequently processing events simultaneously, and should therefore be placed on different processors, a *parallelism matrix* can be used. It stores all pairwise parallel executions of submodels. This matrix can be interpreted as the adjacency matrix of a graph that represents the parallelism inherent in the given coupled PDEVs model: two vertices representing submodels are adjacent in this graph if and only if these have been executed simultaneously. This is illustrated in Figure 6.13.

For each load balancing step and each coupled model, a parallelism matrix is created. They are passed along the model tree in a bottom-up manner and eventually reach the root coordinator, where the graph coloring is done. This results in a set of relevant migration options, for which the *migration gain* is calculated. This can be enhanced to account for migration cost, for example, to only move some very large parts of the model across the entire network if this will still pay off. Finally, the root coordinator selects all beneficial migrations, pauses the simulation, triggers the migrations, and then lets the simulation proceed.



1

2

3

4

5

1

2

3

4

5

Corresponding graph

1

2

3

4

5

1

2

3

4

5

Coloring

FIGURE 6.13 Basic idea of the DEVS-specific load balancing algorithm: The parallelism of submodels is stored as a *parallelism matrix*.

6.3.3 EVENT QUEUES

The event set problem is a pretty well-known problem in the discrete-event simulation world. Sadly the available event list implementations are lacking essential methods for being efficiently usable in the DEVS context. Usually event queues only provide two methods: `dequeue` and `enqueue`. The first one retrieves the event with the minimal time stamp, the second one allows adding a new event to the queue.

If such an event queue shall be used for DEVS, we are faced with the problem that we might need to update the time stamp of an event—due to an external transition. If this is the case, we need the ability to search the old (internal) event of the model, remove it, and to insert an event with the new “tonic.” We called this the `requeue` operation.

If we want to execute PDEVS models, we need to execute *all* events having the minimal time stamp at once. Surely this can be achieved by successively calling the `dequeue` operation as long as we get events having the minimal time stamp. But it would be much easier to have a `dequeueAll` method, which returns all events having the minimal time stamp at once.

Having identified the need for these new methods we started to analyze existing event queue implementations and to modify them to include the new methods. Based on the calendar queue [65], we show the effect of these two methods on the overall execution performance, provided that they are well integrated into an event list.

6.3.3.1 Calendar Queue

The calendar queue is based on the idea of a calendar [65]. Events are collected in buckets (days), the number of buckets is called a year. If there are events for other years, they are added to the corresponding buckets of the current year. Basically, the `requeue` operation can be realized here in two different ways:

- Either we do a $O(n)$ search for the event stamp to be removed
- Or we keep an additional hash table to find the old time stamp and thus to be able to directly jump in $O(1) + O(\log n)$ to the entry to be removed

For our experiments, we realized both, the first one as the “CalendarQueue,” the second technique as “CalendarReQueue.”

The `dequeueAll` method is easy to realize, due to the fact that all events having an identical time stamp are stored in 1 day, and thus can be retrieved easily.

6.3.3.2 Results

Any comparison of algorithms is twofold. On the one side we need to analyze the theoretical boundaries of algorithms (see Table 6.1), and on the other side we need to execute meaningful experiments. In the literature, most studies with event list implementations are based on the “hold” test. This test is just an alternating execution of the `dequeue` and `enqueue` operations, which is surely fine for

TABLE 6.1
Upper Bound or Upper Bound/Expected Performance of the Calendar and Calendar Requeues Available as Plug-Ins for JAMES II

	Enqueue	Dequeue	DequeueAll	Requeue
CalendarQueue	$O(n)O(1)$	$O(n)O(1)$	$O(n)O(1)$	$O(n) + \text{enqueue}$
CalendarReQueue	$O(n)O(1)$	$O(n)O(1)$	$O(n)O(1)$	$O(1) + O(\log n) + \text{enqueue}$

the traditional event list interface. However, due to our new methods, we need an extended test [18,22]. In Listing 6.14 this new test is given. It has been created in accordance to the old “hold” test, but now uses the new methods, and it mimics the usage by a PDEVs simulator. Thus for each set of internal events having the minimal time stamp, a number of `requeue` operations is executed.

LISTING 6.14
Code of the PDEVs Event Queue Test Model—The Model Is Based on the “Hold Model”

```
1 //init empty queue
2 for i :=1 to size do
3 queue.enqueue (new Entry (random_initial_prio))
4 //remember current time
5 stopwatch.start;
6 //do count trials de-/enqueue ops
7 for i :=1 to trials do
8 elist :=queue.dequeueAll;
9 for all e in elist do
10 e.prio :=e.prio+random_prio_increment;
11 queue.enqueue (e);
12 end for
13 for j :=0 to rand(50) do
14 queue.requeue (rand(allEvents),
15 e.prio+random_prio_increment)
16 end for
17 stopwatch.stop;
18 return stopwatch.elapsedSeconds;
```

The results in Figures 6.14 and 6.15 show that the improved variant (the `CalendarReQueue`) outperforms the traditional one for most event time distributions. In these cases the speedup is around 2. However, there are two distributions for which this is not true—here the classical variant is better.

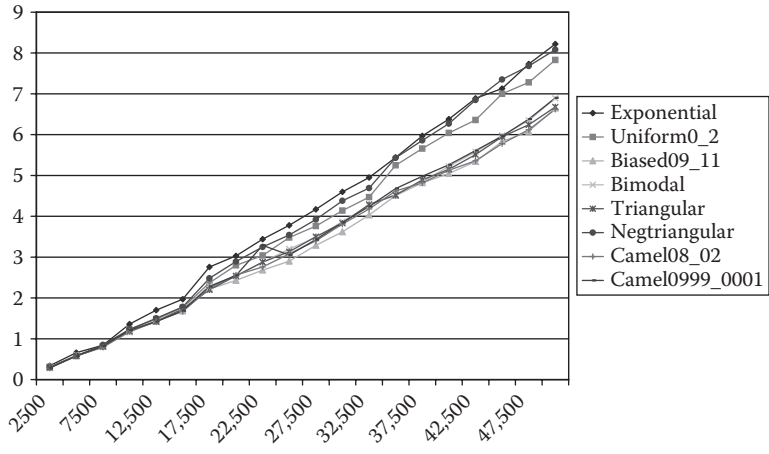


FIGURE 6.14 Runtime results of the calendar event queue (“PDEVs test”).

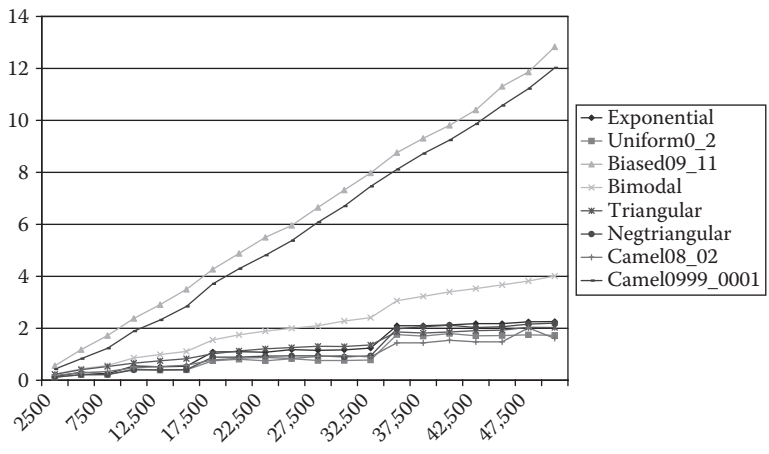


FIGURE 6.15 Runtime results of the calendar (re)event queue (“PDEVs test”).

6.3.4 SUMMARY

The alternative algorithms and data structures for DEVS underline the importance of developing these:

- Alternative implementations may speed up the simulation process of DEVS models significantly
- Alternative implementations may allow larger DEVS models to be computed at all
- There is no algorithm/data structure optimal for everything

6.4 SUMMARY AND OUTLOOK

DEVS is an interesting base for the development of specialized modeling formalisms that might ease the modeling of some systems—making DEVS useful for a plethora of applications. The strict separation of models and simulators facilitates the development of alternative algorithms, which might speed up the simulation of DEVS based models dramatically. If M&S technology is adapted to the needs of DEVS simulators, and if specialized algorithms are developed for supporting the simulation of DEVS models to a further extent, their execution can be done more efficiently. The development of methods based on ideas from the DEVS world, as well as the development of solutions for DEVS, is still not finished. The concepts behind DEVS already inspired or founded further developments, such as component-based modeling, an explicit notion of model validity (experimental frame), and so on. We realized all mentioned formalisms and algorithms as plug-ins for the framework JAMES II (<http://www.jamesii.org>; [66]). Such a set of alternatives provides a base for further research [67], here in regards to efficiency and validity of modeling and simulation based on DEVS.

ACKNOWLEDGMENT

The research documented in this paper was financially supported by the DFG (German Research Foundation).

REFERENCES

1. Zeigler, B. P. *Theory of Modeling and Simulation*. New York: Wiley Interscience, 1976.
2. Pawletta, T., and S. Pawletta. Object-oriented simulation of continuous systems with discrete structure changes. In *Proceedings of the European Multiconference on Modeling and Simulation (ESM'95)*, 627–30, 1995.
3. Barros, F. J. A formal representation of hybrid mobile components. *Simulation* 81, no. 5 (2005): 381–93.
4. Cho, Y., B. Zeigler, H. J. Cho, H. S. Sarjoughian, and S. Sen. Design considerations for distributed Real-Time DEVS. In *Proceedings of Artificial Intelligence and Simulation*, March, Tucson, AZ. New York: IEEE, 2000.
5. Gierke, M., J. Himmelsbach, M. Röhl, and A. Uhrmacher. Modeling and simulation of tests for agents. In *Proceedings of MATES*, Vol. 4196, 49–60. Berlin/Heidelberg: Springer, 2006.
6. Zeigler, B. P., and T. I. Ören. Multifaceted, multiparadigm modeling perspectives: Tools for the 90's. In *WSC '86: Proceedings of the 18th Conference on Winter Simulation*, 708–12. New York: ACM Press, 1986.
7. Uhrmacher, A. M., and R. Arnold. Distributing and maintaining knowledge—Agents in variable structure environments. In *Proceedings of the 5th Annual Conference on AI, Simulation, and Planning in High Autonomy System -Distributed Interactive Simulation Environments*, 178–84, Gainesville, FL. New York: IEEE Press, 1994.
8. Pawletta, T., B. Lampe, S. Pawletta, and W. Drewelow. A new approach for simulation of variable structure systems. In *Proceedings of the 41th KoREMA*, Vol. 2, 57–62, Opatija, Croatia, 1996.
9. Uhrmacher, A. M. Dynamic structures in modeling and simulation—A reflective approach. *ACM Transactions on Modeling and Simulation* 11, no. 2 (2001): 206–32.

10. Himmelspace, J., and A. M. Uhrmacher. Processing dynamic PDEVS models. In *Proceedings of the 12th IEEE International Symposium on MASCOTS*, 329–36. Edited by D. DeGroot and P. Harrison. October, Volendam, The Netherlands. New York: IEEE Computer Society, 2004b.
11. Uhrmacher, A. M., J. Himmelspace, M. Röhl, and R. Ewald. Introducing variable ports and multi-couplings for cell biological modeling in devs. In *Proceedings of the WSC 06*, 832–40, 2006.
12. Shang, H., and G. A. Wainer. A flexible dynamic structure devs algorithm towards real-time systems. In *SCSC: Proceedings of the 2007 Summer Computer Simulation Conference*, 339–45, San Diego, CA. Vista, CA: Society for Computer Simulation International, 2007.
13. Valentin, E. C., A. Verbraeck, and H. G. Sol. Advantages and disadvantages of building blocks in simulation studies: A laboratory experiment with simulation experts. In *Simulation in Industry, 15th European Simulation Symposium*. Edited by A. Verbraeck and V. Hlupic, 141–48, Delft, The Netherlands. Ghent, Belgium: SCS European Publishing House, 2003.
14. Kim, K., W. Kang, B. Sagong, and H. Seo. Efficient distributed simulation of hierarchical devs models: Transforming model structure into a non-hierarchical one. In *SS '00: Proceedings of the 33rd Annual Simulation Symposium*, 227, Washington, DC. New York: IEEE Computer Society, 2000.
15. Zeigler, B. P., H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. London: Academic Press, 2000.
16. Glinsky, E., and G. Wainer. New parallel simulation techniques of devs and cell-devs in cd + + . In *ANSS '06: Proceedings of the 39th Annual Symposium on Simulation*, 244–51, Washington, DC. New York: IEEE Computer Society, 2006.
17. Himmelspace, J., and A. M. Uhrmacher. Sequential processing of PDEVS models. In *Proceedings of the 3rd EMSS*. Edited by A. G. Bruzzone, A. Guasch, M. A. Piera, and J. Rozenblit, 239–44. October, Barcelona, Spain, 2006.
18. Himmelspace, J., R. Ewald, S. Leye, and A. M. Uhrmacher. Parallel and distributed simulation of Parallel DEVS models. In *Proceedings of the DEVS Integrated M&S Symposium*, 249–56, March 25–29, Norfolk, VA. Vista, CA: SCS, 2007.
19. Kim, K. H., T. G. Kim, and K. H. Park. Hierarchical partitioning algorithm for optimistic distributed simulation of devs models. *Journal of Systems Architecture* 44, nos. 6–7 (1998): 433–55.
20. Park, S., and B. P. Zeigler. Distributing simulation work based on component activity: A new approach to partitioning hierarchical devs models, In *Proceedings of the International Workshop on Challenges of large applications in distributed environments*, 124–32, IEEE, 2003.
21. Ewald, R., J. Himmelspace, and A. M. Uhrmacher. A non-fragmenting partitioning algorithm for hierarchical models. In *WSC '06: Proceedings of the 38th Conference on Winter Simulation*, 848–55. December 3–6, Monterey CA, 2006.
22. Himmelspace, J., and A. M. Uhrmacher. The event queue problem and PDEVS. In *Proceedings of the DEVS Integrated M&S Symposium*, 257–64, March 25–29, Norfolk, VA. Vista, CA: SCS, 2007b
23. Chow, A. C., and B. P. Zeigler. Parallel devs: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 1994 Winter Simulation Conference*. Edited by J. D. Tew and S. Manivannan, 716–22, San Diego, CA. Vista, CA: Society for Computer Simulation International, 1994.
24. Zeigler, B. P. *Multifaceted Modelling and Discrete Event Simulation*. New York: Academic Press, 1984.
25. Uhrmacher, A. M., and D. Weyns, eds. *Multi-Agent Systems: Simulation and Applications*. Boca Raton, FL: Taylor and Francis, 2009.

26. Kitano, H. Systems biology: A brief overview. *Science*, 295, no. 5560 (2002): 1662–64.
27. Barros, F. J. Modeling formalisms for dynamic structure systems. *ACM Trans. Model. Comput. Simul.* 7, no. 4 (1997): 501–15.
28. Kuttler, C., and A. Uhrmacher. Multi-level modeling in systems biology by discrete event approaches. *IT Themenheft Systems Biology*, 2006.
29. Kuttler, C., and J. Niehren. Gene regulation in the pi calculus: Simulating cooperativity at the lambda switch. *Transactions on Computational Systems Biology*, VII 4230 (2006): 24–55.
30. Himmelspach, J. *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations und Experimentiersystems-Entwicklung und Evaluation effizienter Simulationsalgorithmen*. Reihe Informatik. Göttingen, Germany: Sierke Verlag, December 2007.
31. Heylighen, F. Autonomy and Cognition as the Maintenance and Processing of Distinctions. In *Self-Steering and Cognition in Complex Systems*, 89–106. New York: Gordon and Breach, 1990.
32. Hu, X., B. P. Zeigler, and S. Mittal. Variable structure in devs component-based modeling and simulation. *Simulation* 81, no. 2 (2005): 91–102.
33. Uhrmacher, A. M., and C. Priami. Discrete event systems specification in systems biology—A discussion of stochastic pi calculus and devs. In *Proceedings of the Winter Simulation Conference 2005*, 317–26, Orlando, FL. New York: IEEE, 2005.
34. Milner, R. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge: Cambridge University Press, June 1999.
35. Uhrmacher, A. M., R. Ewald, M. John, C. Maus, M. Jeschke, and S. Biermann. Combining micro and macro-modeling in devs for computational biology. In *Proceedings of the Winter Simulation Conference*, 871–80, 2007.
36. Degenring, D., M. Röhl, and A. M. Uhrmacher. Discrete event, multi-level simulation of metabolite channeling. *BioSystems* 75, nos. 1–3 (2004): 29–41.
37. Salthe, S. *Evolving Hierarchical Systems*. New York: Columbia University Press, 1985.
38. Pidd, M., and R. Castro. Hierarchical modular modelling in discrete simulation. In *Proceedings of the 1998 Winter Simulation Conference*, Vol. 1, 383–9, Los Alamitos, CA. New York: IEEE Computer Society, 1998.
39. Henzinger, T. A. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, Washington, DC. New York: IEEE Computer Society, 1996.
40. Uhrmacher, A. M. Reasoning about changing structure, a modeling concept for ecological systems. *Applied Artificial Intelligence* 9, no. 2 (1995): 157–80.
41. Uhrmacher, A. M., M. Röhl, and J. Himmelspach. Unpaced and paced simulation for testing agents. In *Simulation in Industry, 15th European Simulation Symposium*, 71–80, Delft, The Netherlands. Ghent, Belgium: SCS-European Publishing House, 2003.
42. Pollack, M. E. Planning technology for intelligent cognitive orthotics. *6th International Conference on AI Planning and Scheduling*, 2002.
43. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York: ACM Press/Addison-Wesley Publishing Co., 2002.
44. Verbraeck, A. Component-based distributed simulations: The way forward? In *PADS '04: Proceedings of the Eighteenth Workshop on Parallel and Distributed Simulation*, 141–48. New York: ACM Press, 2004.
45. Janssen, T. M. V. Compositionality (with an appendix by B. Partee). In *Handbook of Logic and Language*. Edited by J. van Benthem and A. ter Meulen, 417–73. Amsterdam: Elsevier, 1997.

46. de Alfaro, L., and T. A. Henzinger. Interface-based design. In *Engineering Theories of Software-Intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*. Edited by M. Broy, J. Gruenbauer, D. Harel, and C. A. R. Hoare, 83–104. New York: Springer, 2005.
47. Elmqvist, H., S. E. Mattsson, and M. Otter. Object-oriented and hybrid modeling in modelica. *Journal Européen des Systèmes Automatisés* 35, no. 1 (2001): 1–10.
48. Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2007-7, EECS Department. Berkeley, CA: University of California, 2007.
49. Röhl, M. Platform independent specification of simulation model components. In *ECMS 2006*. Edited by SCS, 220–25, 2006.
50. Röhl, M., and A. M. Uhrmacher. Composing simulations from XML-specified model components. In *Proceedings of the Winter Simulation Conference 06*. New York: ACM, 2006.
51. Röhl, M., and A. M. Uhrmacher. Definition and analysis of composition structures for discrete-event models. In *Proceedings of the 2008 Winter Simulation Conference*, eds., S. Mason, R. Hill, L. Moench, and O. Rose, 942–50. *Winter Simulation Conference*, 2008.
52. Pierce, B. C. *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.
53. Tolk, A. What comes after the semantic web-pads implications for the dynamic web. Workshop on *Parallel and Distributed Simulation*, 55–62, 2006.
54. Himmelspach, J., and A. M. Uhrmacher. A component-based simulation layer for James. In *PADS '04: Proceedings of the Eighteenth Workshop on Parallel and Distributed Simulation*, 115–22, May, Kufstein, Austria. New York: IEEE Computer Society Press, 2004a.
55. Fjällström, P. O. Algorithms for graph partitioning: A Survey. *Linköping Electronic Articles Computer & Information Science* 3, no. 10 (1998).
56. Garey, M. R., D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *STOC '74: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 47–63. New York: ACM Press, 1974.
57. Kernighan, B. W., and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. Journal* 49 (1970): 291–307.
58. Lemeire, J., B. Smets, P. Cara, and E. Dirkx. Exploiting symmetry for partitioning models in parallel discrete event simulation, PADS '04: Proceedings of the eighteenth workshop on parallel and distributed simulation, IEEE, 189–94, 2004.
59. Abou-Rjeili, A., and G. Karypis. Multilevel algorithms for partitioning power-law graphs. *IPDPS*, 2006.
60. Karypis, G., and V. Kumar. *MeTis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices (Version 4.0)*, September 1998.
61. Park, S., A. C. Hunt, and B. P. Zeigler. Generic model partitioning with lookahead k: A multi-scale partitioning algorithm for devs biomimetic in silicon devices. In *2005 Agent-Directed Simulation Symposium (ADS'05)*, 2005.
62. Zhou, S. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering* 14, no. 9 (1988): 1327–41.
63. Carothers, C. D., and R. M. Fujimoto. Background execution of time warp programs. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, 12–9. Washington, DC: IEEE Computer Society Press, 1996.

64. Ewald, R., J. Himmelspace, A. Uhrmacher, D. Chen, and G. K. Theodoropoulos. A simulation approach to facilitate parallel and distributed discrete-event simulator development. In *Proceedings of the DS-RT 06*, 209–18, October 2–6, Torremolinos, Malaga, Spain. New York: IEEE Computer Society, 2006.
65. Brown, R. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Commun. ACM* 31, no. 10 (1988): 1220–7.
66. Himmelspace, J., and A. M. Uhrmacher. Plug'n simulate. In *Proceedings of the Spring Simulation Multiconference*, 137–43, March. New York: IEEE Computer Society, 2007a.
67. Himmelspace, J., R. Ewald, and A. M. Uhrmacher. A flexible and scalable experimentation layer for JAMES II. In *Proceedings of the Winter Simulation Conference*. Edited by S. Mason, R. Hill, L. Moench, and O. Rose, 827–35. Winter Simulation Conference, December, 2008.
68. Fillipi, J. B. Jdevs framework. <http://spe.univ-corse.fr/filippiweb/appli/index.html>. Accessed September 2008.

7 Object-Oriented Comodeling Methodology for Development of Domain Specific DEVS Models

Chang Ho Sung and Tag Gon Kim

CONTENTS

7.1	Introduction	178
7.2	M&S: Domain-Specific Complex Systems	179
7.3	Object-Oriented (OO) Modeling.....	181
7.3.1	DEVS Formalism.....	181
7.3.2	UML	183
7.3.2.1	Use Case Diagram	184
7.3.2.2	Class Diagram.....	184
7.3.2.3	Sequence Diagram	184
7.4	Layered Structure	185
7.5	Comodeling Methodology	187
7.5.1	Partition of OO Modeling.....	188
7.5.2	Implementation of Simulation Models	190
7.5.2.1	Dynamic Linked Library	190
7.5.2.2	Flexible Synthesis of Simulators.....	190
7.5.3	Comodeling Tools.....	191
7.5.4	Advantages.....	192
7.5.5	Example: Buffer.....	192
7.6	Case Study: A Simple Military War Game	193
7.6.1	Behavioral Level Model (BM).....	193
7.6.2	Discrete-Event Level Model (DEM)	194
7.6.3	Implementation	195
7.7	Conclusion	197
	References.....	197

7.1 INTRODUCTION

Discrete-event modeling can be considered as a process of abstract knowledge representation about a real-world system. As a model, the representation should be executable by a simulation environment to analyze the modeled system with respect to modeling objectives. The process may be based on the different worldviews for modelers such as event-oriented, process-oriented, object-oriented, and so on. Among them, the object-oriented (OO) approach may be most compatible with a real-world system from the system-theoretic viewpoint of model representation [1].

The system-theoretic representation first specifies a system as a set of inputs, a set of outputs, and a set of states. It then defines a set of operations on the representation. The Discrete Event System Specification (DEVS) formalism, which represents a discrete-event system (DES) from the system-theoretic viewpoint, is known to be compatible with the OO worldview. Moreover, the formalism supports hierarchical, modular specification of discrete-event models, which allows us to assemble previously developed component models in a flexible manner [2].

Given modeling objectives, modeling and simulation (M&S) engineers do not need to model the whole system. They only need to represent good enough abstract knowledge to meet the objectives. To perform this limited knowledge representation, M&S engineers should have thorough knowledge about the modeled system. However, expecting the M&S engineers to have such knowledge is difficult, especially for domain-specific modeling such as war games. Often in practice, the domain engineers define functional requirements as modeling objectives, and the M&S engineers transform the requirements into a simulation model. Thus, the domain engineers design models as the detailed Behavior-Level Models (BM) considering the functions as the behaviors. On the other hand, the M&S engineers design models such as the Discrete-Event Level Models (DEM) satisfying the functional requirements. Furthermore, platform engineers [3] implement the models and compose the real systems. Therefore, domain specific system modeling requires cooperative teamwork among a group of experts; domain engineers, M&S engineers, and platform engineers throughout the whole modeling process. This chapter describes a cooperative framework supporting the teamwork in developing domain-specific DEVS models.

The methodology employs a layered approach in the model development. The upper layer represents an object's abstract behavior, and the lower layer represents the object details. The upper layer is a DES layer that represents a DEM; and the lower layer is an object model (OM) layer that represents a BM. The modeling at the DES layer and the OM layer is mainly the responsibility of M&S engineers and domain engineers, respectively. The model developments are a comodeling process where the M&S and the domain engineers collaborate concurrently at the two different abstraction layers while modeling the same objects.

There have been some efforts to support communication and cooperation between domain engineers and M&S engineers using Unified Modeling Language (UML) [4]. However, the approach is not a concurrent process that would allow the

domain and the M&S engineers to work at the same time. Instead, the approach allows domain engineers to first analyze and develop the system specifications using UML. Next, M&S engineers transform the diagrams into DEVS models with additional information necessary for the discrete-event simulation. Thus, the modeling process in [4] seems to be sequential while the approach described in this chapter supports development work to be a parallel process between the domain and the M&S engineers.

This chapter is organized as follows. Section 7.2 describes modeling and the simulation of domain-specific complex systems and the difficulty of developing those systems. Section 7.3 explains OO modeling and reviews the DEVS formalism and UML as background information. Section 7.4 proposes our comodeling methodology in a layered structure form, which is explained in detail in Section 7.5. In Section 7.6, an example of a war game model is developed within the proposed comodeling methodology. Conclusions are drawn in Section 7.7.

7.2 M&S: DOMAIN-SPECIFIC COMPLEX SYSTEMS

Modeling and simulation of the domain-specific systems require much domain knowledge. Figure 7.1 shows the difficulty of developing models for the domain-specific complex systems. Assume an advanced software engineer has a lot of knowledge of software (S/W) technologies such as programming and the general software development process. When a software engineer develops a weather forecast model, he needs information for a weather system from the National Oceanic and Atmospheric Administration (NOAA). Although he has data details, he cannot develop the model without domain-specific knowledge. In fact, he will find it difficult to understand the information, because he is not an expert about weather phenomena. Therefore, a helper is necessary for the software engineer, and the helper is a professional person who has much knowledge of weather systems.

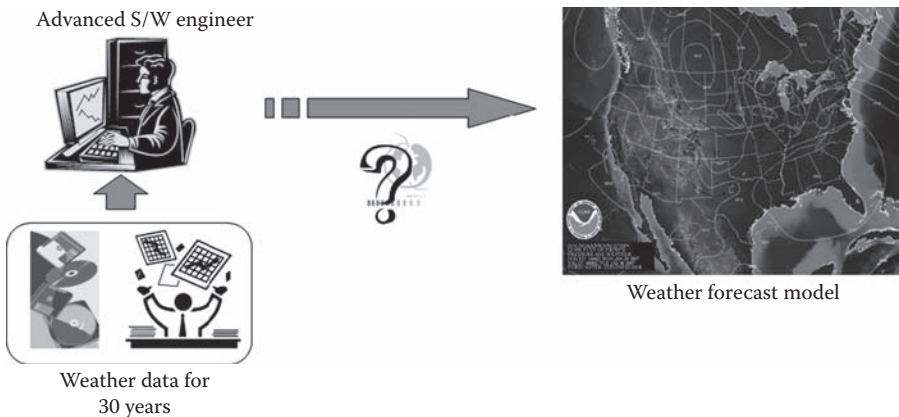


FIGURE 7.1 Difficulty of the development of models for domain-specific complex systems.

In spite of the domain knowledge support, the software engineer may still fail to develop the model because of a lack of M&S skills for developing domain specific models. The simulation model is defined by a set of instructions, rules, equations, and constraints for generating output (I/O) behaviors from inputs. Thus, a modeler has to specify the relationship of the I/O and the dynamics. No matter how much the software engineer may have learned from meteorology, he cannot develop the model without certain M&S theory. Hence, M&S theory experts are needed as well.

It would be difficult for the software engineer to develop domain specific models solely using his S/W knowledge, as has been noted above. Figure 7.2 shows additional information for developing models. Eventually, a domain-specific model, such as a weather forecast model, is developed with the integration of domain knowledge and M&S methodology using S/W technology such as programming skills.

As we have seen, the cooperation among experts of the various fields is essential to M&S of the domain-specific complex systems. Figure 7.3 shows the role of

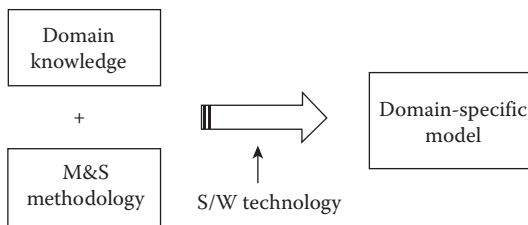


FIGURE 7.2 Additional information for developing domain specific models.

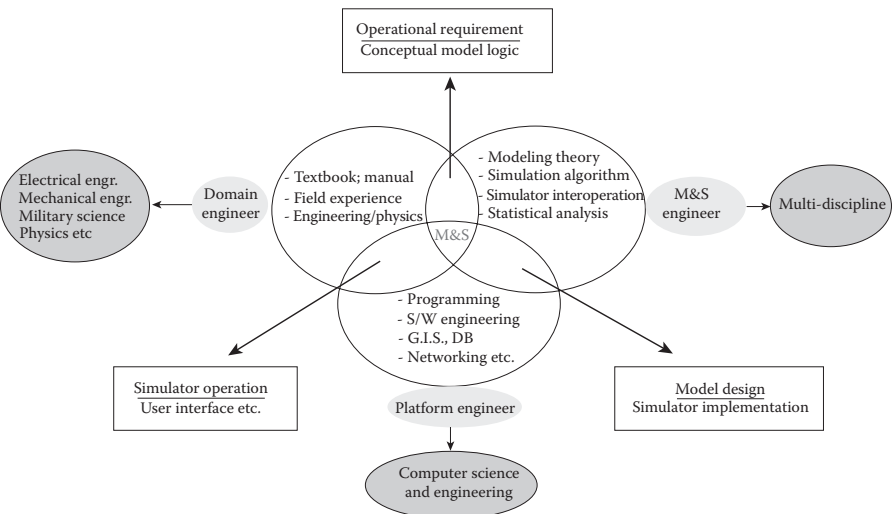


FIGURE 7.3 Experts cooperation for development of domain specific simulation models.

each expert in developing domain specific simulation models. Modeling and simulation of domain-specific systems requires integration of each expert's professional knowledge. Domain requirements consist of textbooks, field experience, physics, and data such as the weather for 30 days. They are the basic units to understanding the dynamics of the systems and used by domain engineers majoring in electrical engineering, military science, physics, and so on. The material contains operational requirements of a simulator, and it is utilized in modeling conceptual logic and architecture as well as providing algorithms detailing operations of a system. M&S engineers observe the domain objects with a view that allows them to analyze the high level aspects of the objects. On the other hand, domain engineers design detailed algorithms of physical phenomena. Furthermore, the model design is used to implement a simulator by platform engineers who use S/W technologies such as several programming languages and tools. In general, domain engineers are persons who analyze the domain requirements and are also end-users to execute the simulator. Therefore, platform engineers have to implement a simulator in cooperation with domain engineers for applying a demand from end-users.

7.3 OBJECT-ORIENTED (OO) MODELING

The OO modeling originated from the discrete-event modeling and simulation language SIMULA 67. However, OO modeling of general software does not need to represent an object at a DES level, which would require additional semantics. The UML is one such modeling methodology that is widely used in the software engineering community [5]. On the other hand, the DEVS formalism supports precise semantics to specify discrete-event models with an OO viewpoint, and DEVS is widely used in discrete-event M&S society. As shown in Figure 7.4, both DEVS and UML are object-oriented modeling frameworks that are descendents from the discrete-event simulation language, SIMULA 67. However, the main difference between the two methods is that UML may not be based on formal semantics of the specification of an object as a discrete-event model. More specifically, UML does not support specification of time advance for discrete-event simulation. Thus, as shown in Figure 7.4, some additional information is required to specify an object completely as a discrete-event model when using UML. On the other hand, any operation on the object can be specified by UML.

7.3.1 DEVS FORMALISM

The DEVS formalism specifies discrete-event models in a hierarchical and modular form [2]. With this formalism, one can perform modeling more easily by decomposing a large system into smaller component models with coupling specifications between them. There are two kinds of models: atomic models and coupled models.

An atomic model is the basic model and has specifications for the dynamics of the model. Formally, a seven-tuple specifies an atomic model M as follows.

$$M = \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, \text{ta} \rangle,$$

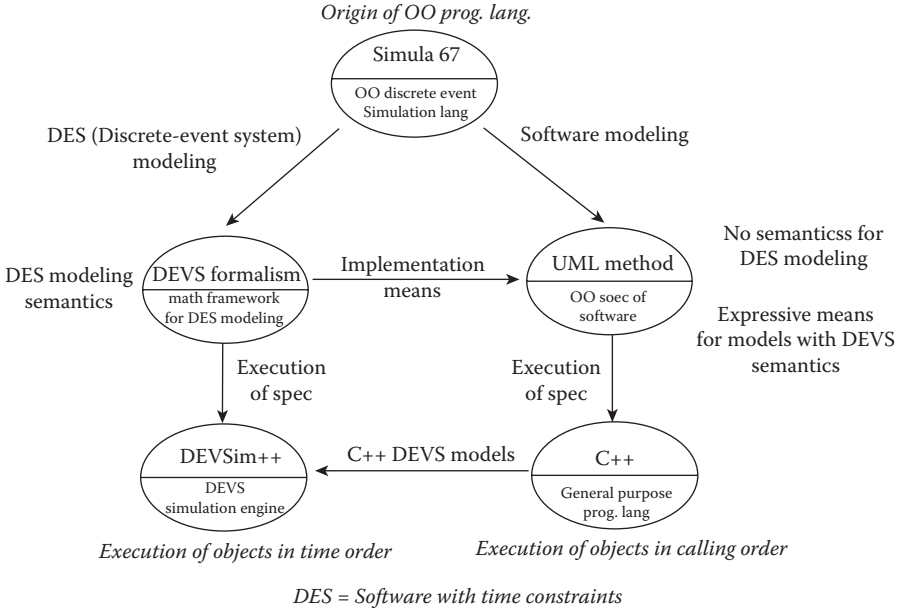


FIGURE 7.4 The OO modeling: DES (DEVS) vs. software (UML).

where

X : a set of input events

Y : a set of output events

S : a set of sequential states

$\delta_{\text{ext}}: Q \times X \rightarrow S$, an external transition function

where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set of M

$\delta_{\text{int}}: S \rightarrow S$, an internal transition function

$\lambda: S \rightarrow Y$, an output function

$ta: S \rightarrow \mathbb{R}^+_{0, \infty}$ (nonnegative real number), time advance function.

A coupled model provides the method of assembly of several atomic and/or coupled models to hierarchically build complex systems. Formally, a coupled model is defined as follows:

$$DN = \langle X, Y, M, EIC, EOC, IC, SELECT \rangle,$$

where

X : a set of input events

Y : a set of output events

M : a set of all component models

$EIC \subseteq DN.X \times \cup M.X$: external input coupling

$EOC \subseteq \cup M.Y \times DN.Y$: external output coupling

$IC \subseteq \cup M.Y \times \cup M.X$: internal coupling

$SELECT: 2^M - \emptyset \rightarrow M$: tie-breaking selector.

An overall system consists of a set of component models, either atomic or coupled, thus comprising a hierarchical structure. Each DEVS model, either atomic or coupled model, corresponds to an object in a real-world system that is modeled. Within the DEVS framework, model design may be performed in a top-down fashion while model implementation may be performed in a bottom-up manner.

A DEVS model can be simulated by communicating hierarchical abstract simulators. The architecture of these simulators is the same as the DEVS model architecture. The abstract simulators are a set of distributed simulation algorithms, which can be implemented in a sequential, as well as in a distributed, computing environment [6]. Since DEVS models are developed in an object-oriented manner, it is natural that an environment for DEVS M&S is implemented in an object-oriented programming language such as C++ or Java. In fact, the first such C++ implementation is DEVSim++ [7] in which modeling facilities and abstract simulators are explicitly separated. In the DEVSim++ environment, modeling facilities are open to modelers but abstract simulators are not accessible externally. The facilities allow modelers to create DEVS models of type atomic or coupled, which are subclasses of the class atomic or coupled model defined within DEVSim++.

Within the DEVSim++ environment, modelers can develop DEVS models using the interface of a model, which is a set of APIs to specify the DEVS models in DEVS semantics. Thus, APIs for specification of the DEVS models are defined such that there is a one-to-one correspondence between APIs and functions in the formalism. For example, APIs for a specification of four characteristic functions in an atomic DEVS are *TimeAdvanceFn()*, *ExtTransFn(Message &e)*, *IntTransFn()*, and *OutputFn(Message &)*. APIs for specifying coupled DEVS models are similarly defined. Once DEVS models are developed by using modeling facilities and APIs, simulation of such models can be performed by abstract simulators embedded in DEVSim++.

Different implementations for DEVS M&S are available in public domains and an effort for standardization of DEVS modeling/simulation environments is ongoing by the DEVS Standardization Organization [8].

7.3.2 UML

The Unified Modeling Language (UML) is a standardized language for specifying, visualizing, and documenting the artifacts of an object-oriented system under development [9,10]. Note that UML is not a kind of modeling methodology, and it just provides a visual syntax that we can use to construct models. It simplifies the complex process of software design and presents the various views of a system as shown in Figure 7.5, making a blueprint for construction [11]. Objects contain information and may perform functions, and UML models have static structure and dynamic behavior. The former relates to the type of objects and relationship among them and the latter denotes functions of the systems. This section describes only three diagrams among the several diagrams of UML that are shown in Figure 7.5 and are used for modeling an object. The 4 + 1 view is a way of looking at the overall architecture of a system, and UML provides excellent support for each of the 4 + 1 views. The diagrams, which are used in this chapter,

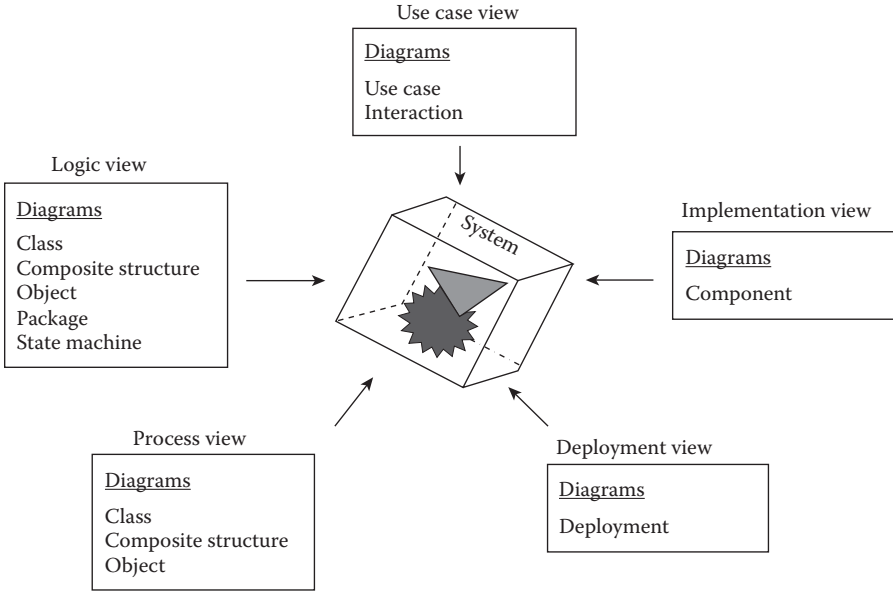


FIGURE 7.5 The 4 + 1 view of a system.

are the use case diagram, class diagram, and sequence diagram that is a subdiagram of interaction diagram.

7.3.2.1 Use Case Diagram

A use case diagram is a behavior diagram that defines a set of use cases that comprise actors and relationships between them. As a behavioral classifier, the diagram defines a sequence of actions, performed by one or more actors and a system, which results in an observable value to one or more actors. For system developers, this is a technique to gather system requirements from a user's point of view.

7.3.2.2 Class Diagram

A class diagram is a structure diagram that shows a set of classes, interfaces, and/or collaborations and the relationships among these elements. A class includes name, attributes, and operations. This diagram is a central modeling technique that is a part of nearly all object-oriented methods and that represents the static part of a system.

7.3.2.3 Sequence Diagram

A sequence diagram is an interaction diagram that focuses on the temporal ordering of messages between objects. A sequence diagram depicts a sequence of actions that occur in a system, which is a very useful tool for easily representing the dynamic behavior of the system. This diagram includes objects and messages in two-dimensional (2-D) form. On the horizontal axis, it shows the life of objects that it represents, while on the vertical axis, it shows the sequence of the creation or invocation of these objects.

7.4 LAYERED STRUCTURE

The main purpose of the OO comodeling methodology is to partition specifications of a domain specific DEVS model into two expert groups: domain engineers and M&S engineers. To do so, we view the DEVS framework as a superset of the UML methodology in its modeling power. In other words, a DEVS model can represent all information that a UML model can specify, but the other way around may not be possible. Assuming that domain engineers and M&S engineers are different, we propose a layered approach to OO discrete-event modeling. The approach represents a discrete-event model by two layers: a DES model layer and an OM layer [12]. More specifically, the OM layer specifies details of object operations on variables of an object; the DES layer specifies abstract operations on states of the same object, which determine simulation time advance and output events generation. In fact, the layered framework establishes a client–server architecture in that the abstract operations of an object exploit the detailed operations of the same object in the overall model representation. Figure 7.6 shows the layered structure for domain specific DEVS model development and simulation.

As shown in Figure 7.6 domain knowledge for the system to be modeled is represented by mathematical equations, rules, and even verbal descriptions, which are available in textbooks, the field, and/or technical military manuals.

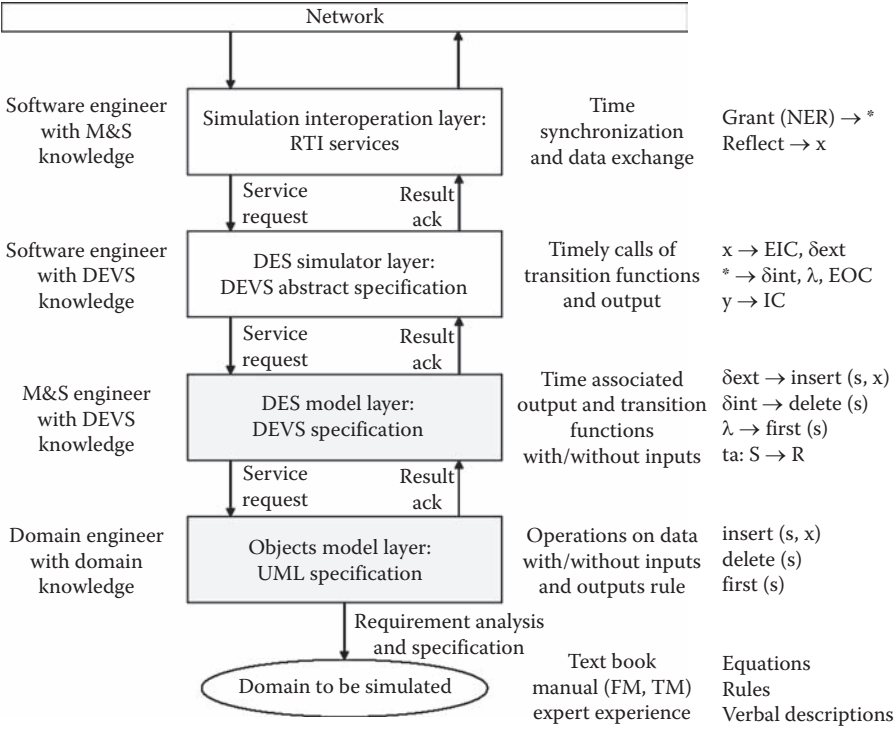


FIGURE 7.6 Layered structure of OO DES modeling.

Such knowledge can even be obtained by an engineer's experience in domain fields. Extraction of appropriate equations and rules from the domain knowledge, which meet the modeling objectives, can be done by domain engineers. To be successful they should clearly identify modeling objectives. This is not the case for the modeling of hardware or processes for which M&S engineers without domain knowledge can understand. Examples of such modeling include performance modeling of computer/communication systems and of manufacturing systems. Technically, extracted information should be specified as objects and their relationship at the OM layer. UML is employed at this phase of specification. More specifically, the simulator architecture should be identified in the form of a collection of objects and relationship between objects. Each object should be represented by inputs, outputs, variables, and their operations. For example, an object QUEUE may have operations such as *insert(s, x)*, *delete(s)*, and *first(s)* where *s* is state and *x* is an input event. However, note that such operations are not complete enough to represent QUEUE as a discrete-event model.

A discrete-event model for QUEUE can be represented by the DEVS formalism. The formalism has four characteristic functions for discrete-event modeling: external transition, internal transition, output, and the time advance function. Consider the external transition function of the QUEUE model, which specifies the update rule of state *s* when an input is received. Intuitively, the function is the same as *insert(s, x)* defined in the OM layer. Likewise, the internal transition function is the same as *delete(s)* and the output function is the same as *first(s)*. Thus, the QUEUE model at the DES layer using the DEVS formalism can directly use services of the QUEUE model provided by the OM layer. To be complete, however, the QUEUE model at the DES layer should have time advance information for discrete-event simulation, which can be provided by the time advance function of the DEVS formalism.

On top of the previous layers, there exist a DES simulator layer and a simulation interoperation layer. The DES simulator layer defines simulation algorithms for underlying DES models. The simulation algorithms are clearly separated from DES models. This separation of layers reduces development cost of a complete simulator because the verified simulation algorithms are invariant and, therefore, reusable irrespective of the underlying models.

The simulation interoperation layer is required only when the simulator should be part of a distributed simulation. A stand-alone simulator that consists of the lower three layers quickly turns out to be interoperable when simply adding the simulation interoperation layer. This layer consists of two parts: High Level Architecture (HLA)/Run-Time Infrastructure (RTI) for external interoperation, and internal protocol conversion for underlying the DES simulation layer. The HLA/RTI is good for interoperation with other simulators. The HLA/RTI, however, does not suffice for this layer because it does not contain any specification for simulation algorithms. The layer requires a specific protocol conversion from HLA/RTI to the underlying DES simulator layer. The DES simulation layer and simulation interoperation layer are beyond the scope of this chapter because it does not affect the cooperation between domain engineers and M&S engineers.

The layered approach makes it possible for domain engineers and M&S engineers to work concurrently in a cooperative manner. This is a major advantage to

the approach in OO model development. More specifically, domain engineers focus solely on UML modeling while M&S engineers are only concerned with DEVS modeling. Of course, the process from UML to DEVS modeling may not always be sequential unless all the operations required in DEVS modeling are not yet available in UML models. In such a case, M&S engineers may request domain engineers to define operations that M&S engineers need. However, we believe that good cooperation between domain engineers and M&S engineers from the onset of the system design may minimize such a backward process or may even eliminate it.

7.5 COMODELING METHODOLOGY

The hardware (HW)/software (SW) codesign represents the satisfaction of system-level objectives by exploiting the trade-offs between hardware and software in a system through their concurrent design. To do so, the system specification is partitioned in hardware and software parts for concurrent processing. After that, each part is implemented and then integrated for cosimulation [13]. The comodeling methodology is analogous to HW/SW codesign. The methodology is a development process of simulation models by separated groups of experts in cooperation. The groups consist of three experts: domain engineers, M&S engineers, and platform engineers as we have seen in the previous section.

- Domain engineers perform the domain analysis. Based on the domain requirements, the domain engineers generate the domain model, which describes the elements in the domain and the relationships between them. Domain models include mathematical equations, algorithms, and strategies.
- M&S engineers are in charge of modeling and simulation of DES from domain requirements. They are especially interested in the high-level abstract behavior of objects.
- Platform engineers are general program developers who implement simulation models using model specifications. They also take charge of construction of the simulation environment.

The development of domain-specific complex systems is a difficult task, which requires a high degree of expert knowledge from the involved engineers. The comodeling methodology is similar to the general software development process. Among the complex process, analysis of simulator requirements and specification is in the first place. Domain engineers play an important role in the analysis, because the requirements are emerging from domain knowledge as well as the domain engineers and they are the actual users of the developed simulator. Simulator requirements include process, model, environment, and numerical requirements [14]. Domain engineers analyze the information such as documentation, kind of experiments, observations, structural dynamics, and physical phenomena. After that, domain engineers, M&S engineers, and platform engineers design the simulator architecture together from requirements and specification for a system to be simulated. Then, the designed model is partitioned into modeling levels. If M&S engineers do not

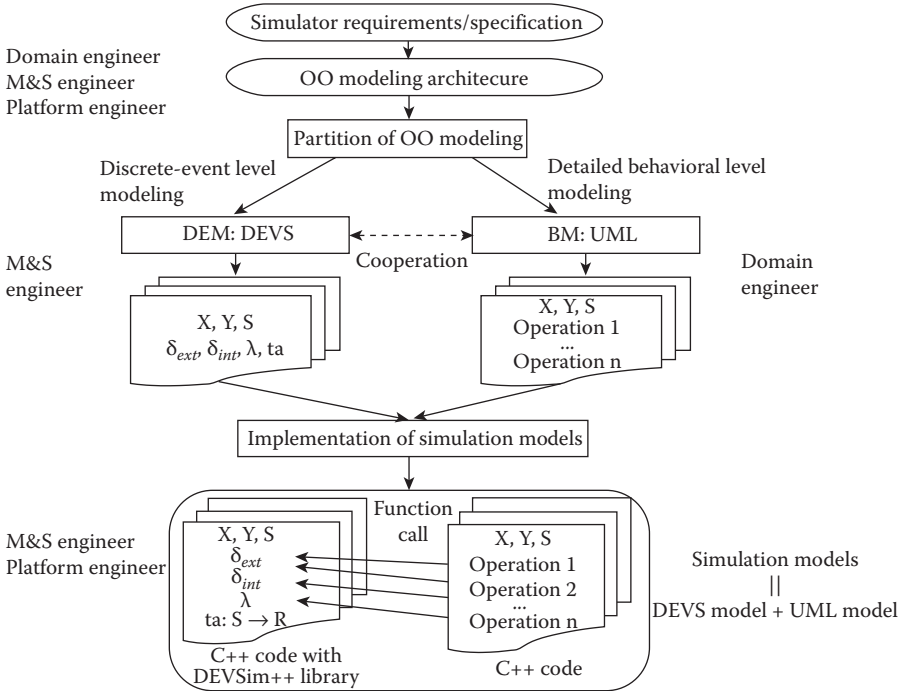


FIGURE 7.7 Procedure of comodeling methodology.

need detailed knowledge on an object in modeling atomic DEVS, they define inputs, outputs, and state sets, and they specify four functions of the DEVS formalism: *ExtTransFn* (δ_{ext}), *IntTransFn* (δ_{int}), *OutputFn* (λ) and *TimeAdvanceFn* (ta). On the other hand, if atomic DEVS modeling needs detailed knowledge about the object, domain engineers define such knowledge in the form of operations on the object using UML. Then, M&S engineers employ the operations as services. Specifications of both DEVS and UML models are implemented by C++ with the DEVSim++ library and C++, respectively, and the specific interface (e.g., a method call) couples DEVS models to UML models. Figure 7.7 shows the general procedure of comodeling methodology.

7.5.1 PARTITION OF OO MODELING

The model to be developed is decomposed into submodels in a top-down manner. The architectural design is done from an OO modeling point of view. The OO modeling approach for system modeling views a system as an object in which its representation and associated operations are explicitly defined. A designed model presents an object and is partitioned with two modeling levels of an object; DEM and detailed BM. A model describes the dynamic behavior of an object, and the behavior includes event sequences of a model, state transition, and operations processed by specific events and state transitions. Among them, state transitions by input and output events

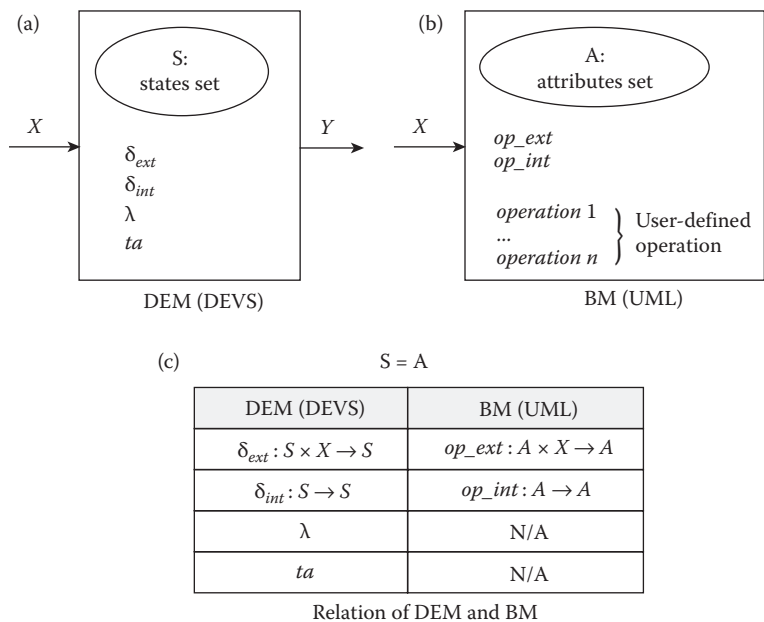


FIGURE 7.8 DEM and BM.

are described by M&S engineers using the DEVS formalism, and detailed operations at specific states are described by domain engineers using the OO modeling language such as UML.

Normal forms of DEM and BM are shown in Figure 7.8. As shown in Figure 7.8a and b, a DEVS model has three sets and four functions and a UML class has attributes and operations. In Figure 7.8b, op_{ext} means an operation of the UML class and this plays a role in coupling between DEM and BM. Other operations are user-defined operations that are employed only in the UML class. Figure 7.8c shows the relation between DEM and BM, which enables their integration for designing complex models. Because both a DEM and a BM represent the same model, the set of states of DEVS is the same as the attributes set of UML. To facilitate their integration, a mapping table is required, which captures the relations between the DEVS functions and UML class operations. In Figure 7.8c, $ExtTransFn$ (δ_{ext}) of DEVS is related to op_{ext} of UML. It means that the operations of UML, which describe detailed behavioral algorithms, are used when an external transition occurs. Likewise, $IntTransFn$ (δ_{int}) of DEVS are related to operations of UML. Other functions of DEVS, $OutputFn$ (λ) and $TimeAdvanceFn$ (ta), are not mapped to operations of UML, because they are not related to the behavior of objects.

Generally, DEM is a high-level model specification and BM is a low-level model specification. The overall simulation model is the integration of high-level and low-level model specifications. Note that the cooperation between two experts is essential because DEM uses the operations of BM.

7.5.2 IMPLEMENTATION OF SIMULATION MODELS

In the comodeling methodology, M&S engineers specify DEVS models that represent system behaviors at the state-transition level. Domain engineers are responsible for developing specific algorithms as individual functions. These algorithms typically require domain-specific knowledge. Then, each engineer implements the simulation model of each specification with platform engineers. Similarly, the model implementation is separated between DEM and BM. Each model is coupled by abstract methods and the detailed algorithm is implemented in BM. The separation of implementation makes it easy to test and maintain simulation models. In general, modification of simulation requirements brings about more frequent change of detailed algorithms than the change of abstract behavior. When a variation of rules or scenarios alters the specific algorithms, M&S engineers do not have to support the modification if the change only affects the behavioral operations of objects. Technically, the switch of functions in the application at run-time or load-time becomes possible because of the dynamically linked library technology, which is described in the next section.

7.5.2.1 Dynamic Linked Library

In computer science, a library is a collection of subprograms used to develop software [15]. A library contains code and data, which provide services to programs linking the library. Executable program and libraries make references known as *links* to each other through the process known as *linking*, which is typically done by a *linker*. The code and data in a library are copied into a new executable at link time, which is called *static* linking.

Dynamic linking means that the linker only records what libraries the executable needs and the index names or numbers, and the data in the library remains in a separate file on disk. The copying of code and data in the library into the memory space of the process is done when the application is loaded or during the execution of the process. This type of library is called a dynamically linked library. The term, Dynamic Link Library (DLL), specifically indicates dynamic libraries in Microsoft Windows environments.

Dynamic loading means that a dynamically linked library loads and unloads at run-time on request. The request to load such a dynamically linked library may be made implicitly at compile-time, or explicitly by the application at run-time.

Traditional static linking binds a program and libraries together at compile-time, and produces one executable code that includes all data and code. In this technology, the program should be recompiled in order to replace a function. However, a dynamically linked library that contains functions to be replaced, is loaded after the application starts. If we implement functions as a separate dynamically linked library each, the simulator is able to execute user-supplied functions in the library without recompiling the simulator program.

7.5.2.2 Flexible Synthesis of Simulators

The majority of cowork between M&S engineers and domain engineers in comodeling is to specify function prototypes. The functions should be designed independently

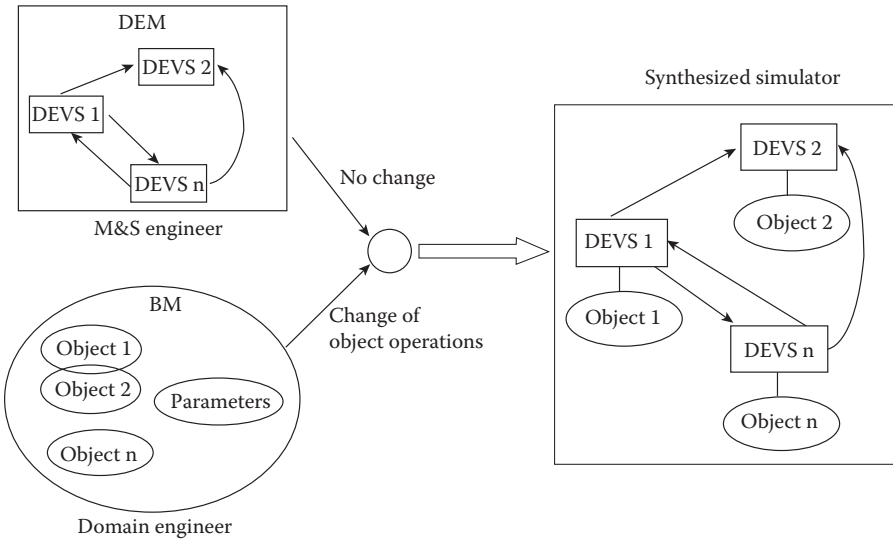


FIGURE 7.9 Flexible simulator synthesis in comodeling.

and as self-contained as possible. DEVS models call associated functions inside each state transition functions. The contents of the function do not affect the model behavior as long as the I/O types of the function are consistent [16].

Domain engineers are able to supply a set of behavior functions in the form of dynamically linked libraries. Combined with implemented DEVS models, a complete simulator is synthesized by selecting supplied dynamically linked libraries. This selection can be done at run-time. Therefore, domain engineers easily compare a different combination of algorithms without changes to the DEM. These procedures are described in Figure 7.9. DEM is a coupled model as a composition of atomic models and developed by M&S engineers using DEVSim++ library. BM is a set of object classes defining numerous attributes and operations, and it means detailed algorithms of objects. Domain engineers implement the BM using C++ in the form of a dynamic linked library. DEM and BM is synthesized at run-time, and after that, a DEVS atomic model is linked to an object class. Although modification of simulator requirements affects the change of contents of object operations, the modified object class is easily linked to a DEVS atomic model. In addition, different simulation results can be achieved under varying operations of objects without recompiling the simulator though the flexible simulator synthesis.

7.5.3 COMODELING TOOLS

There are various tools that specify and implement domain-specific models. For separated modeling, an OO specification for both high and low-level modeling and well-defined semantics to represent both levels are required. Both specifications should be understandable specifications to each other's groups to allow cooperation.

A typical example of formalism for high-level modeling is DEVS and the example for low-level modeling is UML. There are two methods to implement models. As in this chapter, only one OO programming language for both implementation and simulation is employed by calling methods of low-level models. DEVS models are implemented using C++ language with the DEVSim++ environment and UML models use the same language. The other implementation method is to use a different OO programming language for each. Those simulations are complicated because of the difficulty defining interfaces between those programming languages, but HLA [17–19] enables the simulation among heterogeneous simulators. This chapter does not deal with the HLA.

7.5.4 ADVANTAGES

Within the methodology, M&S engineers and domain engineer design simulation models in a closely coupled manner. The methodology should identify modeling levels of objects. Modeling at the DEM level does not require specific knowledge of objects, which only the domain engineers understand. However, if M&S engineers could not understand detailed operations of an object, then the operations should be provided by domain engineers. In this sense, DEVS modeling is viewed as software and UML modeling is viewed as hardware in the HW/SW codesign methodology. Thus, the main advantage of the proposed methodology is a concurrent process in the development of models. For that reason, the methodology lets us save time and cost for the development of complex models, so the experts can perform their jobs effectively and efficiently.

7.5.5 EXAMPLE: BUFFER

Let us present an example to illustrate the comodeling methodology. There is a buffer model, and the buffer queue receives inputs and delivers an output to the processors as depicted in Figure 7.10. We can model a buffer with inputs, outputs, states, and abstracted operations such as *object-insert* and *object-delete*. Inputs and outputs denote events, and the length of the buffer denotes a state of the buffer model. The discrete-event model of the buffer handles received inputs with the *object-insert* operation in the external transition function of DEVS, and sends an output to the processor with the *object-delete* operation in the internal transition function after sending time defined in the time advance function of DEVS. The buffer model works independent of the detailed algorithms that implement the *object-insert* and *object-delete* operations.

Implementation of these *object-insert* and *object-delete* operations requires specific knowledge about queuing algorithms such as FIFO, LIFO, priority queue, and the operations are invoked by calling methods within a programming code. In the comodeling methodology, M&S engineers deal with the discrete-event model of a buffer, and domain engineers implement detailed operations of a buffer.

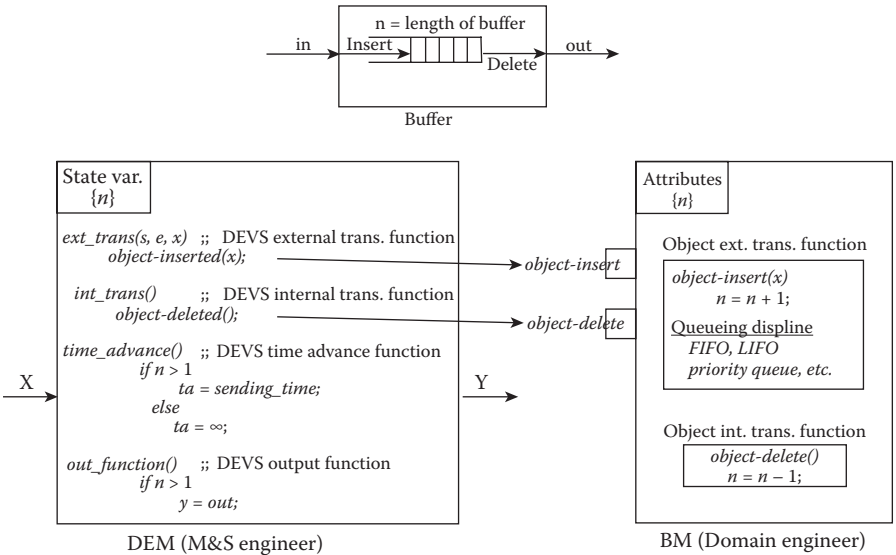


FIGURE 7.10 Example: Comodeling of buffer.

7.6 CASE STUDY: A SIMPLE MILITARY WAR GAME

Let us illustrate a simple military war game model. Assuming that there are many ships in a naval war game model and that the ships are moving on the sea, the positions are ordered by a gamer. The scenario of the war game simulation is as follows:

1. A gamer gives a command to move a ship to some point as a destination.
2. The ship calculates the next position per interval time until it arrives at the destination point.

If the M&S engineers do not know the operational rules for a ship to move, they could not specify a discrete-event simulation model for the ship. In such a case, the rules should be specified by naval domain engineers, and M&S engineers can use the rules for DEM. Let us explain the model specifications of the ship object at the discrete-event level and behavioral level.

7.6.1 BEHAVIORAL LEVEL MODEL (BM)

At the BM level, a ship object can be modeled by attributes and operations defined on the object. The modeling at that level is to define diagrams of UML in OO software development.

To capture the maneuver of a ship, the ship has velocity, heading, and position as its attributes. Similarly, operations on the ship include *move*, the definition of which needs additional operations. Those operations are to calculate heading, velocity, and

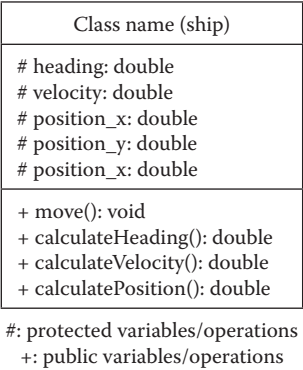


FIGURE 7.11 Class diagram of the system.

next position. Thus, at this BM level domain engineers should offer algorithms for the operations. Figure 7.11 represents a class diagram of the ship object at the BM level in relation to maneuver.

7.6.2 DISCRETE-EVENT LEVEL MODEL (DEM)

At the DEM level, a ship object should be specified in the form of timed state transitions. In other words, the object has a specification of a state transition with an associated occurrence time. M&S engineers specify such information of the ship object using the DEVS formalism as follows:

$$\begin{aligned} M &= \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle \\ X &= \{ \text{“move\_order”} \} \\ Y &= \{ \} \\ S &= \{ WAIT, MOVE \} \\ \delta_{ext} &: WAIT \times \text{“move\_order”} \rightarrow MOVE \\ \delta_{int} &: MOVE \rightarrow MOVE \\ ta(WAIT) &= \infty \\ ta(MOVE) &= t1 \text{ (interval time for move operation)} \end{aligned}$$

As shown in the specification, a ship is waiting for an input command of *move_order* at the initial state of *WAIT*. Time advance at that state is defined as infinite in the DEVS formalism, meaning that nothing can happen at that state unless an input event occurs.

When the state of the ship is *WAIT* and an input event of *move_order* occurs, the ship transitions to the *MOVE* state and the ship begins moving toward a destination point. In this state, the time advance is some finite time because the position of the ship changes at each unit time. In the *MOVE* state, a sequence of operations is calculated in order to compute the next position of the ship after a unit time. Those operations may be defined in the BM level. Figure 7.12 shows the ship object model at both the DEM level and the BM level. Note in Figure 7.12 that the state transition in the

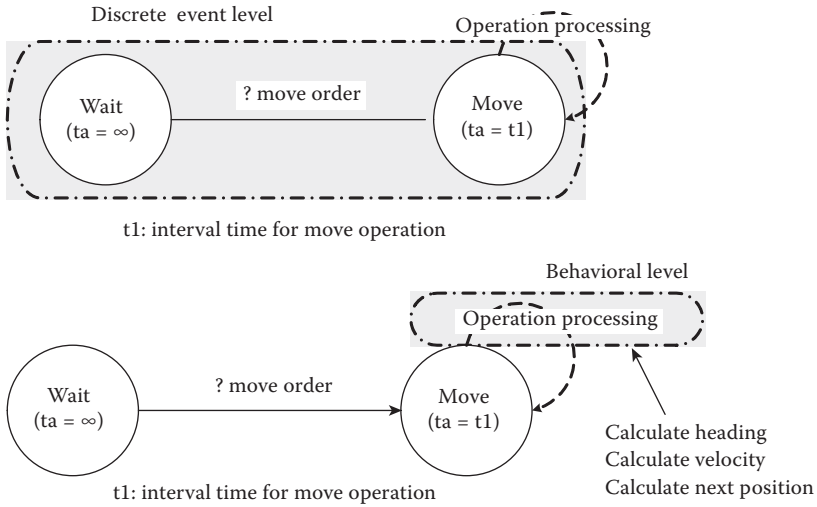


FIGURE 7.12 DEVS graph and classification of discrete event and behavioral level.

MOVE state of the DEM level relates to three operations—calculations of heading, velocity, and next position—defined at the BM level.

We can express the two models of the example as a layered structure. The BM is described at the OM layer and the DEM at the DES layer. The relation between the OM layer and the DES layer is shown by Figure 7.13. This also represents the modeling procedure to develop behavioral models and discrete-event models from domain requirements, which is described in textbooks and rules.

7.6.3 IMPLEMENTATION

Figure 7.14 shows an implementation example of the comodeling methodology. Figure 7.14a presents an example of the discrete-event model that represents maneuver behavior of a ship as we saw in Figure 7.12. When the model receives the *move_order* event, the state of the model is changed to the *MOVE* state and the *move* method is called every *t1* seconds. Figure 7.14b shows a function prototype that is used to calculate the next position of the ship in the *MOVE* state. The prototype shows input and output arguments of the function. The prototype should be defined by cowork of M&S and domain engineers when the model is partitioned or under development. Figure 7.14c displays possible implementation alternatives of the move function. The move function may use various algorithms and strategies such as uniformly accelerated motion, uniform motion along a line, and so on. Each function is developed as a separate dynamically linked library. The simulator that includes the DEVS maneuver model of Figure 7.14a will selectively use one of the candidate libraries. The users of the simulator explicitly select the libraries at run-time. The simulation results will be different according to what library the simulator application uses. In this manner, users can test various combinations of strategies and algorithms without recompiling the simulator itself.

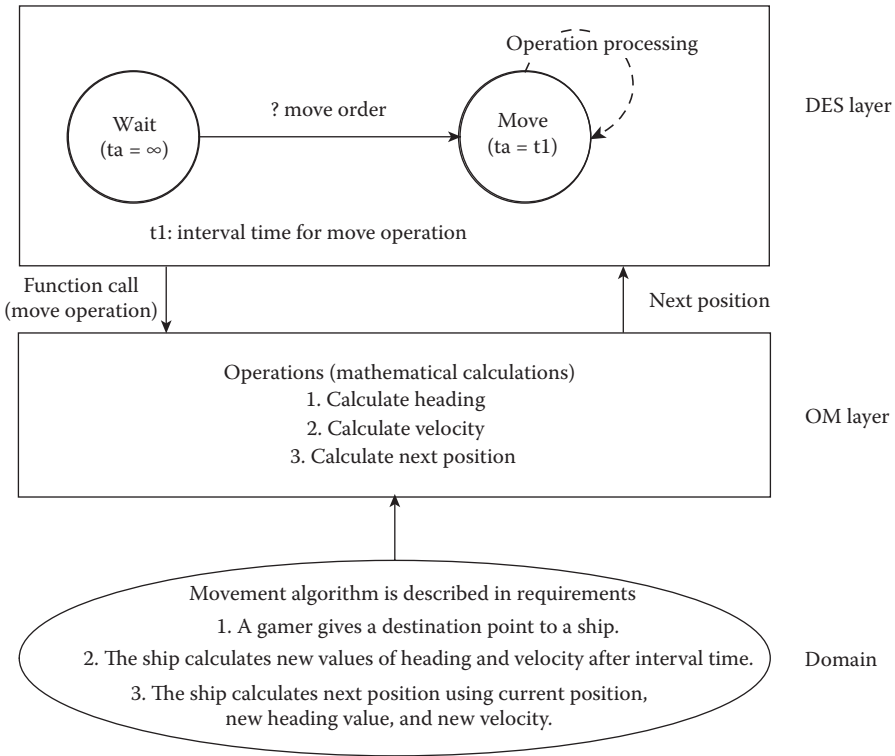


FIGURE 7.13 Example of layered structure of war game model.

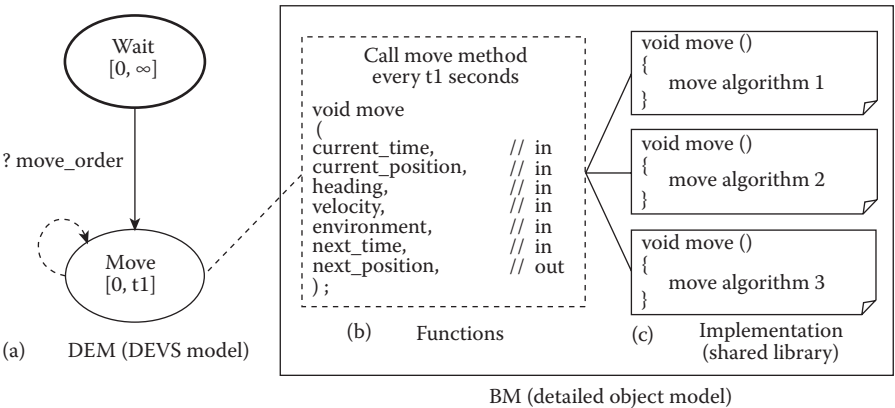


FIGURE 7.14 Implementation example of the ship's maneuver.

7.7 CONCLUSION

The OO comodeling methodology is an employment of development of DEVS models and lets experts do their jobs in the most efficient manner. The methodology partitions model specifications into two modeling parts: DEM and detailed object BM. The M&S engineers define the time associate output and the transition function using the DEVS formalism by employing operations defined in the object behavioral modeling. The discrete-event models are specified regardless of the algorithms used to implement object modeling operations. The domain engineers specify the object behavioral model by defining the operations. Platform engineers lend M&S engineers assistance to implement the simulation models. Detailed behavioral operations are implemented as dynamically linked libraries that are dynamically loaded after the simulator application starts. The proposed methodology enables simulator users to alter model behaviors or algorithms without recompiling the simulator. Accordingly, the comodeling methodology is a framework for the development of flexible and expandable models using OO and DEVS methodology.

REFERENCES

1. Roberts, C. A., and Y. M. Dessouky. "An Overview of Object-Oriented Simulation," *Simulation* 70, no. 6 (1998): 359–68.
2. Zeigler, B. P., H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. New York: Academic Press, 2000.
3. Wada, H., J. Suzuki, S. Takada, and N. Doi. "Leveraging Metamodeling and Attribute-Oriented Programming to Build a Model-Driven Framework for Domain Specific Languages," in *Proceedings of the 8th JSSST Conference on Systems Programming and Its Applications*, March 2005.
4. Hong, S.-Y., and T. G. Kim. "Embedding UML subset into object-oriented DEVS modeling process," in *Proceedings of the Summer Computer Simulation Conference*, 161–66, July, San Jose, CA, 2004.
5. Priestley, M. *Practical Object-Oriented Design with UML*. New York: The McGraw-Hill, 1996.
6. Zeigler, B. P. *Multifaceted Modeling and Discrete Event Simulation*, New York: Academic Press, 1984.
7. Kim, T. G., and S. B. Park. "The DEVS Formalism: Hierarchical Modular Systems Specification in C++," *1992 European Simulation Multiconference*, 152–56, York, UK, 1992.
8. DEVS-STD. [Online]. Available at: <http://www.devs-world.org>, 2005.
9. Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1998.
10. Rumbaugh, J. *Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.
11. Kruchten, P. "Architectural Blueprints—The '4 + 1' View Model of Software Architecture," *IEEE Software*, November 1995, 42–50.
12. Sung, C. H., S.-Y. Hong, and T. G. Kim. "Layered Approach to Development of OO War Game Models Using DEVS Framework," in *Proceedings of the Summer Computer Simulation Conference*, 65–70, July, Philadelphia, PA, 2005, 187.

13. Adams, J. K., and D. E. Thomas. "The Design of Mixed Hardware/Software Systems," in *Proceedings of the 33rd Annual Conference on Design Automation*, 515–20, June 1996, Las Vegas, NV.
14. Nordwig, A. "Object-Oriented Development of Simulation Models for Complex Hybrid Systems," *Modelling, Analysis and Design of Hybrid Systems*, LNCIS 279, 131–49, 2002.
15. Library Website. [http://en.wikipedia.org/wiki/Library_\(computer_science\)](http://en.wikipedia.org/wiki/Library_(computer_science)), 2009.
16. Kim, J.-H., and T. G. Kim. "Parametric Behavior Modeling Framework for War Game Models Development Using OO Co-Modeling Methodology," *2006 Spring Simulation MultiConference*, 69–75, April, Huntsville, AL, 2006, 193.
17. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules, Std 1516. New York: IEEE, 2000a.
18. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Federate Interface Specification, Std 1516.1. New York: IEEE, 2000b.
19. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Object Model Template (OMT), Std 1516.2. New York: IEEE, 2000c.

8 A Framework for Simulation-Based Structure and Parameter Optimization of Discrete-Event Systems

Olaf Hagendorf and Thorsten Pawletta

CONTENTS

8.1	Introduction	200
8.2	Simulation-Based Optimization	201
8.2.1	Parameter Optimization	202
8.2.2	Combined Parameter and Structure Optimization	204
8.3	Meta-Modeling—Specification and Organization of Model Sets	207
8.4	Framework for Modeling, Simulation, and Optimization	210
8.4.1	General Framework Structure	210
8.4.2	Interface: Optimization Module—Model Management Module	211
8.4.3	Interface: Model Management Module—Modeling and Simulation Module	214
8.4.4	Interface: Modeling and Simulation Module—Optimization Module	214
8.4.5	Algorithmic Summary of the Framework	215
8.5	Application Example	215
8.6	Summary	220
	References	221

Simulation with integrated parameter optimization of a given model structure is a well-established technique today. However, with increasing system complexity and flexibility the number of possible structure variants increases. Therefore the potential benefit of automatic model structure optimization becomes significant. During optimization, the introduced framework supports automatic parameter variation in concert with a reconfiguration of model structure. This is achieved by means of a combination of optimization, simulation, and model management methods. Using this approach, simulation is employed to determine the performance of a current

model structure and its parameters. An optimization method searches for an optimal solution with repeated, simultaneous model structure and model parameter changes. The model structure changes are assisted by a model management method.

8.1 INTRODUCTION

The research and application of simulation-based optimization have seen a significant development in recent years. A Google search on “Simulation Optimization” in 2006 found ca. 4,000 entries [1] in comparison to a search in 2008 that found almost 80,000 entries, among the results were articles, conference presentations, books, and software.

Until relatively recently, the simulation community was resistant to the use of optimization tools. Optimization models seem to over simplify the real problem and it was not always clear why a certain solution was the best [5]. The situation changed at the end of the 1990s. An ACM Digital Library [23] search on “Simulation Optimization” found 16,000 articles between 1960 and 2008. A significant number (15,500) of articles have been published during the last 20 years and only 500 articles in the 28 years before. Two reasons for this change may be the advances in modeling and simulation (M&S) methods and an increase of computing power over the past two decades that has enabled simulation-based optimization. Currently there are several algorithms to change simulation model parameters to establish solutions with good performance and methods to compare different solutions in terms of quality. Many commercially available discrete-event or Monte Carlo simulation software packages contain optimization methods to search for optimal input and system parameter values. Several such packages are described in April and colleagues [2].

This chapter addresses a fundamental problem of simulation-based optimization: The technique is well established but is restricted to the optimization of system parameters. In using this established technique, the model structure is considered to be fixed as the structure of model elements is defined during model development before an optimization experiment. As model performance is optimized, it may be necessary to redesign the model structure. This would conventionally be done manually by an analyst using previous simulation results, observations, or decisions based on previous experience. With increasingly complex, highly flexible, and dynamic structure models, the number of possible structure variants increases and the potential benefit of automatic model structure optimization would be significant.

The focus of this chapter is the description of a methodology for a simulation-based parameter and structure optimization for modular, hierarchical discrete-event systems. In contrast to current approaches that use modeling and simulation, here the model structure is variable and thus it is open to optimization. The variation of model structure and model parameters is controlled by a super-ordinate optimization module. The introduced simulation-based optimization framework consists of three main elements: (i) model management, (ii) modeling and simulation, and (iii) optimization.

- i. As a basis for the model management method the System Entity Structure/ Model Base (SES/MB) approach, introduced by Rozenblit, Zeigler et al. [20,27,28] is employed. The SES/MB approach is a generative, knowledge

base framework consisting of a tree-like SES and a MB containing basic components. It supports the definition of a set of modular, hierarchical models and the generation of specific model structures using predefined basic components from a MB. Because of this characteristic a modular, hierarchical modeling and simulation method has to be employed.

- ii. The modeling and simulation approach based on the Discrete Event System Specification (DEVS) formalism introduced by Zeigler [26,27] is an established method in the field of modular, hierarchical modeling and simulation. Dynamic Structure DEVS (DSDEVS) as an extension of DEVS offers methods to allow structural changes during a simulation run [6,15,25,27]. In countless applications, for example in Hagendorf and colleagues [8,9], the advantages of a dynamic structure modeling and simulation method are considerable. A DSDEVS method based on work in [9,15,16] is integrated in the novel simulation-based optimization approach. However, detailed aspects of DSDEVS systems are not considered in this chapter.
- iii. The optimization method controls the variation of model parameters and structure. Genetic algorithms have delivered robust solutions for various simulation-based optimization problems, for example in [17,18,24]. The genetic algorithm documented in [24] will be employed as an optimization method in the framework.

Section 8.2 provides a short preview of conventional simulation-based optimization aspects and introduces the fundamentals of a combined structure and parameter optimization approach. Section 8.3 briefly describes the applied SES/MB approach as a model set organization and a model generating meta-modeling method with necessary changes of the original approach. The synthesis of the three fundamental methods, optimization, model management, and modeling and simulation to perform a simulation-based structure and parameter optimization is presented in section 8.4. Finally, the usage of the new optimization approach is demonstrated by an industrial application in Section 8.5.

8.2 SIMULATION-BASED OPTIMIZATION

For all its achievements, a disadvantage of modeling and simulation is the missing optimization capability. For many years simulation experiments, as shown in Figure 8.1, have been state of the art. An analyst creates a model, for example based on a real system, transforms the model to an executable model, and executes a simulation with it. If necessary, after a review of simulation results the model configuration (i.e., model parameters and/or model structures) has to be manually changed by an analyst. Using a manual procedure only a relatively small number of system configurations can be examined until a suitable solution is chosen.

Through the combination of modeling and simulation with optimization methods, a simulation-based optimization approach is achieved that can reduce the effort of this manual procedure. Mathematical optimization generally means establishing a function minimum or maximum. Simulation-based optimization means finding the

best model configuration by minimizing a function of output variables estimated with a simulation method [21].

8.2.1 PARAMETER OPTIMIZATION

An established approach of a simulation-based optimization is the simulation-based parameter optimization. The overall goal of this optimization approach is the identification of improved settings of user selected model parameters under the control of performance measures. There is an extensive and varied body of literature on this topic that includes several tutorials, reviews, and summaries of the current state of the art (e.g., [3,4,7,14,21,22]). Law and Kelton describe [11] commercially available simulation tools with integrated optimization techniques using this approach of the simulation-based parameter optimization. Figure 8.2 shows a principle example of a simulation-based parameter optimization experiment. The procedure to create an executable model follows the procedure described in Figure 8.1. A crucial difference is the detachment of model and model parameters. Based on this detachment, the optimization method is able to alter model parameters to minimize the result of an objective function. The objective function measures the model performance with current model parameters. In most instances improving the model performance means minimizing the objective function result. Model parameter adjustments are carried out in a loop until a stop criterion is fulfilled. Examples of stop criteria are (i) going below a minimum alteration rate of objective function result or (ii) exceeding

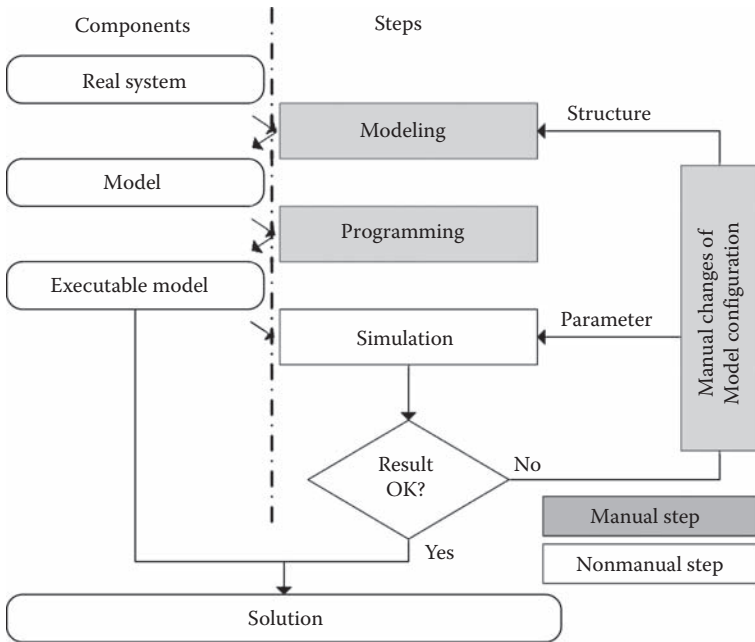


FIGURE 8.1 An example of a conventional simulation experiment.

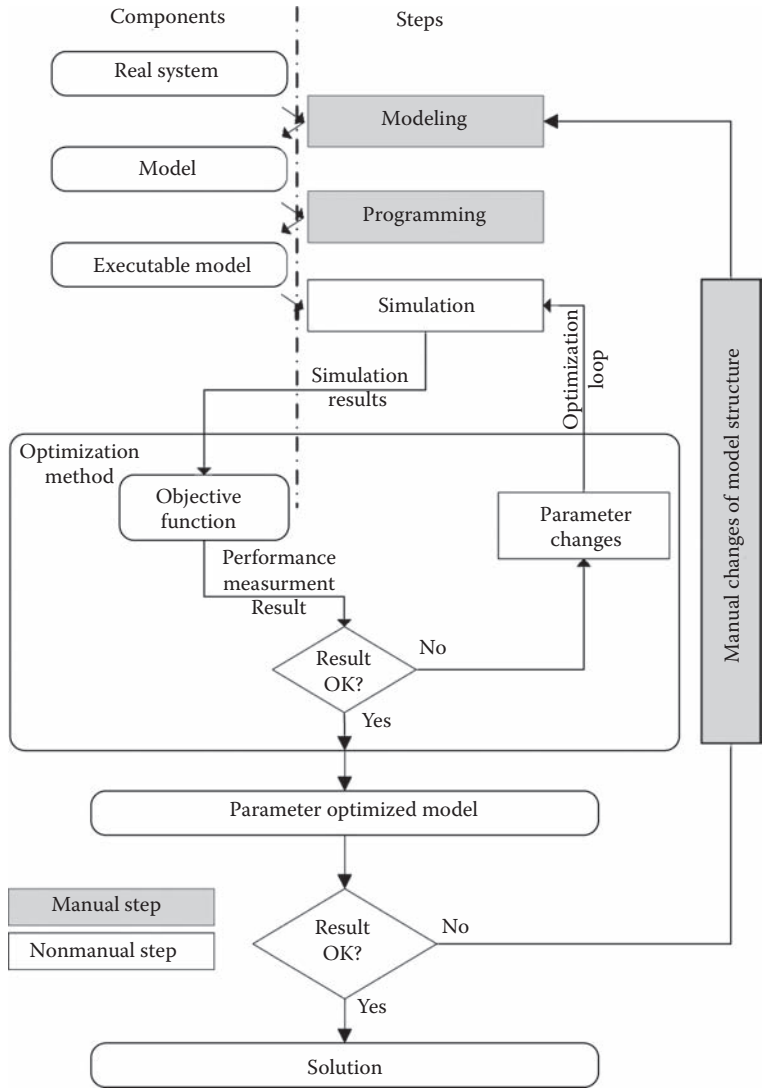


FIGURE 8.2 An example of a simulation-based parameter optimization experiment.

the maximum number of optimization cycles. The result of a successful optimization experiment is a parameter optimized model.

According to Swisher and Hyden [21], a simulation-based parameter optimization problem O with a set of m model parameters $X = \{x_1, \dots, x_m\}$ can be described as follows:

- A parameter set $X = \{x_1, \dots, x_m\}$ has the domain set $D = \{d_1 \dots d_m\}$.
- The multidimensional (one for each parameter) search space S is defined by $S = \{s = \{(x_1, v_1) \dots (x_m, v_m)\} \mid v_i \in d_i\}$.

- A set Y is the output set defined by $Y = \{y_1 \dots y_n\} = Y(X)$ and estimated by simulation. Simulation experiments are often based on stochastic parameters and properties. Hence the output set Y is stochastic too.
- The objective function F establishes a single stochastic value from output set $Y : F = F(Y) \rightarrow \mathbb{R}^+$. The result of the objective function is a measure of the current model performance.
- Because of the stochastic nature of Y and consequently of F , an estimation function R , the simulation response function defined by $R(X) = E(F(Y(X)))$, is optimized; that is, in the scope of this approach it is minimized.
- Depending on the optimization problem and analysis required, the exchange of the last two steps, evaluation of objective function F and simulation response function R , can save computational effort. Hence, the simulation response function is defined by $R(X) = E(Y(X))$ and subsequently the objective function by $F(X) = F(R(X))$.

Each parameter set $X_i \in S$ can be seen as a possible solution of O . The optimization method has to search the space S to find the parameter set $X_{\text{opt}} \in S$ with $E(F(Y(X_{\text{opt}}))) \leq E(F(Y(X))) \forall X \in S$. The resulting parameter set X_{opt} is considered the global optimum of O .

This approach is restricted to automated parameter optimization. It is important to note that automatic structure changes during optimization are not possible with this approach. Instead, structure changes are carried out manually by an analyst and each manual structure change requires a repetition of the automated parameter optimization.

8.2.2 COMBINED PARAMETER AND STRUCTURE OPTIMIZATION

The extension of the optimization approach with the ability to additionally change model structure to improve system performance is a development of the idea introduced in Section 8.2.1. This extension is mainly directed toward a simulation-based structure and parameter optimization as presented in Figure 8.3.

The approach of a simulation-based parameter and structure optimization differs in the following points from the simulation-based parameter optimization described in Section 8.2.1:

- An analyst does not generate a single model of the real system. In this case he has to organize a set of models. One way of achieving this is to define a model that describes a set of model variants instead of one single model of the system under analysis. Models that define the creation and interpretation of a set of models are named meta-models. If a model is the abstraction of an aspect of the real world, a meta-model is yet another, super-ordinate abstraction of the model itself. That is, when a model describes the behavior and structure of a real system then a meta-model describes the behavior and structure of different models that all

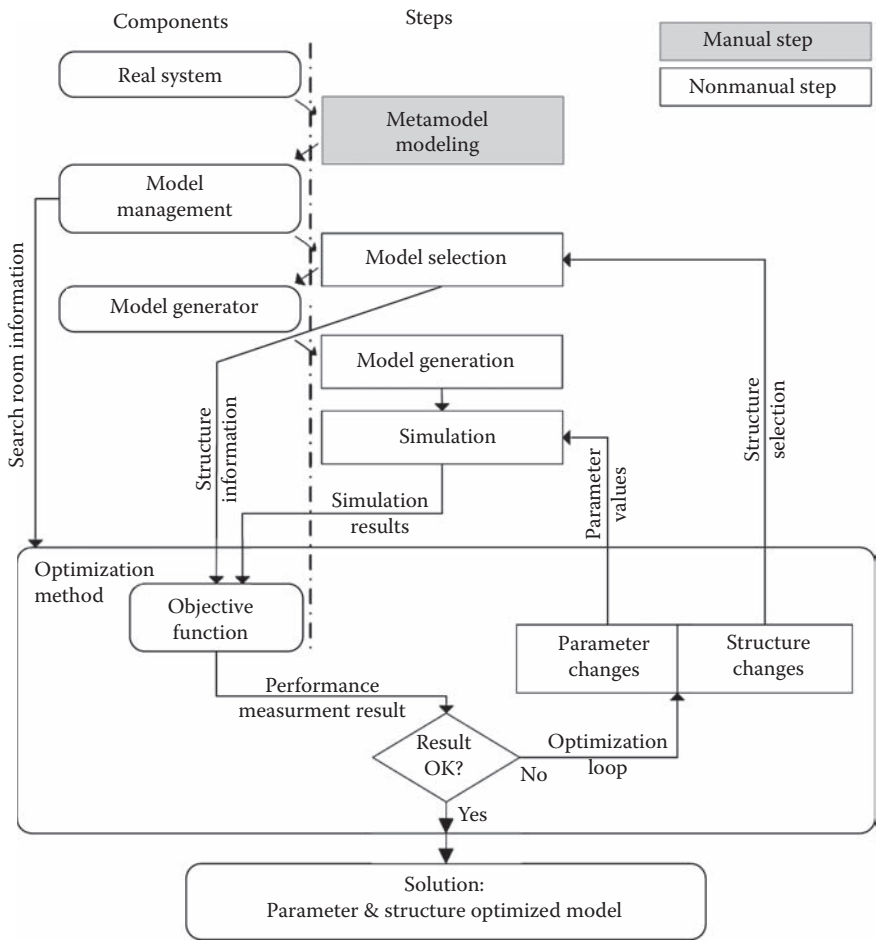


FIGURE 8.3 Components and steps of a simulation-based parameter and structure optimization experiment.

describe the behavior and structure of the same real system in a slightly different way.

- The model management organizes the set of model structures and provides a model selection method.
- The model selection is controlled by an optimization method. The selection method delivers the selected model structure information to a model generator that generates an executable model.
- The objective function receives simulation results and additional information gathered during model selection to estimate the performance of the current model configuration.
- The optimization method investigates the search space with the simultaneous model parameter and model structure changes without manual

involvement. The intention of the optimization method is to find a point in the search space with the optimal objective function result.

- The optimization process is separated into an initialization and an optimization phase:
 1. In the initialization phase, the model management module delivers information about the search space defined by the set of all model configurations to the optimization module.
 2. In the optimization phase, the model management module receives information from the optimization module about the currently investigated point in the search space. This information is used to select a new model structure and to initialize the model parameters.

A prerequisite for an optimization is the definition of a search space. In the approach presented here, the search space is multidimensional as a result of the combination of model structure and model parameter variants. During the optimization loop, several points of the search space are examined. Each point defines a model structure with an appropriate parameter set. The extension of the formal description of a simulation-based parameter optimization problem O , defined in Section 8.2.1, to a combined simulation-based structure and parameter optimization leads to O^* shown in a schematic diagram in Figure 8.4.

- The model parameter set X_p and its domain set D_p , in Section 8.2.1 defined as X and D , are extended by structure parameter set X_s and its domain set D_s . The extended set of definitions are: $X^* = X_p \cup X_s = \{x_{p1} \dots x_{pm}, x_{s1} \dots x_{sn}\}$ and $D^* = D_p \cup D_s = \{d_{p1} \dots d_{pm}, d_{s1} \dots d_{sn}\}$ with m model parameters in set X_p and n structure parameters in set X_s . The sets X_p and D_p are defined by the current model. The model management has to provide the sets X_s and D_s by analyzing the meta-model.
- The multidimensional (one for each parameter) search space $S = S_p \cup S_s$ is spanned by sets of model parameter and structure variants.
- The objective function F^* is defined by $F^*(Y(X^*), P(X_s))$ with simulation results $Y(X^*) = Y(X_s \cup X_p)$ and results based on structure related variables $P(X_s)$ that are established during the model selection. Because of the stochastic nature of the simulation results $Y(X^*)$ an estimation function R , the simulation response function, is calculated. The results based on structure related variables $P(X_s)$ are not stochastic. Hence, the simulation response function is defined by $R(Y(X^*))$ and subsequently the objective function by $F^*(R(Y(X^*)), P(X_s))$.

Through the inclusion of a model management method, the optimization method can simultaneously control parameter changes as well as the model structure changes to find an optimal system configuration. The model management method takes a crucial role in this approach. The description of a model management method based on meta-modeling follows in the next section.

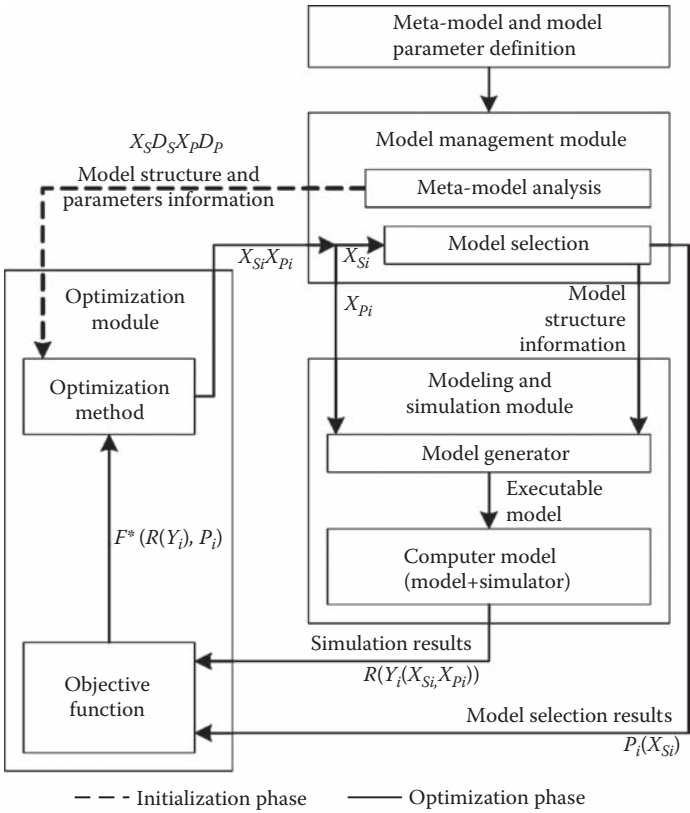


FIGURE 8.4 Schematic diagram of a simulation-based parameter and structure optimization experiment.

8.3 META-MODELING—SPECIFICATION AND ORGANIZATION OF MODEL SETS

Zeigler introduced [27] a simulation-based system design approach. It is a plan-generation-evaluation process. The plan phase organizes design alternatives with different model structures and model parameters within defined system boundaries to satisfy given design objectives. During the generation phase a specific model design is chosen and the corresponding model is generated. This model is simulated during the evaluation phase using an experimental frame derived from the design objectives.

The System Entity Structure/Model Base approach (SES/MB) [20,27] is such a simulation-based system design approach. It is specifically configured to define, organize, and generate modular, hierarchical models and was developed to assist an analyst in model organization and generation. To represent a set of modular, hierarchical models, the SES/MB approach is able to describe three relationships:

decomposition, taxonomy, and coupling. Decomposition means the formalism is able to decompose a system object called “entity” into subentities. Taxonomy means the ability to represent several possible variants of an entity called “specialization.” To interconnect subentities, the definition of coupling relationships are necessary. With these features the SES/MB approach meets the needs of the model management method in the proposed simulation-based optimization concept.

Fundamental properties of the SES/MB approach are [20,27]:

- A modular, hierarchical model is constructed based on: (i) the declarative system knowledge coded in a SES and (ii) predefined basic system models stored in a MB.
- The partitioning of a modular, hierarchical model is highly dependent on the design objectives. Model parameters are a typical example. They are not really a part of the model composition structure but nevertheless they can become a part of the SES if they are crucial for describing design alternatives.
- The model generation from a SES/MB is a multistage process. The first step is a graph analyzing and pruning process to extract a specific system configuration. Based on this information a modular, hierarchical model is generated.

The SES is represented by a tree structure containing alternative edges starting at decision nodes. With the aid of different edge types and decision nodes a set of different model variants can be defined. To choose a specific design and to create a specific model variant, the SES has to be pruned. The pruning process decides at decision nodes which alternative(s) to choose as a consequence of specified structure conditions and selection rules. The result of this process is a Pruned Entity Structure (PES) that defines one model variant. A composition tree is derived from a PES. The composition tree contains all necessary information to generate a modular, hierarchical model using predefined basic components from MB. Figure 8.5 shows the principal organization and the transformation process: $SES \rightarrow PES \rightarrow \text{Composition Tree} + MB \rightarrow \text{Modular, Hierarchical Model}$.

The used SES definition is based on definitions published in Zeigler and colleagues [20,27]. Figure 8.6 depicts an example to demonstrate the tree elements. The SES definition differentiates four main types of nodes: (i & ii) entity, (iii) specialization, (iv) aspect, and (v) multi-aspect. An entity node represents a system object. There are two subtypes of entity node in fact (i) atomic entity and (ii) composite entity. An atomic entity cannot be broken down into subentities. The MB contains a basic component for each atomic entity. A composite entity is defined in terms of other entities. Thus, the root node is always of type composite entity, while all leaf nodes are always of type atomic entity. The root node and each composite entity node of the tree have at least one successor node of type specialization, aspect, or multiple-aspect. That means there is an alternate

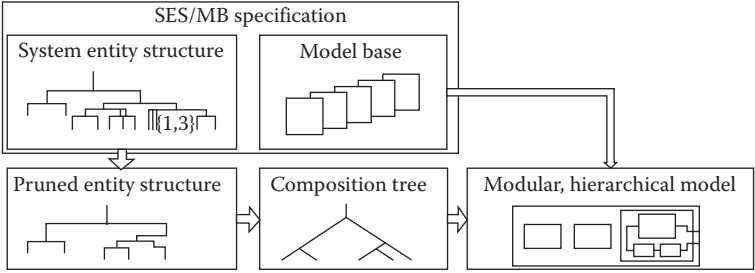


FIGURE 8.5 SES/MB formalism-based model generation.

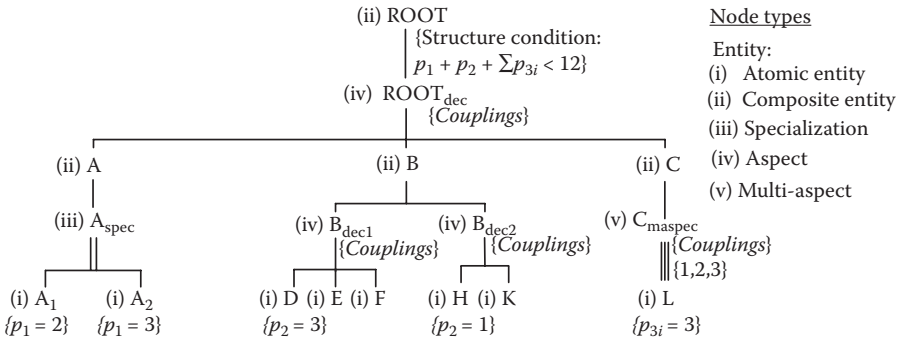


FIGURE 8.6 An example of a SES.

mode between entity nodes and other node types. The node type definitions can be briefly summarized:

- Atomic entity node = (name, {av₁,... av_n})
Composite entity node = (name, successors, {av₁,... av_n}, structure condition)
An entity node is defined by a name and is of type atomic or composite. Both node types may have attached variables av. A composite entity node can have a single successor node of type specialization or multi-aspect or multiple successor nodes of type aspect. A composite *entity node* can have attached structure condition.
- Specialization node = (name, successors)
A specialization node is defined by a name and a set of successor nodes. In the tree it is indicated by a double-line edge. A specialization node defines the taxonomy of a predecessor entity node and specifies how the entity can be categorized into specialized entities. A specialization node always has successor nodes of type atomic entity to represent the possible specializations. The specialization node A_{spec} in Figure 8.6 has two specializations defined by the nodes A₁ and A₂.

- Aspect node = (name, successors, coupling specifications)
An aspect node is defined by a name, a set of successor nodes, and coupling information. It is indicated by a single-line edge in an SES tree. An aspect node defines a single possible decomposition of its parent node and can have multiple successors of type atomic and/or composite entity. The coupling specification is a set of couplings and describes how the subentities, represented by the successor nodes, have to be connected. Each coupling is defined by a two-tuple. Each tuple consists of a subentity source and destination information, for example (SourceEntity.outputport, DestinationEntity.inputport). The composite entity B in Figure 8.6 has two decomposition variants defined by the aspect nodes B_{dec1} and B_{dec2} .
- Multiple aspect node = (name, successor, coupling specification, number range property)
The definition of a multiple aspect node is similar to an aspect node with an additional number range property. It has only one successor node of type atomic entity. It is indicated by a triple-line edge in a SES tree. A multiple aspect node also defines a decomposition of a composite entity, but all subentities have to be of the same entity type. Only the number of subentities is variable according to the attached number range property. The multiple aspect node C_{maspec} in Figure 8.6 illustrates the decomposition of composite entity C that may be composed by one, two, or three subentities L .
- Structure conditions are added to composite entity nodes. They are used as alternative structure knowledge representation instead of selection rules and structure constraints as defined in Rozenblit and Zeigler [20]. A modified pruning process necessitates an alternative representation. During the pruning, subtrees are cut. The remaining structure conditions are evaluated to verify the PES. Only if all structure conditions are true the PES is valid. Figure 8.6 shows an example of a structure condition added to the composite entity node $ROOT$. If the generated model structure contains the atomic entity nodes A_2, D, E, F, L , it would be valid because the condition $p_1 + p_2 + 1 * p_3 = 3 + 3 + 1 * 3 < 12$ is true.

8.4 FRAMEWORK FOR MODELING, SIMULATION, AND OPTIMIZATION

In this section a complete framework for combined parameter and structure optimization experiments is introduced. After a brief description of the general framework structure, its methods are discussed in detail and finally the entire algorithm is summarized.

8.4.1 GENERAL FRAMEWORK STRUCTURE

Based on the fundamental approach of a parameter and structure optimization experiment in Figure 8.4, the detailed structure of the introduced framework is depicted in Figure 8.7.

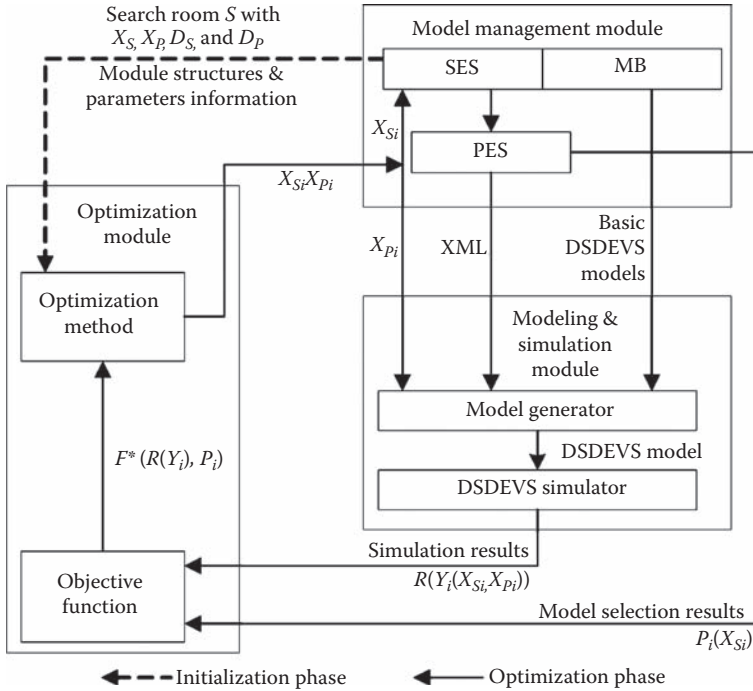


FIGURE 8.7 Structure of the optimization framework.

The framework consists of the three fundamental components:

- Model Management Module based on the SES/MB approach introduced in Section 8.3
- Modeling and Simulation Module based on DSDEVS [9,16]
- Optimization Module with a Genetic Algorithm (GA) as an optimization method

The appropriate interfaces to combine the above components are described next.

8.4.2 INTERFACE: OPTIMIZATION MODULE—MODEL MANAGEMENT MODULE

The optimization process consists of an initialization and an optimization phase. During the initialization phase, the Model Management Module has to analyze the SES tree to transform formal meta-model structure information into numerical data useable by the Optimization Module. Together with the model parameters the information is sent as initialization data to the Optimization Module. The information, coded in the four sets X_S , D_S , X_P , and D_P is used to build the set $X^* = X_P \cup X_S$ and the corresponding domain set $D^* = D_P \cup D_S$. During the optimization phase that is repeated in each optimization loop cycle, the optimization method calculates a

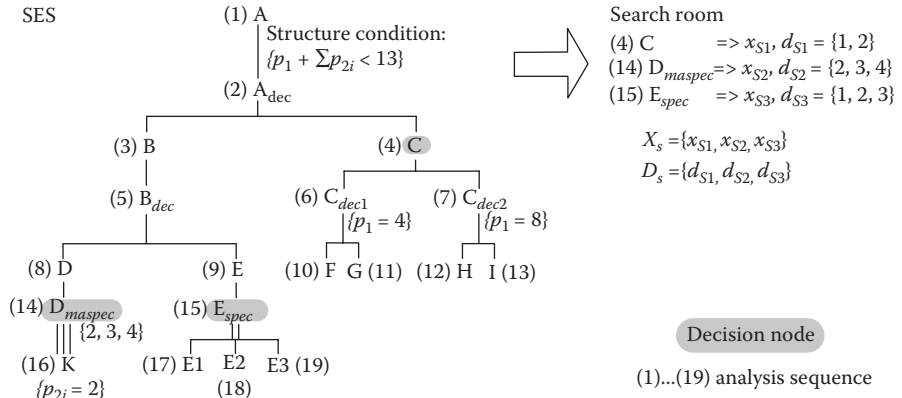


FIGURE 8.8 Transformation SES $\rightarrow$ set X_S and set D_S .

numerical data set $X_i^* = X_{pi} \cup X_{Si}$. The set X_i^* is sent to the Model Management Module, which determines, based on this information a new model configuration that is, a new model structure and initial model parameters.

The main task of the first transformation is to convert SES structure information to a structure parameter set X_S and the corresponding domain set D_S . This is done by a tree analysis using a breadth-first or depth-first algorithm, starting at the root node, traversing the tree and considering every node. If a node is a decision node (i.e., a specialization node, multiple aspect node, or composite entity node with alternative successor nodes), a structure parameter x_{Si} is added to the structure parameter set X_S and a corresponding domain d_{Si} to the domain set D_S . The domains of the specialization node and composite entity node are $\{1, \dots, \text{number of variants}\}$. The domain of a multiple aspect node is defined by its attached number range property.

Figure 8.8 illustrates the algorithm for creating structure parameter set X_S and the corresponding domain set D_S using a breadth-first algorithm. It starts at the root node A, a nondecision node. The next nodes are nondecision nodes A_{dec} and B. The composite entity node C is the first decision node. It has two alternative successors. A first parameter x_{S1} is added to set X_S with the domain $d_{S1} = \{1, 2\}$. The next examined nodes are B_{dec}, C_{dec1}, C_{dec2}, D, E, F, G, H, and I—they are nondecision nodes. The next examined node, the multiple aspect node D_{maspec} is a decision node. The value of its number range property is $\{2, 3, 4\}$. A second parameter x_{S2} is added to X_S with the domain $d_{S2} = \{2, 3, 4\}$. The next node, the specialization node E_{spec} is again a decision node. It has three alternative successor nodes. A third parameter x_{S3} is added to X_S with the domain $d_{S3} = \{1, 2, 3\}$. The last nodes analyzed, K, E₁, E₂, and E₃ are nondecision nodes. The example SES has three decision nodes. The resulting structure parameter set is $X_S = \{x_{S1}, x_{S2}, x_{S3}\}$ with the corresponding domain set $D_S = \{d_{S1}, d_{S2}, d_{S3}\}$ with the above determined domains. These sets, X_S , D_S , the model parameter set X_p , and its domain set D_p are used by the optimization method as the search space definition. Additional SES tree information (i.e.,

structure conditions and attached variables) are irrelevant during the initialization phase.

The second transformation is the reverse of the first. The Model Management Module receives a point in the search space from the Optimization Module; that is, the numerical data set $X_i^* = X_{Pi} \cup X_{Si}$, where set X_{Si} codes the model structure and set X_{Pi} codes its parameters. It has to synthesize the corresponding model structure and has to infer the model parameters. The transformation has to traverse the tree in the same direction as during the first in the initialization phase. At each decision node the next element of current structure parameter set X_{Si} is used to decide: (i) which successor of a composite entity node with alternative successor nodes is chosen, (ii) which successor of a specialization node is chosen, or (iii) how many successors of a multiple aspect node are incorporated into the PES. After pruning, the model structure is verified with the evaluation of the remaining structure conditions. If a structure is invalid the specific set X_i^* will be refused and this information is sent to the Optimization Module. In case of an invalid model configuration, the Optimization Module marks this point in the search space as prohibited and determines a new one.

Figure 8.9 illustrates the principle of this transformation. The breadth-first analysis starts at the root node A and continues as already described before. The first decision node is composition entity node C. The first element of X_{Si} is $x_{S1} = 1$; that is, the first aspect node C_{dec1} is chosen for the PES. The next decision node is the multiple aspect node D_{maspec} and the corresponding set element is $x_{S2} = 4$; that is, the PES contains four nodes K. The last decision node is the specialization node E_{spec} and the corresponding set element is $x_{S3} = 2$; that is, the PES contains the second specialization of node E_{spec} . After pruning, the attached variables are calculated and the PES is verified by evaluating the structure condition. In the example, the aspect node C_{dec1} and four atomic entity nodes K were chosen. Therefore, the structure condition at node A is evaluated as follows: $p_1 + \sum p_{2i} = 4 + 8 < 13$ and it follows that the PES is valid.

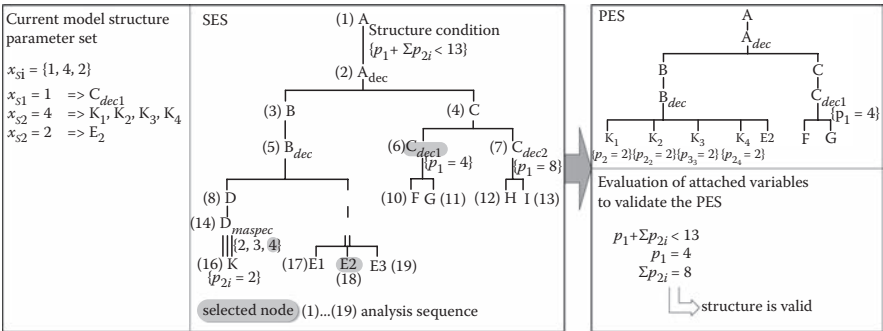


FIGURE 8.9 Transformation $X_{Si} + \text{SES} \rightarrow \text{PES}$.

8.4.3 INTERFACE: MODEL MANAGEMENT MODULE— MODELING AND SIMULATION MODULE

Each optimization cycle requires a change and adaptation of the simulation model. If the structure parameters in X_{Si} are changed, a new simulation model structure has to be generated. Otherwise, if just the model parameters in X_{Pi} are changed, it is adequate to reinitialize the model parameters. As illustrated in Figure 8.7 all necessary information is sent from the Model Management Module to the Model Generator of the Modeling and Simulation Module. The Model Management Module creates XML files describing the model structure. The DSDEVS basic components predefined in the MB, XML files, and current model parameters coded in set X_{Pi} are used by the Model Generator to generate the entire DSDEVS model.

The use of a standardized XML model description for information exchange decouples the two modules. It is based on W3C XML schema Finite Deterministic DEVS Models introduced in Mittal [12,13]. The XML interface uses the atomic and coupled model descriptions with model and port names. The coupled model description described in Mittal [13] is currently a work in progress and does not contain all the necessary description elements for this approach. Therefore, the composition description of coupled models used in the framework additionally defines submodel names and coupling specification. The decoupling of Model Management Module and Modeling and Simulation Module using XML files eases the modeling and verification of the basic components.

8.4.4 INTERFACE: MODELING AND SIMULATION MODULE—OPTIMIZATION MODULE

The objective function, defined in the Optimization Module (see Figure 8.7) estimates the performance of the current model configuration. The function gets its input parameters from the Modeling and Simulation Module. These are the simulation results $Y_i(X_{Si}, X_{Pi})$ and simulation response function results $R(Y_i(X_{Si}, X_{Pi}))$, respectively. Further input parameters are delivered by the Model Management Module. These are the model structure results $P_i(X_{Si})$, which are based on the evaluation of attached variables after pruning the SES. An example is illustrated in Figure 8.8. The aspect nodes C_{dec1} and C_{dec2} and the atomic entity node K define the attached variables p_1 and p_{2i} . After pruning illustrated in Figure 8.9, the values of p_1 and p_2 are calculated as follows: $P_i(X_{Si}) = \{p_1; \sum p_{2i}\} = \{4; 8\}$. These values may be used as further objective function parameters.

The result $F^*(R(Y_i), P_i)$ of the objective function is evaluated by the optimization method. As a consequence of the often stochastic nature of simulation problems, a random-based optimization method is preferable. Two established random-based algorithms inspired by the principle of the evolution of life are the Genetic Algorithm (GA) introduced by Holland [10] and the Evolutionary Strategy (ES) introduced by Rechenberg [19]. The origins of ES are continuous parameter problems whereas current GAs support hybrid problems. A disadvantage of the original GA is the missing memory. It is possible that in different generations the same individual is repeatedly

examined. Because of the time-consuming fitness estimation of an individual in simulation-based optimization, the addition of a memory method is vitally important. It has to store already examined individuals with their resulting $F^*(R(Y_i), P_i)$.

8.4.5 ALGORITHMIC SUMMARY OF THE FRAMEWORK

As described in the preceding, the proposed simulation-based parameter and structure optimization framework is composed of different methods that form a uniform optimization approach. The following algorithm summarizes the fundamental operations using a GA as an optimization method.

Initialization Phase:

0. Analyze the SES and establish $X^* = X_P \cup X_S$ and $D^* = D_P \cup D_S$
1. Initialize a population of individuals (generation 0) with different $X_i^* = X_{Pi} \cup X_{Si}$

Optimization Phase (repeat until stop criterion is fulfilled):

2. Estimate the fitness of all individuals of the current generation
Repeat for each individual
 - 2.1. Check memory if individual is known. In case of “true”: continue with next individual
 - 2.2. Prune SES with X_{Si}
 - 2.3. If structure condition is valid establish $P_i(X_{Si})$ or otherwise mark individual as invalid and continue with next individual
 - 2.4. Generate DSDEVS model
 - 2.5. Simulate DSDEVS model and get result $Y_i(X_{Si}, X_{Pi})$
 - 2.6. Evaluate the simulation response function $R(Y_i(X_{Si}, X_{Pi}))$ by repeating Step 2.5
 - 2.7. Evaluate the objective function $F^*(R(Y_i), P_i)$
 - 2.8. Store X_i^* and $F^*(R(Y_i), P_i)$ in memory
3. Select pairs with m individuals and create descendants using crossover
4. Mutate the descendants
5. Exchange individuals of the current generation with descendants based on a substitution schema to create a new generation

The next section demonstrates the application of the introduced framework with a project from an industry.

8.5 APPLICATION EXAMPLE

The example is based on developments and problems in the photofinishing industry and investigates a small part of a production process to demonstrate the approach. Photofinishing laboratories specialize in high volume production of thousands to millions of pictures per day. As a consequence of significant changes in the photography market, notably the introduction of digital cameras with a considerable reduction of analog and an increase of digital orders during recent years, a mix of analog

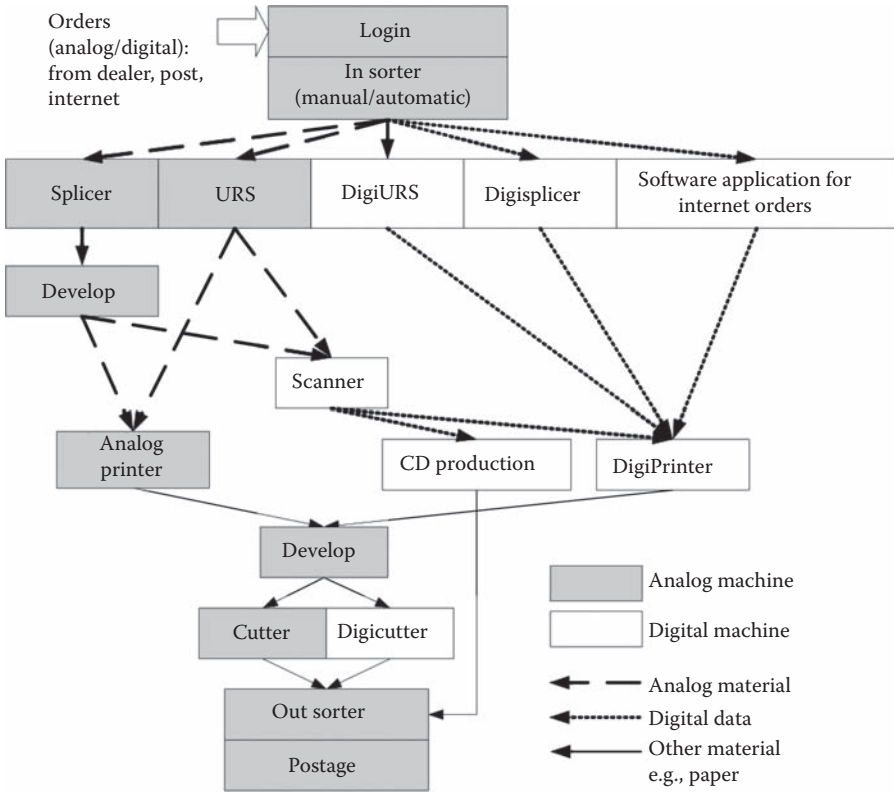


FIGURE 8.10 General product flows of a photofinishing lab.

and digital production facilities are used. The changes have led to concentration from many, local working, smaller laboratories to a few, large, nationwide working laboratories and fierce competition between them. The situation is driving an urgent need to be as cost effective as possible.

Figure 8.10 shows general structure and product flow through the different departments of a typical photofinishing laboratory. The material arrives in several forms at the login department. After sorting the product mixes, some 10–1000 single orders are combined into batches, each batch containing only one product type, for example, specific paper width and surface. The batch creation is done with different machine types: (i) a splicer combines undeveloped film rolls onto a film reel, (ii) a universal reorder station (URS) combines analog reorders to a strap of film strips, (iii) a digital URS scans the analogue reorders and produces a digital batch, (iv) a digital splicer handles data carriers (CDs, flash cards, etc.), and (v) software applications combine digital images received over the Internet. Undeveloped analog batches have to be developed and analog material can be scanned. The next steps are CD production, printing, paper development, and cutting. Finally items are packed and identified for delivery to customers. There are several possible material routes through

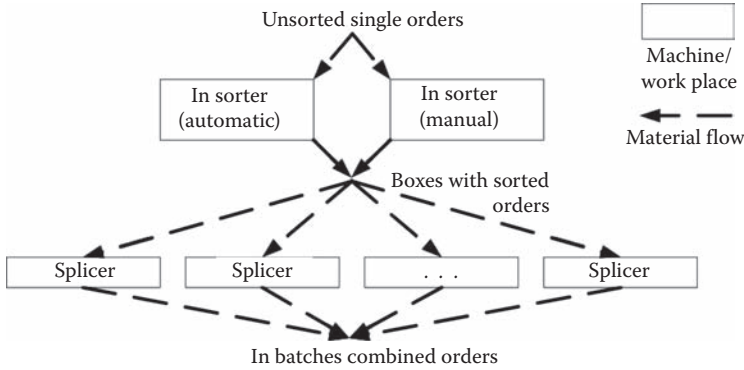


FIGURE 8.11 Product flow of the considered example.

production with the same end product but different processing time, machine and operator requirements, and costs. It is possible to employ fewer operators than available workstations and produce on time if an appropriate production structure and effective organization method are used to manage production.

The example is based on developments and problems in the photofinishing industry and investigates a small part of a production process to demonstrate the approach described in this chapter. For this example the login and splicer departments are studied in detail with a structure as in Figure 8.11. The source material (unsorted, single orders) is sorted by product type manually or automatically into boxes. The sorted orders are combined onto batch reels at splicers. An automatic sorter is handled by one or two operators, whereas manual sorting is done by the number of available operators without the need of a machine. The handling time depends on the number of machines, machine type, and the number of operators. A splicer is handled by one operator with a fixed average handling time. Operators can be moved between machines. The production time of a fixed number of orders, and, consequently, the cost, vary depending on the type and number of machines used, number of operators, and the strategy to organize operators. The task is to minimize the production time of a given number of orders while minimizing cost.

To validate the introduced framework, the global optimum estimated through simulation of all system variants is compared with the result of an optimization experiment. In both experiments the performance rating of one variant is done by the same objective function.

The simulation output of a single run delivers the production time and cost $Y = \{y_{\text{production time}}, y_{\text{costs}}\}$ of the currently investigated model variant. They are passed to the objective function. This function is defined by the term:

$$F = F(Y) = \alpha_1 * y_{\text{production time}} + \alpha_2 * y_{\text{costs}} \rightarrow \text{minimum}.$$

The factors α_1 and α_2 define the relevance of the variables. With $\alpha_1 = 1/\text{max\_production\_time}$ and $\alpha_2 = 1/\text{max\_costs}$ both variables are within the range 0 . . 1 and have

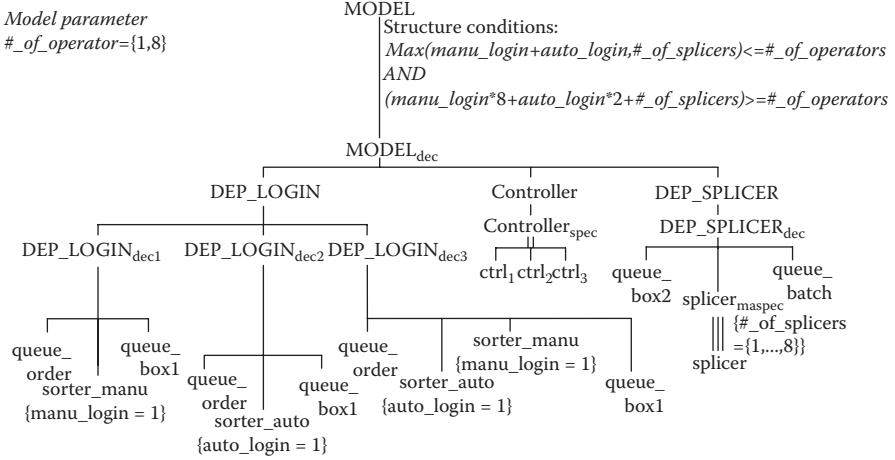


FIGURE 8.12 SES of the example.

the same relevance. The maximal value of the production time can be calculated by one simulation run and the maximal value of the costs is defined by the maximal number of operators, a model parameter with defined range.

Figure 8.12 depicts the SES, describing possible model structures of the considered example. The model variants are characterized by: (i) the usage of automatic and/or manual sorting, (ii) the usage of one to eight splicers, and (iii) the usage of one of three different department organization strategies to move operators between departments. Depending on chosen alternatives during the pruning process, several structure-related attached variables will be initialized with different values. The SES defines 72 model structure variants in all. Besides, there is one variable model parameter, the *number of operators* with a range of one to eight. The combination results in 576 model variants. Not all model variants define useful combinations. For example, a model with four operators and eight splicers delivers the same result as a model with four operators and four splicers. To exclude the useless variants the root node MODEL defines a structure condition that reduces the valid number of model variants to 275.

To solve this example, the search space has to be defined in terms of a structure parameter set, a model parameter set, and their corresponding domain sets. Using the principle introduced in Section 8.4.2, the structure parameter set and the corresponding domain set are defined by:

$$X_S = \{x_{DEP\_LOGIN}, x_{controllerspec}, x_{splickersmaspec}\}$$

$$D_S = \{d_{DEP\_LOGIN}, d_{controllerspec}, d_{splickersmaspec}\} \text{ with}$$

$$d_{DEP\_LOGIN} = \{1; 2; 3\}; d_{controllerspec} = \{1; 2; 3\}; d_{splickersmaspec} = \{1; 2; 3; 4; 5; 6; 7; 8\}.$$

The model parameter set and the corresponding domain set are defined by:

$$X_P = \{x_{\#\_of\_operators}\}$$

$$D_p = \{d_{\#\_of\_operators}\} \text{ with } d_{\#\_of\_operators} = \{1; 2; 3; 4; 5; 6; 7; 8\}.$$

Hence, the resulting search space is defined by:

$$X = X_P \cup X_S = \{x_{DEP\_LOGIN}, x_{controllerspec}, x_{splicermspec}, x_{\#\_of\_operators}\}.$$

Each model variant defines one point in the search space. With the principle introduced in Section 8.4, a PES can be derived and a corresponding model can be generated. One point in the search space is $X_{132} = \{2; 2; 2; 2\}$. This means that the aspect node DEP_LOGIN_{dec2} and the specialization $ctrl_2$ are chosen, the number range property value of the multiple aspect node $splicer_{maspec}$ is two, and the model parameter $\#\_of\_operators$ is also two. Figure 8.13 depicts the PES of model variant 132. The generated modular, hierarchical simulation model is illustrated in Figure 8.14. This model variant delivers the minimal objective function value. The fitness values of all 275 model variants are depicted in Figure 8.15.

The MATLAB® GA toolbox [24] is employed as the optimization method in this example. The default MATLAB GA parameter settings were used, except for a decreased population size of 15 and an adjusted stop criterion: if the weighted average change in the fitness function value over 20 generations is less than 0.01, the algorithm stops. The optimization experiment was repeated 100 times with different

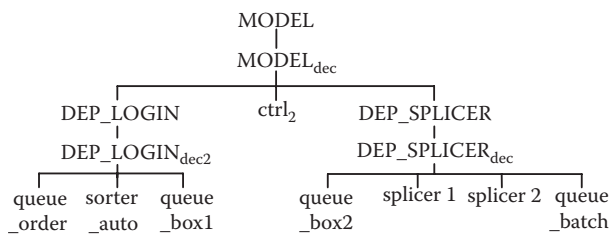


FIGURE 8.13 PES of 132th variant.

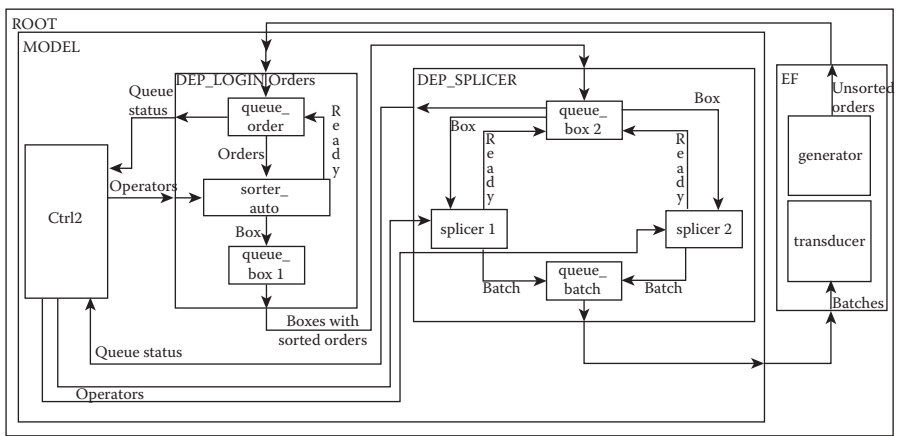


FIGURE 8.14 Modular, hierarchical model of 132th variant.

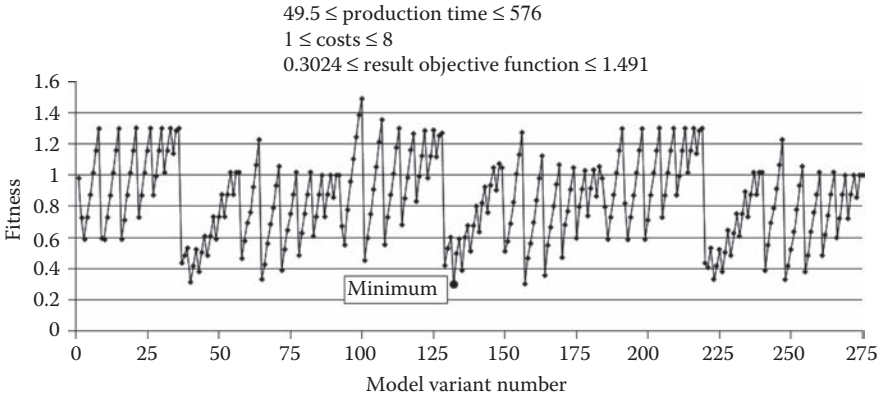


FIGURE 8.15 Fitness values of all variants with the optimum at X_{132} .

random number streams. The optimal structure and its parameter set were found after 194 simulation runs on average in contrast to 275 simulation runs of a complete enumeration experiment. The optimal solution X_{132} with the fitness 0.3024 was found 47 times. Other suboptimal solutions with a fitness value smaller than 0.35 were found 21 times. Nonoptimal solutions were found 32 times.

The results show that the introduced optimization framework delivers an optimal solution with significantly less simulation runs in comparison to a complete simulation study of all model variants.

8.6 SUMMARY

This chapter briefly summarized the fundamental aspects of simulation-based optimization. It introduced a novel approach for structure optimization of modular, hierarchical discrete event systems. The approach combines a model management method, modeling and simulation methods, and an optimization method to enable a concerted structure and parameter optimization. Appropriate interfaces between the different methods have been developed. The core of the interfaces are two transformations. The first one transforms formal structure information into numerical data, which are amenable to an optimization method. The second one transforms a specific numerical data set, calculated by the optimization method, into a specific model structure with its corresponding model parameters.

A prototype of the introduced framework was implemented with the Scientific and Technical Computing Environment MATLAB®. It consists of a MATLAB-based SES toolbox for model management, a MATLAB-based DSDEVS simulation toolbox, and the Genetic Algorithm and Direct Search Toolbox from The MathWorks®. The software prototype has been successfully used to prove the approach with first applications. The results of the described example demonstrate the advantages of the introduced approach. Using the implemented framework, the optimal structure and its corresponding model parameters are found with significantly less simulation runs in comparison to a complete simulation study of all model variants.

REFERENCES

1. April, J., Marco Better, M., Glover, F., Kelly, J., and Laguna, M. *Enhancing Business Process Management with Simulation Optimization. Proceedings of the 2006 Winter Simulation Conference*, 642–649, 2006.
2. April, J., Kelly, J., Glover, F., and Laguna, M. *Practical Introduction to Simulation Optimization. Proceedings of the 2003 Winter Simulation Conference*, 27–28, 2003.
3. April, J., Glover, F., Kelly, J., and Laguna, M. Simulation/Optimization Using “Real-World” Applications. *Proceedings of the 2001 Winter Simulation Conference*, 134–38, 2001.
4. Azadivar, F. Simulation Optimization Methodologies. *Proceedings of the 1999 Winter Simulation Conference*, 93–100, 1999.
5. Barnett, M. Modeling & Simulation in Business Process Management, 1–10. BP Trends Newsletter, White Papers & Technical Briefs, 2003. Available at www.bptrends.com. [Accessed November 20, 2008].
6. Barros, F. J. *Modeling and Simulation of Dynamic Structure Discrete Event Systems: A General Systems Theory Approach*. PhD Thesis. Coimbra, Portugal: University of Coimbra, 1996.
7. Fu, M. C., and Glover, F. W. Simulation Optimization: A Review, New Developments, and Applications. *Proceedings of the 2005 Winter Simulation Conference*, 83–95, 2005.
8. Hagendorf, O., Colquhoun, G., Pawletta T., and Pawletta S. A DEVS-Approach to ARGESIM Comparison C16 ‘Restaurant Business Dynamics’ using MatlabDEVS. *Simulation News Europe* no. 44/45, 58, December 2005.
9. Hagendorf, O., Pawletta, T., Pawletta, S., and Colquhoun, G. *An Approach for Modelling and Simulation of Variable Structure Manufacturing Systems*, 365–370, Liverpool, UK: ICMR, 2006.
10. Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Ann Arbor, MI: The University of Michigan Press, 1975.
11. Law, A. M., and Kelton, W. D. *Simulation Modeling and Analysis*. New York: McGraw-Hill, 2000.
12. Mittal, S. *DEVS Unified Process for Integrated Development and Testing of Service Oriented Architectures*, PhD Thesis. Tucson, AZ: University of Arizona, 2007.
13. Mittal, S. *W3C XML Schema Finite Deterministic DEVS Models*, <http://www.saurabh-mittal.com/fddevs/>, 2007.
14. Olafsson, S., and Kim, J. Simulation Optimization. *Proceedings of the 2002 Winter Simulation Conference*, 79–84, 2002.
15. Pawletta, T., Lampe, B., Pawletta, S., and Drewelow, W. A DEVS-Based Approach for Modeling and Simulation of Hybrid Variable Structure Systems. *Modeling, Analysis, and Design of Hybrid Systems*, Lecture Notes in Control and Information Sciences 279, 107–729, Edited by S. Engel, G. Frehse, and E. Schnieder. New York: Springer, 2002.
16. Pawletta, T., Deatcu, C., Pawletta, S., Hagendorf, O., and Colquhoun, G. *DEVS-Based Modeling and Simulation in Scientific and Technical Computing Environments*, 431–438, DEVS/HPC/MMS 2006, Huntsville, AL, 2006.
17. Pierrehval, H., Caux, C., Paris, J. L., and Viguier, F. Evolutionary Approaches to the Design and Organization of Manufacturing Systems. *Computers and Industrial Engineering* 44 (2003): 339–364.
18. Ray, J. P., and Tomas S. C. Simulation Optimisation Using a Genetic Algorithm. *Simulation Practice and Theory* 6 (1998): 601–11.
19. Rechenberg, I. *Evolutionstrategie* (German). Friedrich Frommann Verlag, 1972.

20. Rozenblit, J. W., and Zeigler, B. P. Concepts for Knowledge-Based System Design Environments. *Proceedings of the 1985 Winter Simulation Conference*, 223–231, 1985.
21. Swisher, J. R., and Hyden, P. D. A Survey of Simulation Optimization Techniques and Procedures. *Proceedings of the 2000 Winter Simulation Conference*, 119–128, 2000.
22. Swisher, J. R. Discrete-Event Simulation Optimization using Ranking, Selection, and Multiple Comparison Procedures: A Survey. *ACM Transactions* 4 (2003): 134–154.
23. The ACM Digital Library. <http://portal.acm.org>, 2009.
24. The MathWorks®. *Genetic Algorithm and Direct Search Toolbox™*, <http://www.mathworks.com/products/gads/>, 2008.
25. Uhrmacher, A. M., and Arnold, R. Distributing and Maintaining Knowledge: Agents in Variable Structure Environment. *5th Annual Conference on AI, Simulation and Planning of High Autonomy Systems*, 178–94, 1994.
26. Zeigler, B. P. *Multifaceted Modelling and Discrete Event Simulation*. New York: Academic Press, 1984.
27. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modelling and Simulation*. New York: Academic Press, 2000.
28. Zhang, G., and Zeigler, B. P. The System Entity Structure: Knowledge Representation for Simulation Modeling and Design. In *Artificial Intelligence, Simulation, and Modeling*, 47–73, Edited by L. E. Widman, K. A. Loparo, and N. R. Nielsen. New York: John Wiley, 1989.

9 Parallel Simulation of DEVS and Cell-DEVS Models in PCD++

Gabriel A. Wainer, Qi Liu, and Shafagh Jafer

CONTENTS

9.1	Introduction	224
9.2	Parallel Simulation	225
9.2.1	Parallel DEVS Abstract Simulator	225
9.3	Optimistic PCD++ —A Time Warp Based Parallel Simulation Engine for CD++.....	227
9.3.1	Introduction to Optimistic Simulation in PCD++	228
9.3.2	Message-Passing Organization.....	231
9.3.3	Cell-DEVS Algorithms for Optimistic Execution.....	233
9.3.4	Enhancements to Optimistic PCD++ and the Warped Kernel.....	234
9.3.4.1	Rollbacks at Virtual Time 0.....	234
9.3.4.2	User-Controlled State-Saving (UCSS) Mechanism.....	235
9.3.4.3	Messaging Anomalies.....	236
9.3.5	Optimization Strategies	238
9.3.5.1	Message Type-Based State Saving (MTSS)	238
9.3.5.2	One Log File per Node	238
9.4	Hybrid Optimistic Algorithms.....	239
9.4.1	Local Rollback Frequency Model.....	240
9.4.2	Global Rollback Frequency Model	241
9.5	Lightweight Time Warp Protocol	243
9.5.1	A Brief Review of the Simulation Process in Optimistic PCD++.....	245
9.5.2	Rule-Based Dual-Queue Event Scheduling Mechanism	245
9.5.3	Aggregated State Management and an Optimal Risk-Free State-Saving Strategy	248
9.5.4	Lightweight Rollback Mechanism	249
9.5.5	LTW Implications.....	250
9.6	Performance Analysis of LTW	251
9.6.1	Performance Analysis of the Conservative PCD++	252
9.6.2	Performance Analysis of Optimistic PCD++	253
9.6.2.1	Effect of the One Log File per Node Strategy	253
9.6.2.2	Effect of the Message Type-Based State-Saving Strategy	255
9.6.2.3	Performance of the Optimistic PCD++	256

9.6.3	Effect of Hybrid Optimistic Algorithms	258
9.6.3.1	Fire Propagation Model	259
9.6.4	Comparing Time Warp and Lightweight Time Warp	260
9.6.4.1	Experiment Platform and Metrics.....	260
9.6.4.2	Environmental Models.....	261
9.6.4.3	Test Results and Analysis	262
9.7	Summary	267
	References.....	268

9.1 INTRODUCTION

As we can see in the numerous examples found throughout this book, Discrete Event System Specification (DEVS) modeling and simulation (M&S) has become a widely used tool for tackling complex problems and supporting efficient decision-making in a broad array of domains. Nevertheless, as the system under study becomes more and more complex, the simulations tend to be time-consuming and resource-demanding. In the quest for better performance, parallel and distributed simulation technologies have received increasing interest, as these technologies allow executing simulations on a computing system using multiple processors interconnected by a communication network.

A parallel or distributed simulation typically comprises a collection of concurrent processes, each modeling a different part of the physical system and executing on a dedicated process, and interacting with each other by exchanging time-stamped event messages. The subtask executed by each process consists of a sequence of event computations, where each computation may modify the state of the process and/or schedule new events that need to be executed on the present process or on other processes. Unlike sequential simulations (which ensure that all events are simulated in time stamp order) parallel and distributed simulations use varied resources executing concurrently at different speeds, thus, we need to employ advanced synchronization mechanisms in order to guarantee the same results obtained with a sequential execution.

Such synchronization is key to parallel and distributed simulation. It ensures that each process complies with the *local causality constraint* [1], which requires that events are processed in nondecreasing time stamp order. Errors resulting from out-of-order event execution are referred to as *causality errors*. Synchronization techniques for Parallel Discrete Event Simulation (PDES) systems generally fall into two categories: *conservative approaches* that strictly avoid violating the local causality constraint, and *optimistic approaches* that allow violations to occur, but provide mechanisms to recover from them through an operation known as *rollback*. Usually, optimistic approaches can exploit a higher degree of parallelism available in the simulation, whereas conservative approaches tend to be overly pessimistic and force sequential execution when it is not necessary. Moreover, conservative approaches generally rely on application-specific information to determine which events are safe to process. Optimistic algorithms can also execute more efficiently with such information, but this is not needed for correct execution, allowing more transparent synchronization and simplifying software development. On the other hand, the overhead of state saving and rollback operations

incurred in optimistic simulations constitutes the primary bottleneck that may result in degradation of system performance.

CD++ [2] is an open-source M&S environment that implements both P-DEVS and Cell-DEVS formalisms and has been used to successfully solve a variety of sophisticated problems (see, e.g., Refs [3–5]). The CD++ environment has been ported to different platforms, including an embedded version [6], a stand-alone one, several parallel versions (conservative and optimistic synchronization protocols) [7–9], and a distributed version that supports Web-based simulations over the Internet [10]. In this chapter, we discuss the advanced techniques that have been developed for parallel simulation of DEVS and Cell-DEVS models in the PCD++ family of simulators. Specifically, we will cover the software architecture, parallel event execution paradigm, synchronization protocols, and performance optimizations in PCD++.

9.2 PARALLEL SIMULATION

The first parallel simulator, introduced in Troccoli and Wainer [7], was the first attempt to reduce simulation time in CD++ using parallel execution of models. It has been shown that this parallel simulator can speed up the execution of both DEVS and Cell-DEVS models in comparison to the stand-alone version [9]. This parallel simulator, presented in this section, was based on an approach exploiting the parallelism inherent to the DEVS formalism. Under that scheme, a single root coordinator acts as a global scheduler for every node participating in the simulation. Based on this structure, all events with the same timestamp are scheduled to be processed simultaneously on the available nodes. The simulator introduces two different types of coordinators; Head and Proxy to reduce interprocess communication. The simulator consists of a hierarchical structure creating a one-to-one correspondence between the model components and simulation objects.

9.2.1 PARALLEL DEVS ABSTRACT SIMULATOR

DEVS separates the model from the actual simulator engine. The abstract simulator creates a one-to-one correspondence between the model and the simulation entity as illustrated by Figure 9.1.

The simulation is carried out by DEVS processors that are of two types: simulator and coordinator. The simulator represents an atomic DEVS model, where the

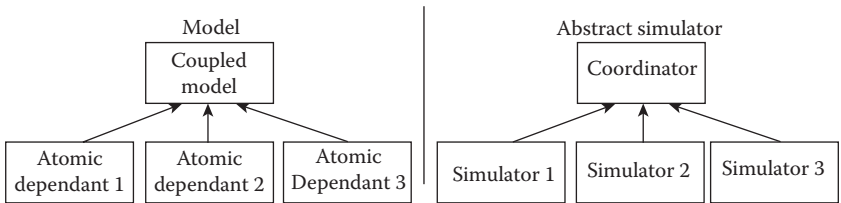


FIGURE 9.1 Correspondence between the model and the DEVS processors.

coordinator is paired with a coupled model. The simulator is in charge of invoking the atomic model's transition and external event function. On the other hand, the coordinator has the responsibility of translating its children's output events and estimating the time of the next imminent dependant(s). As shown in Figure 9.1 every coordinator has a set of child DEVS processors. When running parallel and distributed simulations, the whole model is divided among a set of logical process (LP), each of which will execute on a different CPU. In general terms, each LP will host one or more simulation objects. For the present discussion, those simulation objects will be DEVS processors.

At the beginning of the simulation, one LP will reside on each machine (physical process). Then, each LP will host one or more DEVS processors. This implies that not all of a coordinator's children are necessarily sitting on the same logical process. Due to the one-to-one correspondence, each coupled model is mapped to only one coordinator. A coordinator communicates with its child processors through intraprocess messaging if they reside on the same LP, and through interprocess messaging if they are sitting on remote LPs. Figure 9.2 shows a scenario at which a coupled DEVS model consisting of six atomic components is simulated using this simulator. The coordinator itself and three of its child processors are on the same logical process (LP0), where the other three child processors are hosted on another logical process (LP1). When the number of remote child processors of a coordinator is high, this design mechanism will lead to considerable overheads due to inter-process messages that are sent back and forth among the coordinator and its child processors. To overcome this issue, the concept of Head and Proxy Coordinators was introduced [7].

In the new design, a coordinator is assigned with each LP. As a result, all child processors will have a local coordinator through which they can communicate with remote child processors. The Head coordinator is responsible for synchronizing the

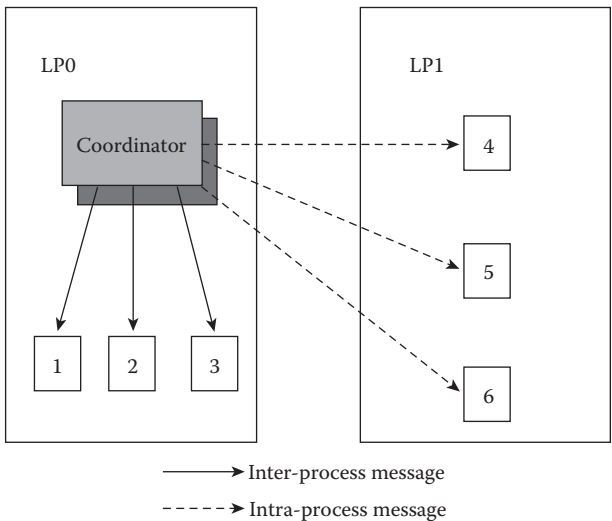


FIGURE 9.2 A single coordinator with remote and local child processes.

model execution, interacting with upper level coordinators, and exchanging messages among the local and remote model components. The Proxy coordinator is responsible for message exchange among the local model components, and forwarding local components messages to the Head coordinator if it resides on another LP. This structure organizes a DEVS processor into a hierarchy that does not have a one-to-one correspondence with the model hierarchy. Thus, a parent-child relationship that takes into account the existence of Head and Proxy coordinators must be defined as follows [7]:

1. For each simulator, the parent coordinator will be the parent's model local processor.
2. For each Proxy coordinator, the parent coordinator will be the model's Head coordinator.
3. For each Head coordinator, the parent coordinator will be the parent's model local processor; just as if it was a simulator.

Under this design, the simulation advances as a result of exchange of messages in the form of (type, time) between the parent and child DEVS processors, based on the original algorithm in Chow and Zeigler [11]. Two different types of messages exist: synchronization and content messages. The Collect message ($@, t$) is sent from a parent DEVS processor to its imminent children to tell them to send their outputs. The Internal message ($*, t$) is sent from a parent DEVS processor to its imminent children to tell them to invoke their transition function (either an external, internal, or confluent). The results produced by a model can be translated into Output messages (y, t) that are exchanged among a child DEVS processor and its parent. Finally, the External messages (q, t) represent the external messages arrived from outside the system or the ones generated as a result of an output message being sent to an influence.

9.3 OPTIMISTIC PCD++ —A TIME WARP BASED PARALLEL SIMULATION ENGINE FOR CD++

Originally introduced in Jefferson [12], Jefferson's Time Warp protocol is the first and most well-known optimistic synchronization. A Time Warp simulation is executed via several LPs interacting with each other by exchanging time-stamped event messages. Each LP maintains a *Local Virtual Time* (LVT) that changes in discrete steps as each event is executed on the process. A causality error arises if an LP receives an event with a time stamp less than its LVT. Such events are referred to as *straggler events*. Upon the arrival of a straggler event, the process recovers from the causality error by undoing the effects of those events executed speculatively during previous computations through the rollback operation. Due to the nature of optimistic execution, erroneous computations on an LP can spread to other processes via false messages. These false messages are cancelled during rollbacks by virtue of *anti-messages*. When an LP sends a message, an anti-message is created and kept separately. The anti-message has exactly the same format and content as the positive (original) message except in one field, a negative flag. Whenever an anti-message

meets its counterpart positive message, they annihilate one another immediately, hence canceling the positive one.

The Time Warp protocol consists of two distinct pieces that are sometimes called the local control and global control mechanisms [1]. The local control mechanism is provided in each LP to implement the rollback operations. To do so, an LP maintains three major data structures: an *input queue* of arrived messages, an *output queue* of negative copies of sent messages, and a *state queue* of the LP's states. The global control mechanism is concerned with such global issues as space management, I/O operations, and termination detection. It requires a distributed computation involving all of the processes in the system. The central concept of the global control mechanism is the *Global Virtual Time* (GVT), which serves as a floor for the virtual time of any future rollback that might occur. Any event occurring prior to GVT cannot be rolled back and may be safely committed. Therefore, the historical events kept in the input and output queues whose time stamp is less than the GVT value can be discarded. Similarly, all but the last saved state older than GVT can be reclaimed for each process. Furthermore, I/O operations with virtual time less than GVT can be irrevocably committed with safety. Destroying information older than GVT is done via an operation known as *fossil collection*. GVT computation and fossil collection are crucial components of the global control mechanism to reclaim memory and to commit I/O operations.

Over the years, many algorithmic and data structure based optimizations have appeared in the literature to improve the efficiency of the original Time Warp protocol (see, e.g., Refs [13–16]). The WARPED simulation kernel [17] is a configurable middleware that implements the Time Warp protocol and a variety of optimization algorithms. It relies on the Message Passing Interface (MPI) for high-performance communications on both massively parallel machines and on workstation clusters. Although the Time Warp protocol has been discussed in a great number of studies, its applicability to simulating DEVS models is only rarely explored in the PADS literature (but see, e.g., Refs [18–20]). The optimistic PCD++ engine, has been developed to allow optimistic simulation of complex and large-scale DEVS and Cell-DEVS models on top of the WARPED kernel [8,9], as shown in Figure 9.3.

9.3.1 INTRODUCTION TO OPTIMISTIC SIMULATION IN PCD++

The MPI layer and the operating system provide the communication infrastructure. The WARPED kernel offers services for creating different types of Time Warp LPs.

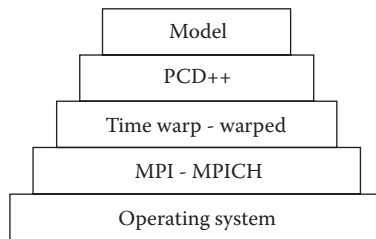


FIGURE 9.3 Layered architecture of the optimistic PCD++ simulator.

The PCD++ simulator defines two loosely coupled frameworks: the modeling framework and the simulation framework. The former consists of a hierarchy of classes rooted at *Model* to define the behavior of the DEVS and Cell-DEVS models; the latter defines a hierarchy of classes rooted at *Processor*, which, in turn, derives from the abstract LP definition in the WARPED kernel, to implement the simulation mechanisms. That is, the PCD++ processors are concrete implementations of LPs to realize the abstract DEVS simulators.

The Optimistic PCD++ employs a flattened structure consisting of four types of DEVS processors: *Simulator*, *Flat Coordinator (FC)*, *Node Coordinator (NC)*, and *Root* [8]. Introducing the FC and NC eliminates the need for intermediary coordinators in the DEVS processor hierarchy and, hence, minimizes communication overhead. Parallelism is achieved by partitioning the LPs onto multiple nodes. PCD++ processors exchange messages that can be classified into two categories: *content* and *control* messages. The former includes External (x, t) and Output message (y, t), which encode the data transmitted between the model components; the latter includes the Initialization (I, t), Collect ($@, t$), Internal ($*, t$), and Done message (D, t), which are used to implement a high-level control flow in line with the P-DEVS formalism.

Figure 9.4 shows the PCD++ structure of the LPs involving two nodes. A single Root coordinator is created on Node0 to start the simulation and to interact with the environment. The simulation is managed by a set of NCs running asynchronously on different nodes in a decentralized manner. The FC synchronizes its child Simulators and is responsible for routing messages between the child Simulators and the parent NC using the model coupling information. Specifically, the FC keeps track of the imminent Simulators (i.e., those Simulators that have state transitions scheduled at the present simulation time) and triggers state transitions at these imminent child Simulators by forwarding control messages received from the NC to them. In addition, when the FC receives a (y, t) from a child Simulator, it searches the model coupling information to find the ultimate destinations of the output message. A destination is ultimate if it is an input port on an atomic model or an output port on the topmost coupled model. If the (y, t) is sent eventually to remote Simulators or to the environment, the FC simply forwards the (y, t) to the parent NC. Otherwise, the FC translates the (y, t) into a (x, t) using the $Z_{i,j}$ translation function and directly

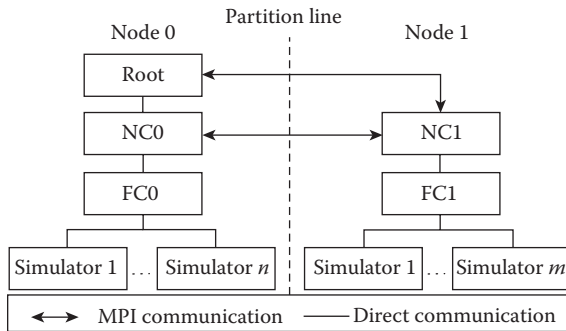


FIGURE 9.4 Optimistic PCD++ processor structure on two nodes.

forwards the (x, t) to the local receivers, which are recorded in a *synchronize set* for later state transitions. A Simulator executes the DEVS functions defined in its associated atomic model upon the arrival of control messages from the FC.

The message-processing algorithms originally proposed in Wainer and Glinsky [8] have been redesigned in Liu and Wainer [9] to allow for a more appropriate division of functionalities among the PCD++ processors and to address a variety of issues in parallel optimistic simulations. The major portion of the redesign effort focused on the message-processing algorithms for the NC. In the following, we summarize the main aspects of the new algorithms with an emphasis on the role of the NCs in the optimistic simulation.

The NC acts as the local central controller on its hosting node and the endpoint of internode MPI communication. It performs a number of important operations, including:

1. Performing inter-LP communications. A structure called *NC Message Bag* is used to contain the received external messages from other remote NCs. The time of the *NC Message Bag* is defined as the minimum time stamp among the messages contained in it, while an empty bag has a time of infinity.
2. Handling external events from the environment. The NC uses a structure called *Event List* to hold these external events. The current position in the *Event List* is referred by an *event-pointer* defined in the NC's state.
3. Driving the simulation on the hosting node. The NC advances the local simulation time to the minimum among: the time stamp of the external event pointed by the *event-pointer*, the time of the *NC Message Bag*, and the closest state transition time given in the (D, t) coming from the FC.
4. Managing the flow of control messages in line with the P-DEVS formalism. The NC uses a *next-message-type* flag to keep track of the type of the control message (either @ or *) that should be sent in the next simulation cycle.

Like the FC, the NC checks the destination of each received (y, t) message. If the (y, t) is sent to the environment, the NC directly forwards it to the Root coordinator. Furthermore, the NC finds out the remote nodes on which the ultimate receiving Simulators are based, in the model coupling and partition information. It then translates the (y, t) into a (x, t) and sends it to the remote NC on each of those nodes. On the receiving end, the (x, t) will be eventually delivered to the destination Simulators located on that node. When the NC receives a (D, t) from the FC, the NC calculates the next simulation time (referred to as *min-time*), if the *next-message-type* is @. If the *min-time* is larger than the user-specified stop time, the NC sets a *dormant* flag and exits. Otherwise, it sends any external messages scheduled at *min-time*, to the FC. Then, it sends a control message to the FC and resets the *next-message-type* accordingly, the *next-message-type* is set to * after the NC sends a $(@, t)$ to the FC [in which case, the output functions of the imminent Simulators will be invoked when the $(@, t)$ arrives]. The imminent Simulators perform internal transitions immediately after the output operations. Thus, the NC triggers the internal transitions by sending a $(*, t)$ to the FC in the next simulation

cycle. On the other hand, if there is no imminent Simulator, the NC sends a $(*, t)$ whenever external messages are flushed to the FC to trigger the external transitions in the receiving Simulators.

In an optimistic simulation, some LPs may have processed all their local events while waiting for other LPs. The lagging-behind LPs may send messages to the waiting LPs and reactivate them. The *dormant* state is used by the NC where all events scheduled on the local node have been processed. The NC exits the dormant state and reactivates the simulation on the hosting node upon the arrival of external messages from other remote NCs. In this case, the NC spontaneously flushes the received external messages with the minimum time stamp in its *NC Message Bag* to the FC. It also sends a $(*, t)$ to the FC to trigger the appropriate state transitions at the receiving Simulators.

9.3.2 MESSAGE-PASSING ORGANIZATION

Based on the message-processing algorithms just presented, we show a sample message-passing scenario using an *event precedence graph*, where a vertex (black dot) represents a message, and an edge (black arrow) represents the action of sending a message with the message type placed nearby. A line with a solid arrowhead denotes a (synchronous) intranode message and a line with a stick arrowhead denotes an (asynchronous) internode message. A lifeline (dashed line) is drawn for each PCD++ processor. The execution sequence of messages is marked by the numbers following the message type.

Figure 9.5 illustrates the flow of messages on a node with four LPs: an NC, an FC, and two Simulators (S1 and S2). As we can see, the execution of messages on a node at any simulation time can be decomposed into, at most, three distinct phases: *initialization phase (I)*, *collect phase (C)*, and *transition phase (T)*, as demarcated

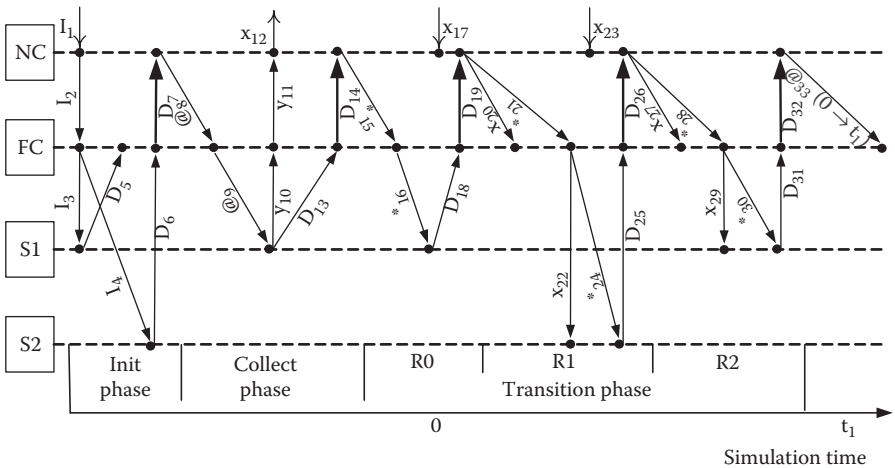


FIGURE 9.5 An example message-passing scenario on a node.

by the (D, t) message in bold black arrows received by the NC. Only one initialization phase exists at the beginning of the simulation, including the messages in the range of $[I_1, D_7]$. The collect phase at simulation time t starts with a $(@, t)$ sent from the NC to the FC and ends with the following (D, t) received by the NC. For example, the collect phase at time 0 comprises messages in range $[@_8, D_{14}]$. This phase is optional, it happens if there are imminent Simulators on the node at that time. Finally, the transition phase at simulation time t begins with the first $(*, t)$ sent from the NC to the FC and ends at the last (D, t) received by the NC at time t . In the diagram, messages in the range of $[*_15, D_{32}]$ belong to the transition phase at time 0. The transition phase is mandatory for each simulation time and may contain multiple rounds of computations [each starting with (x, t) followed by a $(*, t)$ sent from the NC to the FC and ends with a (D, t) to the NC]. In the example, the transition phase at time 0 has three rounds: R_0 with messages in range $[_15, D_{19}]$, R_1 with messages in $[x_{20}, D_{26}]$, and R_2 with messages in $[x_{27}, D_{32}]$. During each round, state transitions are performed *incrementally* with additional external messages and/or for potentially extra Simulators. Hereinafter, we will denote a transition phase of $(n + 1)$ rounds as $[R_0 \dots R_n]$.

To have a better understanding of the simulation process on each node, a high-level abstraction, called as *wall clock time slice* (WCTS), was introduced in Liu and Wainer [9]. A WCTS at virtual time t , stands for the execution of the events scheduled at time t on all the LPs mapped on a node. Figure 9.6 illustrates the WCTS representation of the simulation process on a node. Several properties of the WCTS are given as follows [9].

1. The simulation on a node starts with WCTS-0, the only WCTS that has all three phases.
2. WCTSs are linked together by messages sent from the NC to the FC (shown as black arrows in Figure 9.6). At the end of each WCTS, the NC calculates the next simulation time and sends out messages that will be executed by the local LPs at this new virtual time, initiating the next WCTS on the node. The linking messages between two adjacent WCTSs have a send time equal to the virtual time of the previous WCTS and receive time equal to that of the next, all other messages executed within a WCTS have the same send and receive time.
3. The completion of the simulation on a node is identified with a WCTS that sends out no linking messages (e.g., WCTS- t_n in Figure 9.6). The whole simulation finishes when all participating nodes have completed their portions of the simulation.

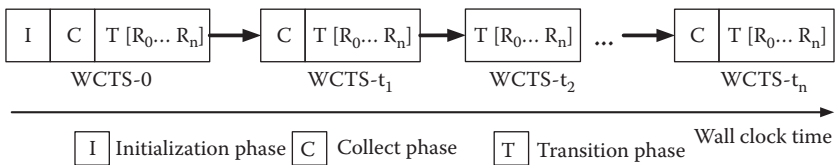


FIGURE 9.6 WCTS representation of the simulation process on a node.

4. WCTSs are *atomic* computation units during rollback operations. That is, the events executed within a WCTS are either committed as a whole (when GVT advances beyond the WCTS) or cancelled altogether. In the latter case, the simulation resumes after the rollbacks from unprocessed messages sent out from the previous WCTS (with virtual time less than the current rollback time).

9.3.3 CELL-DEVS ALGORITHMS FOR OPTIMISTIC EXECUTION

P-DEVS utilizes a message bag to store all simultaneous events scheduled for an atomic model so that all of them are available when the state transition at simulation time t is executed. This is necessary since all these simultaneous events created as (x, t) messages must be included in the computation to produce the correct state transition. Without explicit synchronization between the LPs, however, this requirement may not be satisfied in the optimistic simulation. Since the NC triggers state transitions speculatively based only on local information currently available on the hosting node, the state transitions at the atomic models may only involve (x, t) messages (those actually received by the Simulators). After the state transitions occurred at the Simulators, additional (x, t) messages may arrive, invalidating the results of the previous (speculative) state transitions. As shown in Figure 9.5, for instance, x_{23} arrives after the state transition triggered by $*_{16}$ has occurred at Simulator S1. This (x, t) message will be sent to the Simulator in the following round (R_2 in Figure 9.5) and, thus, involved in the additional state transition at S1 (x_{29} and $*_{30}$).

Since the state transitions are performed *incrementally* at the Simulators in the transition phases, the algorithms need to be adapted to this *asynchronous state transition paradigm* to obtain the same results as in a sequential simulation. A brief description of the new computation model under the asynchronous state transition paradigm is given as follows (detailed algorithms can be found in Liu and Wainer [9]).

1. Applying preemptive semantics to the state transition logic. For a transition phase with $(n + 1)$ rounds of computations, the state transitions in all but the last round (R_n) are based on incomplete information and, hence, false transitions. Only R_n has the best chance to perform the correct transition, which is the case if the WCTS is not rolled back later on in the simulation. Since the state transition in a later round involves additional external messages, it has a better chance to perform the correct computation and, thus, generate the correct results. Therefore, the state transition logic should be implemented so that the computation of the later round preempts that of the previous round. In the end, the potentially correct results obtained in R_n preempt those erroneously generated in R_{n-1} , and the simulation advances to the next virtual time. Both the value and state of the cell must follow this preemptive logic during the multiround state transitions. To do so, each cell needs to record its *previous value* and *previous state* passed in from the previous virtual time at the beginning of

R_0 for each WCTS. For time 0, the previous value and state are the cell's initial value and state defined by the modeler. Except the R_0 at time 0, the entry point of R_0 is identified by a change in the simulation time. Hence, a cell can record its previous value and state once a time change is detected at the beginning of the state transition algorithm. For time 0, this task is performed in the initialization phase.

2. Handling user-defined state variables. User-defined state variables may be involved in the evaluation of local rules defined for the cells. With the multi-round transition phase, this computation becomes more complex. During each round, a potentially different rule is evaluated and the state variables referenced in the rule are computed. Consequently, potentially wrong values are assigned to the variables and passed to the next round. The computation errors accumulate, and finally, wrong values are passed to the simulation at the next virtual time. To ensure correct computation of the state variables, a cell needs to record the values of the user-defined state variables at the beginning of each R_0 . These recorded values are inherited from the potentially correct computation of R_n of the previous WCTS. During the following rounds at a specific simulation time, the state variables are first restored to the recorded values. Only after this restoration operation, a new computation is performed. Therefore, the cell always uses potentially correct values as the basis for a new computation.
3. Handling external events. In CD++, the *port-in* transition function (for evaluating external events received from external models) is given a higher priority than the local transition rules. Under the new asynchronous state transition paradigm, the computation results of the *port-in* transition function may be overwritten by the local transition rules in later rounds. To preserve the effect of external events throughout the multi-round transition phase, an *event-flag* is set so that no further changes can be done to the cell's value in the following rounds at this time. The influence of the external event is spread out in the cell space as expected, and afterward the cell's value is again under control of its local transition rules.

9.3.4 ENHANCEMENTS TO OPTIMISTIC PCD++ AND THE WARPED KERNEL

In order to carry out optimistic simulations, several other issues need to be handled. This section covers the essential enhancements to the PCD++ and the WARPED kernel to ensure correct and efficient execution of simulations.

9.3.4.1 Rollbacks at Virtual Time 0

During rollbacks, the state of an LP is restored to a previously saved copy with virtual time *strictly less than* the rollback time. However, the problem of handling rollbacks at virtual time 0 is left unsolved in the WARPED kernel. If an LP receives a straggler message with time stamp 0, the state restoration operation will fail (negative virtual time is not found in the state queue). In optimistic PCD++, this problem is solved by explicitly synchronizing the LPs at an appropriate stage with an MPI Barrier so that no straggler message with time stamp 0 will ever be received by any LP in the

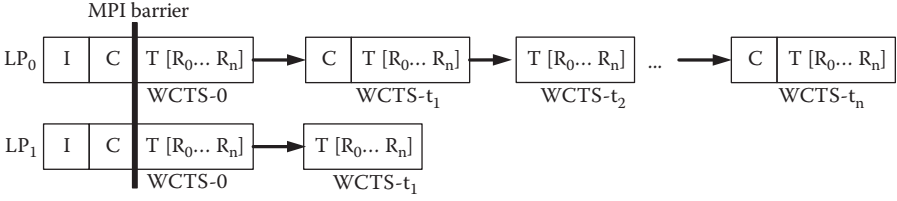


FIGURE 9.7 Using an MPI Barrier to avoid rollbacks at virtual time 0 in optimistic PCD++.

simulation. The best place to implement the MPI Barrier is after the collect phase in $WCTS-0$, as illustrated in Figure 9.7.

The underlying assumption of this approach is that all internode communication happens in the collect phase. Hence, messages with time stamp 0 are sent to remote LPs only in the collect phase of $WCTS-0$. The LPs are synchronized by the MPI Barrier at the end of this collect phase so that these messages can be received by their destinations before the simulation time advances beyond time 0. Therefore, no straggler with timestamp 0 will be received by any LP afterward. Once the LPs exit from the barrier, they can safely continue optimistic execution based on the standard rollback mechanism. The states saved for the events executed at virtual time zero provide the necessary cushion for later rollbacks on the processes. The cost of this approach is small, since the length of the synchronized execution is small when compared with the whole simulation.

9.3.4.2 User-Controlled State-Saving (UCSS) Mechanism

WARPED implements the *copy state-saving* (CSS) strategy using state managers (of type *StateManager*), which save the state of an LP after executing each event. The *periodic state-saving* (PSS) strategy is realized using state managers (of type *InfreqStateManager*) that save LP's state infrequently after a number of events. Simulator developers can choose to use either type of state managers at compile time. Once selected, all the LPs will use the same type of state managers throughout the simulation. This rigid mechanism has two disadvantages. (1) It ignores the fact that simulator developers may have the knowledge as to how to save states more efficiently to reduce the state-saving overhead. (2) It eliminates the possibility that different LPs may use different types of state managers to fulfill their specific needs at runtime. To overcome these shortcomings, a two-level *user-controlled state-saving* (UCSS) mechanism was introduced in Ref. [9] to provide a more flexible and efficient mechanism.

As shown in Figure 9.8, a flag called *skip-state-saving* is defined in each LP. The CSS policy only takes effect when the flag is false; otherwise, no state is saved after executing the current event. The flag is reset to false so that a new state-saving decision can be made for the next event. When the PSS strategy is used, an additional flag called *do-state-saving* with a lower priority is defined, if this flag is set to true by an LP, the state manager will save the state after every event (just like CSS). Therefore, simulator developers have the full power to choose the best possible combination of state-saving strategies at runtime.

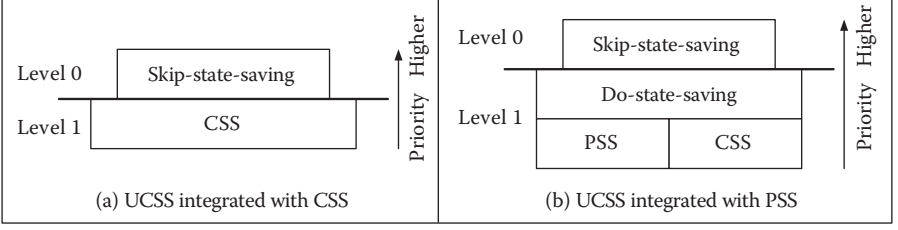


FIGURE 9.8 UCSS mechanism integrated with CSS and PSS strategies.

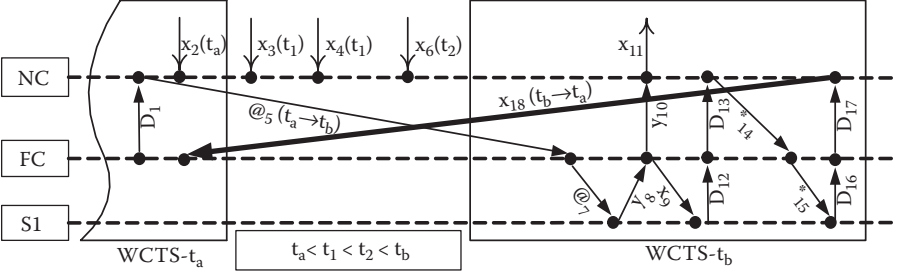


FIGURE 9.9 Example scenario of messaging anomalies.

9.3.4.3 Messaging Anomalies

In PCD++, the NC calculates the next simulation time ($\min$ -time) based on the time of its NC Message Bag. However, external messages with a time stamp less than the $\min$ -time may arrive after the calculation, invalidating the previous computation. In this case, the NC's speculative calculation of the $\min$ -time leads to *messaging anomalies* that cannot be recovered by the kernel rollback mechanism alone. Messaging anomalies will be detected when the control returns to the NC in the transition phase at the next (wrong) simulation time. Once found, the NC needs to perform cleanup operations to restore the simulation to the status before the previous wrong computation. An example scenario is shown in Figure 9.9.

Suppose that, when the last done message (D_1) is executed by the NC at the end of WCTS- t_a , there is no external message in its NC Message Bag and the closest state transition time carried in D_1 is t_b . Hence, the NC calculates the $\min$ -time as t_b , and sends a collect message ($@_5$) with send time t_a and receive time t_b to the FC, initiating WCTS- t_b on the node. Meanwhile, external messages x_2 , x_3 , x_4 , and x_6 (with time stamps less than t_b) arrive at the NC, invalidating the previously computed $\min$ -time t_b . Thus, the linking messages between WCTS- t_a and WCTS- t_b (e.g., $@_5$) are proven to be *false messages*. During the execution of D_{17} at the end of R_0 in WCTS- t_b , the NC calculates the $\min$ -time again based on its present NC Message Bag, which now contains the lagging external messages. The resulting $\min$ -time is t_a , the timestamp is x_2 . Hence, the NC sends an external message (x_{18}) with send time t_b and receive time t_a ($t_b > t_a$) to the FC. Since x_{18} is a straggler message for the FC, rollbacks propagate from the FC to the other processors immediately. Nonetheless, these rollbacks violate two assumptions made by WARPED: First, the rollback on FC is triggered by

an *abnormal straggler message* (x_{18}) with a send time *greater* than its receive time. Since the events are ordered by their send times in the output queues, this abnormal straggler message is misplaced in the NC's output queue, resulting in causality errors later on in the simulation. Secondly, the rollbacks occur right *in the middle of* executing the done message (D_{17}) by the NC. Therefore, the rollbacks are not transparent to the NC anymore.

Messaging anomalies can be classified into two categories. (1) *Anomaly with empty NC Message Bag*: In Figure 9.10a, if there are lagging external messages with time stamp t_a [e.g., $x(t_a)$] inserted into the *NC Message Bag*, the abnormal straggler message sent to the FC will have a time stamp of t_a . Hence, the LPs are rolled back to the end of $WCTS-t_{pre}$, the WCTS before $WCTS-t_a$. In this case, all the lagging external messages are removed from the *NC Message Bag* and no erroneous data is left in the state queues. (2) *Anomaly with non-empty NC Message Bag*: In Figure 9.10b, if there is no lagging external message with time stamp t_a arriving at the NC, the abnormal straggler message will have a time stamp of t_1 ($t_1 > t_a$). Hence, the LPs are rolled back to the end of $WCTS-t_a$, and the lagging external messages remain in the *NC Message Bag* after the kernel rollbacks.

To recover from potential messaging anomalies, the NC's message-processing performs additional cleanup operations after the completion of kernel rollbacks under abnormal situations [9]. The cleanup operations remove any remaining erroneous data generated during the anomalies from the input, output, and state queues. Only after the cleanup is the control passed to the *WARPED* kernel so that the simulation on the node can resume forward execution.

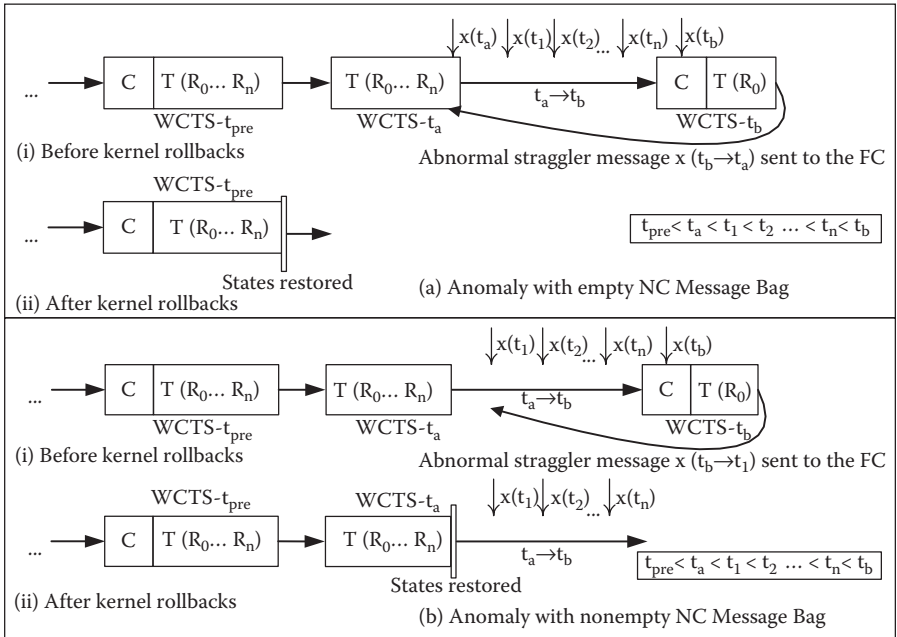


FIGURE 9.10 Two possible types of messaging anomalies in optimistic PCD++.

9.3.5 OPTIMIZATION STRATEGIES

Two optimization strategies have been integrated with the optimistic PCD++ simulator; the *message type-based state-saving* (MTSS) that reduces the number of states saved in the simulation, and the *one log file per node* strategy to break the bottleneck caused by file I/O operations [9].

9.3.5.1 Message Type-Based State Saving (MTSS)

During rollbacks, the state of a PCD++ processor is always restored to the *last* one saved at the end of a WCTS with virtual time strictly less than the rollback time. Hence, it is sufficient for a processor to save its state only after processing the last event in each WCTS for rollback purposes. The state-saving operation can be safely skipped after executing all the other events. The last event in a WCTS is processed at the end of R_n in the transition phase. Although the actual number of rounds in a transition phase cannot be determined in advance, we can at least identify the *type* of the messages executed at the end of the transition phases by a given processor. For the NC and FC, it must be a (D, t) , and for the Simulators, it should be a $(*, t)$. Since the Root coordinator only processes output messages, it still saves state for each event. The resultant state-saving strategy is called a MTSS, a specific type of UCSS. Considering that there are a large number of messages executed in each WCTS, and that they are dominated by external and output messages, MTSS can reduce the number of states saved in the simulation significantly. Consequently, the rollback overhead is reduced as fewer states need to be removed from the state queues during rollbacks. MTSS is risk-free in the sense that there is no penalty for saving fewer states, and it can be easily implemented using the UCSS mechanism. A PCD++ processor simply sets the *skip-state-saving* flag to true in all but the algorithm for the required type of messages. For example, a Simulator will set the flag to true in its algorithms for processing (I, t) , $(@, t)$, and (x, t) messages. The flag is left unchanged for $(*, t)$ since the Simulator needs to save state for these messages.

9.3.5.2 One Log File per Node

The PCD++ simulator provides message logging facilities for debugging, simulation monitoring, and visualization purposes. However, file I/O operation is a well-known performance bottleneck in parallel simulations, especially when the files are accessed via a Network File System (NFS). This is particularly severe for Time Warp since a file queue must be maintained in the kernel for each opened file (containing uncommitted data) and all the file queues are involved in rollback operations. To remove the bottleneck the *one log file per node* strategy was implemented [9]. Only one file queue is created for the NC on each node, which is shared among all the local LPs.

The advantages of this strategy are summarized as follows.

1. The required number of file descriptors for logging purposes is upper-bounded by the number of nodes involved in the parallel simulation, rather than increasing linearly with the size of the model.
2. The simulation bootstrap time is reduced considerably due to the decrease in the number of files opened in this process.

3. The rollback operations are accelerated since only one file queue needs to be maintained in the kernel.
4. The communication overhead is reduced as well since the data concentrated in a single file queue is flushed to the file system in bigger chunks, and less frequently, over the network.

9.4 HYBRID OPTIMISTIC ALGORITHMS

The near-perfect state information (NPSI) protocols proposed by Srinivasan [21] are a new class of synchronization for PDES that outperforms Time Warp, both temporally and spatially. NPSI-based protocols dynamically control the rate at which processes exploit parallelism, achieving a more efficient parallel simulation. In optimistic protocols such as Time Warp, LPs execute events aggressively, assuming freedom from errors. Thus, the aggressive event execution would include risk that is the potential at which erroneous results propagate to other LPs [23]. The NPSI protocols aim at controlling both aggressiveness and risk of optimism adaptively by computing an error potential (EP). The EP of a process is defined as a function of the states of other LPs participating in the simulation. It works as an elastic force that sometimes blocks and sometimes frees the progress of the LP.

The optimism implemented by the Time Warp protocol incurs three time costs: state saving, rollback, and memory management [21]. Furthermore, by restricting optimism, time cost gets introduced; the lost opportunity cost is defined as the potential loss in performance when an LP is suspended while it was safe for it to continue. Thus, protocols that control optimism define the cost function as follows:

$$\begin{aligned} \text{Total cost} = & \text{state saving cost} + \text{rollback cost} + \text{memory management cost} \\ & + \text{lost opportunity cost.} \end{aligned}$$

Since the time cost of state saving can be a function of the size of the state and the frequency rate at which it is saved, the Total cost function can be rewritten by omitting the state saving cost as:

$$\text{Total cost} = \text{rollback cost} + \text{memory management cost} + \text{lost opportunity cost.}$$

By limiting optimism, the first two costs can be reduced but the lost opportunity cost would increase in return, as illustrated in Figure 9.11.

The best performance is attained when the controlled optimism eliminates both rollback and memory management cost and in return adds zero lost opportunity cost [21]. Optimism limiting protocols can thus achieve a good balance by precisely identifying incorrect computations and avoiding their propagation. This can be done by providing each LP with perfect state information about other LPs, but due to various latencies in computing distributed snapshots, it is impossible to obtain this information. The NPSI mechanism approximates perfect state information by using a dynamic feedback system that operates asynchronously with respect to LPs,

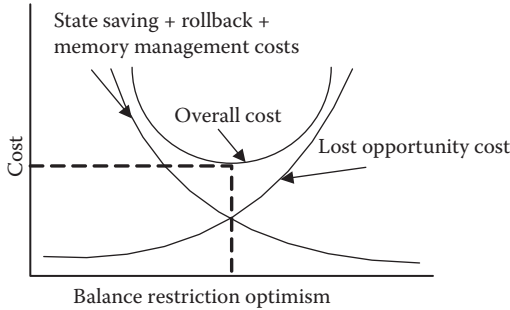


FIGURE 9.11 Tradeoff introduced by limiting optimism. (From Srinivasan, S. and Reynolds, P. F., *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 2, 103–39, 1998.)

providing them with NPSI at low-cost. NPSI controls aggressiveness and risk of LPs by dynamically computing NPSI.

In order to control the optimism of PCD++, we have modified WARPED to implement an NPSI mechanism based on the number of rollbacks. The idea is to reduce the number of rollbacks by suspending the simulation object within the LP that has a large number of rollbacks and therefore blocking it from flooding the net with anti-messages. However, the LP will still be able to receive input events and they will be inserted into the corresponding message bags. After a predefined duration, the suspend simulation object is released and will go on simulating. We have implemented two new protocols, namely Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM) [22] to limit the optimism of the PCD++ simulator.

The main concept is to associate each LP with an EP to control the optimism of the LP. During the simulation run, the value of each EP is kept updated by evaluating a function M1, which uses state information that is received from the feedback system. Then, a second function M2 dynamically translates every update of the EP in delays of execution events.

9.4.1 LOCAL ROLLBACK FREQUENCY MODEL

The LRFM protocol is based on local information of the LPs; the simulation object within the LP will be suspended or allowed to continue only based on the number of rollbacks it had. First, M1 and M2 functions must be defined:

Function M1: The EP is the number of rollbacks that the object had from a time T_1 until the current time T_2 , having that $T_2 - T_1 \leq T$, where T is the interval after which the local number of rollbacks of the simulation object gets reset to zero.

Function M2: If the number of rollbacks at the interval T is greater than a specified value, then the object is suspended. The LP where the object resides on will still be able to receive incoming events, but the events are not processed until the object

```

1. In each LP, at the beginning predefine:
   max_rollback and period
2. In each simulation object, at the simulation start:
   previous_time = 0
3. In each object, when the LP is scheduled to run:
   actual_time = Warped.TotalSimulationTime()
   if (actual_time - previous_time >= period)
       simulateNextEvent()
       previous_time = actual_time
       rollbacks = 0
   else
       if (rollbacks < max_rollback)
           simulateNextEvent()
       /*else, SUSPEND the simulation object */

```

FIGURE 9.12 LRFM algorithm.

is given permission to resume. However, if the number of rollbacks is less than the predefined value, the object simulates, using optimistic behavior.

To implement this protocol each LP has to be informed about two values: *max_rollback*, and *period*, where *max_rollback* is the maximum number of rollbacks allowed before, and *period* is the duration of suspension. The algorithm is presented in Figure 9.12. We can identify the following three scenarios:

1. The LRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given permission to continue.
2. The LRFM period has not yet expired, if the number of rollbacks of the simulation object is less than the allowable range (i.e., *max_rollback*), the simulation object continues.
3. The LRFM period has not yet expired, but the number of rollbacks within the simulation object has exceeded *max_rollback*, thus, the simulation object gets suspended for the entire duration of the current LRFM period.

In order for an LP to be able to simulate objects that must not be delayed, we have modified the scheduler policy to choose the next object to simulate. It chooses the first object of the input event list (i.e., the ones with the lowest timestamp) only if the rollbacks count does not exceed *max_rollback*; else, the scheduler checks the next object until it finds an object in a condition to be simulated or until it reaches the end of the list.

9.4.2 GLOBAL ROLLBACK FREQUENCY MODEL

In the GRFM protocol each simulation object uses global information in such a way that among all the simulation objects residing on all LPs, the one with the largest number of rollbacks must be suspended for the duration of time defined at the

beginning of the simulation. Therefore, at each simulation cycle all the LPs must broadcast the information regarding the rollback counts of all of their simulation objects. As in LRFM, M1, and M2 functions must first be defined:

Function M1: The EP is the number of rollbacks that the object had minus the maximum number of rollbacks of the other simulation objects (both local and remote) participating in the simulation, from a time $T1$ until the current time $T2$, with $T2 - T1 \leq T$, where T is the interval after which the local number of rollbacks of the simulation object gets reset to zero.

Function M2: If the number of rollbacks at the interval T is greater than the number of rollbacks of the other simulation objects, then the object is suspended. The LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object resumes. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates the usual optimistic behavior.

This algorithm is presented in Figure 9.13. As we can see, there are three different scenarios:

1. The GRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given permission to continue.

```

1. In each LP, at the beginning predefine: period
2. In each simulation object, at the beginning predefine:
   previous_time = 0
   max_rollbacks = 0
3. In each simulation object, when the LP is scheduled to run:
   actual_time = Warped.TotalSimulationTime()
   if (actual_time - previous_time >= period)
       simulateNextEvent()
       previous_time = actual_time
       rollbacks = 0

   else

       if (rollbacks < max_rollbacks)
           simulateNextEvent()
       /*else, SUSPEND the simulation object*/
4. For i from 1 until the number of LPs
   if (i is NOT this LP id)
       send to LP i the number of rollbacks of the objects of
       the LP id
   Subroutine that receives the number of rollbacks from
   other LP:
   For j from 1 until the numbers received
       If (rollbacks[j] > max_rollbacks)
           max_rollbacks = rollbacks[j]

```

FIGURE 9.13 GRFM algorithm.

2. The GRFM period has not yet expired; if the number of rollbacks of the simulation object is less than the allowable range, the simulation object continues its normal execution.
3. The GRFM period has not yet expired, but the number of rollbacks within the simulation object has exceeded *max_rollbacks*, thus, the simulation object is suspended for the entire duration of the current GRFM period.

The main difference of GRFM and LRFM is the way *max_rollbacks* is initialized. In LRFM, the maximum allowable rollbacks is predefined by the user at run time, while in GRFM the maximum allowable rollbacks is set to the largest number of rollbacks of all participating simulation objects. That is, whenever a simulation object is scheduled to execute, it must send the number of rollbacks it had so far to all other simulation objects, both local and remote ones. As a result, at any time *max_rollbacks* is the largest number of rollbacks among all the existing simulation objects.

9.5 LIGHTWEIGHT TIME WARP PROTOCOL

As mentioned earlier, the operational overhead incurred in a Time Warp based optimistic simulation constitutes the primary bottleneck that may degrade system performance. Broadly speaking, Time Warp has two forms of operational overhead: *time overhead* and *space overhead*. The former consists of the CPU time required to perform the local and global mechanisms, while the latter is the result of historical data stored by each LP in its input, output, and state queues. Several technical challenges must be addressed to tackle the issues related to performance, scalability, and increased complexity of Time Warp based large-scale parallel simulation systems, including the following:

1. With a large number of LPs loaded on each available node, memory resources can quickly be exhausted due to the excessive amount of space required for saving past events and states. Consequently, the simulator is forced to reclaim the storage space of historical data with frequent GVT computation and fossil collection, an operation that itself is an important contributor to the overall operational overhead. Advanced algorithms such as *pruneback* [24], *cancelback* [25], and *artificial rollback* [26] attempt to recover from a memory stall, at the expense of additional computation and communication overhead. It is desired to have a protocol that can support large-scale optimistic simulation even when memory resources are tight, while at the same time reducing the overhead of GVT computation and fossil collection to a minimum (doing it only when absolutely necessary).
2. One potential performance hazard in large-scale optimistic simulation is that the cost of rollbacks increases dramatically, simply because a massive number of LPs are involved in the rollback operation on each node. Prolonged rollbacks not only result in poor system performance, but also increase the probability of *rollback echoes* [1], an unstable situation where simulation time does not progress. Therefore, it is imperative to fashion a

new approach that can dramatically reduce the rollback cost without introducing an additional runtime overhead.

3. Different implementations of the event sets in Time Warp have been the focus of research for several years (see, e.g., Refs [27–30]). A primary motivation behind these efforts is to improve the efficiency of queue operations as the number of stored events increases in large-scale and fine-grained simulations. In addition to using advanced data structures, the simulation performance would also be improved if we could keep the event queue relatively short throughout the simulation, an alternative approach that warrants close examination.
4. Dynamic load balancing has been recognized as a critical factor in achieving optimal performance in both general-purpose parallel applications and large-scale PDES systems where the workload on different nodes and the communication patterns between them are in constant fluctuation (see, e.g., Refs [31–34]). Algorithms for dynamic load balancing usually rely on metrics whose values are valid only for a short period. This problem is especially severe in optimistic simulations since a potentially unbounded number of uncommitted events and states associated with an LP need to be transferred to a different node before the invalidation of the metric values. Only a few studies address specifically the issue of facilitating load migration in Time Warp systems. For example, Reiher and Jefferson proposed a mechanism to split an LP into phases to reduce the amount of data that must be migrated [35]. More recently, Li and Tropper devised a method that allows for reconstructing events from the differences between adjacent states so that only the state queue needs to be transferred between processors [36]. However, this approach works only for systems with fine event granularity and small state size such as VLSI circuits. An agile load migration scheme is needed to reduce the communication overhead in Time Warp based large-scale parallel simulation systems.
5. The way simultaneous events are handled has serious implications on both simulation correctness and performance. To ensure correct simulation results, P-DEVS introduces (partial) causal dependency among simultaneous events, requiring a control flow to enforce an orderly event execution at each virtual time. From a performance perspective, however, this expanded execution of simultaneous events could increase the overhead for state-saving, rollback, and fossil collection, an issue that has not yet been addressed by existing TW-based DEVS systems.

To address these problems in a systematic way, a novel protocol, known as Lightweight Time Warp (LTW), has been proposed for high-performance optimistic simulations of large-scale DEVS and Cell-DEVS models [37]. LTW is able to set free the majority of the LPs from the heavy machinery of the Time Warp mechanism, while the overall simulation still runs optimistically, driven by only a few fully fledged Time Warp LPs. Although the following discussion is centered on realizing the LTW protocol in optimistic PCD++, the basic concepts also apply to a broad range of PDES systems under certain conditions and with appropriate control over the LPs.

The roles of the various LPs under LTW are as follows [37]:

1. The NC is the only LP that executes based on the standard Time Warp mechanism.
2. The FC becomes a mixed-mode LP that serves as an interface between full-fledged and lightweight LPs.
3. The Simulators are turned into lightweight LPs, free from the burdens of maintaining historical data in their input, output, and state queues.

9.5.1 A BRIEF REVIEW OF THE SIMULATION PROCESS IN OPTIMISTIC PCD++

As introduced in Section 9.3, on each node, the optimistic PCD++ simulator creates only one NC and FC, whereas many Simulators coexist in a typical large-scale simulation. Hence, a substantial reduction in the operational overhead at the Simulators would result in a significant improvement in the overall system performance. Note that even though the LPs are grouped together on each node, their LVT values may differ. Thus, the key characteristics of the simulation process can be summarized as follows.

1. Simulators only communicate with their parent FC (i.e., no direct communication between the Simulators). Thus, their states can change only as the result of executing events coming from the FC. The FC has full knowledge of the timing of state changes at its child Simulators.
2. Rollbacks on a node begin either at the NC (as a result of straggler or anti-messages arrived from other remote NCs), or at the FC (in the case of messaging anomalies). In both cases, rollbacks propagate from the FC to its child Simulators on a node. Thus, the FC knows when the rollbacks will occur at the Simulators. Besides, a WCTS is an atomic computation unit for the FC and Simulators during rollbacks.
3. The advance of the simulation time on each node is controlled entirely by the NC. The FC and the Simulators do not advance their LVTs voluntarily, nor do they send messages across WCTS boundaries.

Based on these assumptions, we now discuss LTW protocol main parts: a rule-based dual-queue event scheduling mechanism, an aggregated state-saving scheme, and a lightweight rollback mechanism (while the detailed algorithms can be found in Liu and Wainer [37]).

9.5.2 RULE-BASED DUAL-QUEUE EVENT SCHEDULING MECHANISM

Under Time Warp, the input queue is persistent, in the sense that the events remain in the queue until being fossil collected when the GVT advances beyond their time stamps. However, keeping this not only consumes lots of memory, but also increases the cost of queue operations. LTW solves this problem by introducing an additional volatile input queue that is used to hold, temporarily, the simultaneous events exchanged between the FC and its child Simulators within each phase of a WCTS.

On the contrary, the persistent input queue is used only by the NC and FC to contain the events sent between them. A key observation is that, during a rollback, the incorrect events previously exchanged between the FC and its child Simulators are essentially annihilated with each other (due to the atomicity of the WCTS). Therefore, it is safe to exclude these events from the persistent queue. An example message flow between the LPs mapped on a node under this dual-queue arrangement is illustrated in Figure 9.14.

From the Time Warp perspective, the simulation process only involves a small fraction of the total events executed by the LPs, as shown in Figure 9.15. Comparing both figures, we can see that the events executed by the FC and the Simulators within each phase of a WCTS can be considered as being collapsed into a single aggregated event. Note that the linking messages between adjacent WCTSs (e.g., x_{23} and $*_{24}$) are still kept in the persistent queue, allowing the simulation to resume forward execution after rollbacks.

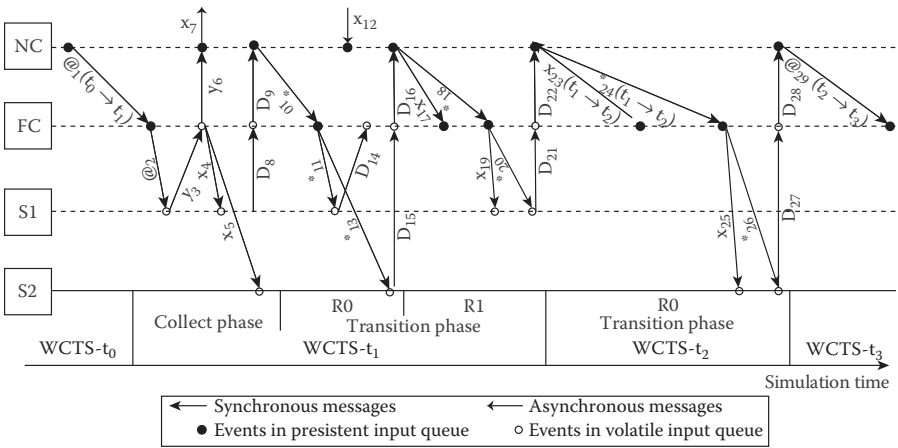


FIGURE 9.14 Message flow between the LPs using both persistent and volatile event queues.

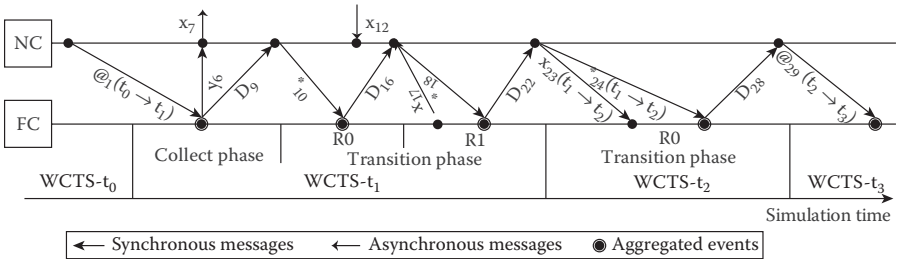


FIGURE 9.15 Message flow between the LPs from the time warp perspective.

The volatile input queue has two appealing properties that allow us to reduce the memory footprint and the cost of queue operations significantly:

1. Events in the queue are discarded and their memory reclaimed immediately after execution (reducing the memory footprint of the system).
2. Events in the queue always have the same time stamp. They are inserted into the queue as the simulation moves into each phase of a WCTS, and removed as the execution proceeds. At the end of each phase, the queue becomes empty. This means that a simple FIFO queue suffices, and queue operations can be performed efficiently in $O(1)$ time.

Consequently, the persistent queue becomes shorter, allowing for more efficient operations as well. For those events in the volatile queue, their counterpart anti-messages are no longer saved in the output queues of the sending LPs, further reducing memory consumption and speeding up the forward execution of the simulation. Similarly, message annihilations are not required to cancel these events during rollbacks, minimizing the rollback overhead and enhancing the stability and performance of the system. In addition, this also facilitates fossil collection due to the significant reduction in the number of past events and anti-messages stored in the persistent input and output queues, which, in turn, allows for more frequent GVT computation and fossil collection.

A rule-based event scheduling scheme has been proposed to schedule the execution of events from both queues [37]. This scheme is implemented on each node by a scheduler that maintains two pointers, called the *p-ptr* and *v-ptr*, to reference the next available events in the queues, respectively. The persistent queue contains events sorted in Lowest Time Stamp First (LTSF) order, including unprocessed events and those processed but have not yet been fossil collected. On the other hand, the volatile queue only holds simultaneous events not yet processed in the current phase of a WCTS, as illustrated in Figure 9.16.

The *p-ptr* pointer may need to be updated when the persistent queue is modified (event insertion and/or annihilation) to ensure that it always points to the first unprocessed event with the minimum time stamp, whereas the *v-ptr* pointer simply points to the event at the front of the volatile queue. At each event selection point, the scheduler compares the two events based on a set of predefined rules and chooses one of them as the next event to be executed in the current cycle, allowing the scheduler to adjust the priorities of the input queues dynamically on an

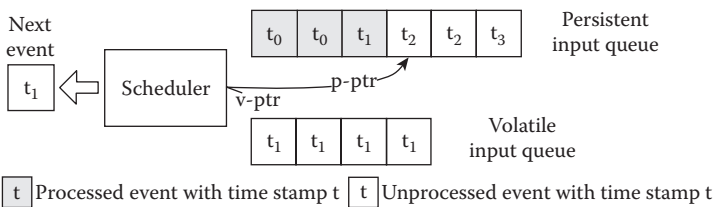


FIGURE 9.16 Rule-based dual-queue event scheduling.

event-by-event basis. The scheduling rules are executed in the order of appearance in the list.

1. **Idle condition.** The simulation becomes idle if the volatile queue is empty and the persistent queue does not contain events with time stamps before or at the simulation stop time. The simulation may be reactivated later upon the arrival of messages from other nodes.
2. **Simulation progress.** If the volatile queue becomes empty the scheduler selects the next persistent event with a time stamp earlier than the simulation stop time. The NC can: (1) advance simulation time on the node, or (2) resume forward execution from the unprocessed persistent events after a rollback, or (3) reactivate the simulation from the idle state upon the arrival of remote messages, which are inserted into the persistent queue.
3. **Aggressive internode communication.** During a collect phase, the NC may send messages to remote nodes. As these are potentially straggler messages at the receiving end, a delay in their delivery could postpone rollbacks at the destination, degrading performance. Thus, the scheduler grants a higher priority to the persistent events with the same time stamp in order to process internode messages immediately.
4. **LTSF execution.** In all other cases, the next volatile event is selected to execute, enforcing a Least-Time-Stamp-First execution on the node.

Note that an event selected from the volatile queue is removed (it will be deleted by the receiving LP after execution), whereas an event chosen from the persistent queue is simply marked as processed and the *p-ptr* is moved to the next available event.

9.5.3 AGGREGATED STATE MANAGEMENT AND AN OPTIMAL RISK-FREE STATE-SAVING STRATEGY

In a Time Warp system, each LP maintains its own state queue in order to restore to state variables during rollbacks. This approach allows for wide generality and straightforward implementation. However, it has several disadvantages. The historical states are scattered among the individual LPs, prohibiting efficient batch operations from being applied to the state queues. Also, state restorations are triggered entirely by straggler and/or anti-messages, putting stress on the underlying communication infrastructure. LTW allows the Simulators to delegate the responsibility of state management to the FC. As a result, the Simulators are turned into truly lightweight LPs, totally isolated from the complex data structures required by Time Warp.

In this new state management scheme, the FC employs an aggregated state manager that maintains the state queue for the FC itself, but also those used by the child Simulators. As illustrated in Figure 9.17, the state queue for a Simulator uses a *dirty bit* to identify the Simulators whose states have been modified during the computation of the current WCTS. The state-saving operation is carried out only when the FC detects that the events previously sent to the Simulators have been processed, and is performed only for those Simulators with dirty bits set to true. Note that no dirty bit

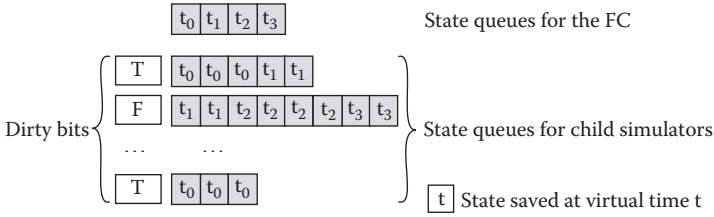


FIGURE 9.17 Aggregated state manager for the FC.

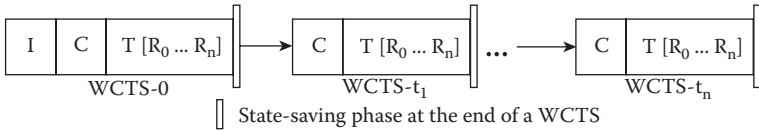


FIGURE 9.18 Introducing a state-saving phase at the end of each WCTS.

is associated with the state queue for the FC itself since the FC is always involved in the computation of each WCTS.

With this aggregated state management scheme, we now introduce an optimal risk-free state-saving strategy. As discussed in Section 9.3, the optimistic PCD++ simulator utilizes a MTSS strategy that allows it to save states only after executing certain types of events. Specifically, the NC and FC save states only after processing (D, t) messages, while the Simulators save state only after executing $(*, t)$ messages. Using the MTSS strategy, a transition phase may have multiple rounds of computation, thus, an LP could still save many states in each WCTS. During rollbacks, an optimal state-saving strategy should save only a single state at the end of each WCTS. The following strategy satisfies this condition and it is risk-free (no penalty is incurred as the result of saving fewer states).

As shown in Figure 9.18, a state-saving phase is added to the end of each WCTS. Before advancing the simulation time, the NC instructs the FC to save states for the current WCTS. At this moment, all events belonging to the current WCTS have been processed by the FC and the Simulators and, thus, the saved states will reflect the latest values of the state variables in the LPs. Only when the state-saving phase completes, can the NC send linking messages to the FC to initiate the next WCTS for the new simulation time. The state of the NC itself is saved after processing (D, t) messages, just like in the MTSS strategy.

9.5.4 LIGHTWEIGHT ROLLBACK MECHANISM

As the Simulators are turned into lightweight LPs, they are neither expected nor allowed to carry out rollbacks on their own in LTW. Instead, the Simulators must rely on the FC to recover from causality errors. As rollbacks always propagate from the FC to the Simulators on each node, the FC has the full knowledge of when to perform rollbacks for its child Simulators. One of the most elegant features of LTW is that the incorrect input events previously executed by the Simulators have already

been deleted in the volatile queue during forward execution. As a result, a great amount of CPU time that would otherwise be wasted on message annihilation can be saved, leading to an accelerated rollback, and significant improvement in performance overall.

The rollback of the FC itself is still triggered by straggler and/or anti-messages from the NC. The lightweight rollback mechanism must restore properly the state at the Simulators. One difficulty is that Simulators execute asynchronously and may not have the same LVT. Only the states of the Simulators involved in incorrect computations need to be restored during rollbacks. To solve this problem, the FC uses an array of *latest state change time* (LCT) to keep track of the time when the Simulators modify their states. The LCT is updated whenever the FC sends a $(*, t)$ to a Simulator. If a rollback occurs, the FC cancels speculative interactions with the NC (based on the standard Time Warp mechanism). It then invokes the scheduler to delete all volatile events with a time stamp greater than or equal to the rollback time. Finally, the FC instructs the aggregated state manager to recover the state for each Simulator whose LCT is greater than or equal to the rollback time. After the state restoration, the LCT is updated to the LVT of the recovered state.

In this way, LTW can perform rollbacks much more efficiently than the standard Time Warp mechanism due to the reduction of message annihilations in the persistent input queue. Moreover, all the Simulators can be rolled back without sending any anti-messages, reducing overhead.

9.5.5 LTW IMPLICATIONS

Though largely a local control protocol, LTW also has an impact on several aspects of the TW global control mechanism. First, fossil collection on each node is accelerated, not only because the fossil data in the persistent queues are minimized, but also because most of the states are managed in a centralized manner, allowing for efficient batch operations. Secondly, agile process migration is possible since only the state queues need to be transferred to move lightweight LPs for dynamic load balancing. The appropriate decision points for process migration would be at the end of each state-saving phase when all the volatile events have been executed (and deleted) and the states of the LPs have been saved.

Additionally, LTW can be seamlessly integrated with other TW optimizations to further improve performance. For instance, various state-saving and cancellation strategies can be applied to the TW domain directly. LTW can be considered as complementary to Local Time Warp [38]: the former is a purely optimistic approach to reducing operational cost within each local simulation space, while the latter is a locally optimistic approach to mitigating cascaded rollbacks in the global space. It is easy to combine both approaches in a consistent way.

Though only a single LTW domain is considered here, the protocol can be readily extended to support hybrid systems that require multiple LTW domains coexisting on each node to implement domain-specific formalisms. Besides, the basic concepts derived from the LTW protocol could also apply to a wide range of TW systems through carefully choosing the level of event granularity and imposing an appropriate control over the LPs.

9.6 PERFORMANCE ANALYSIS OF LTW

We studied the performance of PCD++ quantitatively. Our experiments were carried out on a cluster of HP PROLIANT DL servers that consists of 32 compute nodes (dual 3.2 GHz Intel Xeon processors, 1 GB PC2100 266 MHz DDR RAM) running Linux WS 2.4.21 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6.

The performance of optimistic PCD++ was tested with Cell-DEVS models with different characteristics, including a model for forest fire propagation [39] and a 3-D watershed model representing a hydrology system [40]. The fire propagation model computes the ratio of spread and intensity of fire in a forest based on specific environmental and vegetation conditions. The watershed model represents the water flow and accumulation depending on the characteristic vegetation, surface waters, soil, ground water, and bedrock. The watershed model was coded as a 3-D Cell-DEVS model to simulate the accumulation of water under the presence of constant rain.

Conservative PCD++ was analyzed by running the above mentioned fire model as well as another CELL-DEVS model on Synapsin-Vesicle Reaction [41]. We have modeled the reserve pool of synaptic vesicles in a presynaptic nerve terminal, predicting the number of synaptic vesicles released from the reserve pool as a function of time under the influence of action potentials at differing frequencies. Time series amounts for the components are obtained using rule-based methods [42]. Creating this model in PCD++ allows a spatial description of synapsin-vesicle interactions and PCD++ makes it possible to have graphical representations that are comparable to the real scene observed in electron microscopes.

Performance metrics were collected during the simulations to gauge the performance and profile of the simulation system. These metrics fall into two categories based on their intended purposes: namely, performance measurement and system profiling. The former group includes three values collected to measure execution time, memory consumption, and CPU utilization. The latter group consists of a variety of values at runtime to profile the simulation system, including the number of states saved during the simulation, the time spent on state-saving operations, and the bootstrap time for simulation initialization. We use two different speedups measuring the overall speedup to reflect how much faster the simulation runs on multiple machines than it does on a single one (as perceived by the users), whereas the algorithm speedup is calculated from the actual running time (i.e., without considering the simulation bootstrap time, to assess the performance of the parallel algorithms alone).

A simple partition strategy was adopted for all the models in the following tests. It evenly divides the cell space into horizontal rectangles, as illustrated in Figure 9.19 for a 30×30 model partitioned over three nodes. Using different partition strategies could have a big impact on the performance of the simulation. Since the workload on the nodes is unpredictable and keeps changing during the simulation, it is hard, if not impossible, to predict the best partition strategy for a given model before the simulation. This problem can be alleviated by using some dynamic load-balancing techniques in the simulation algorithms.

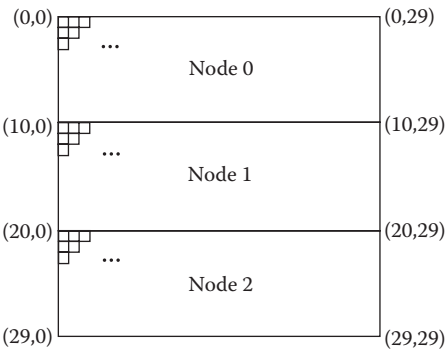


FIGURE 9.19 A simple partition strategy for cell-DEVS models.

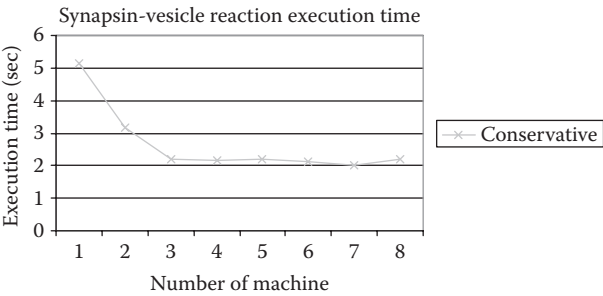


FIGURE 9.20 Synapsin-vesicle model execution time on conservative parallel simulator.

9.6.1 PERFORMANCE ANALYSIS OF THE CONSERVATIVE PCD++

The *Synapsin-Vesicle Reaction* model consists of 676 cells arranged in a 26×26 mesh with a total execution time of 3.7621 seconds when run on the stand-alone single-processor CD++. Figure 9.20 represents the execution time resulting from running the model with the conservative parallel simulator on 1–8 nodes.

As seen on the graph, the conservative simulator improves the total execution time significantly when more than two nodes are available. As the number of participating nodes increases, the speed up factor decreases. The main reason is communication overhead among the participating LPs that leads to a noticeable time added to the duration of the model execution.

The *Fire Propagation* model consists of 900 cells arranged in a 30×30 mesh with a total execution time of 6.2145 seconds when run on a stand-alone CD++. Figure 9.21 represents the execution time resulting from running the model with the conservative simulator on 1–8 nodes.

As seen on the graph, the parallel simulator improved the execution time of the model. The conservative simulator presented the best results on four nodes, while still all other scenarios with more than one node have lower execution results with respect to simulation on a single node.

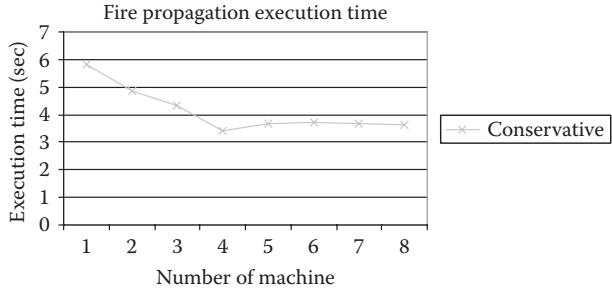


FIGURE 9.21 Fire propagation model execution time with the conservative parallel simulator.

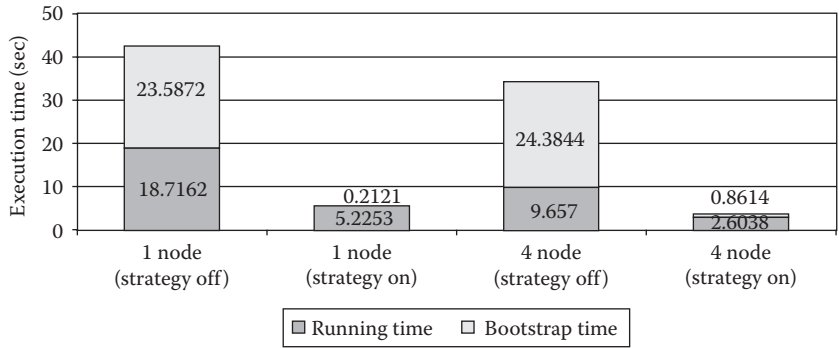


FIGURE 9.22 Execution and bootstrap time before and after one log file per node strategy on 1 and 4 nodes.

9.6.2 PERFORMANCE ANALYSIS OF OPTIMISTIC PCD++

In this section, we evaluate the performance of optimistic PCD++ under the standard Time Warp protocol and the PCD++ optimization strategies discussed in Section 9.3.

9.6.2.1 Effect of the One Log File per Node Strategy

The fire propagation model (900 cells arranged in a 30×30 mesh) was executed on 1 and 4 nodes with and without using the strategy during a period of 5 hours. The resulting execution time (T) and bootstrap time (BT) are illustrated in Figure 9.22, where the BT for four nodes is the arithmetic average of the BT values collected on these nodes.

As we can see, BT is greater than the actual running time (when the strategy is turned off), clearly indicating that the bootstrap operation is a bottleneck. When the strategy is turned on, the bootstrap time is reduced by 99.1% on one node and by 96.47% on four nodes. Further, the running time is decreased by 72.08% on one node and by 73.02% on four nodes due to more efficient communication, I/O, and rollback operations.

The CPU usage (%CPU) monitored in our experiments also suggests that the file I/O operation is a major barrier in the bootstrap phase. The CPU usage collected

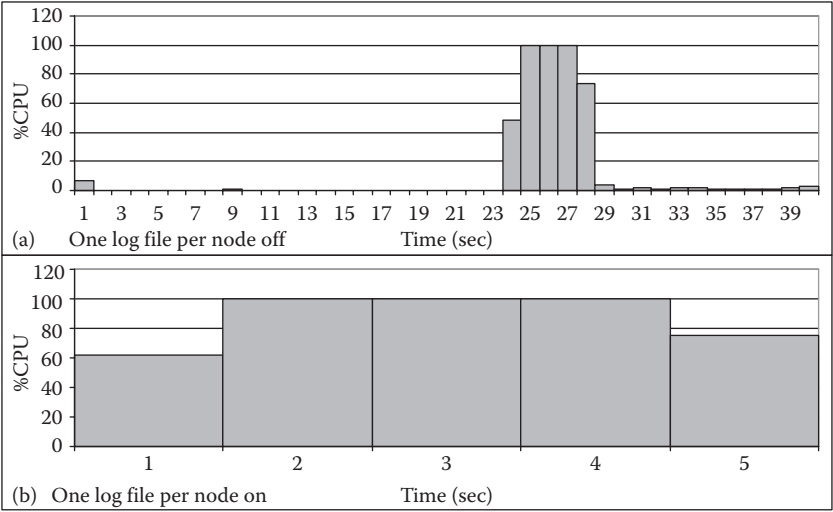


FIGURE 9.23 CPU usage before and after one log file per node strategy on one node.

before and after applying the one log file per node strategy on a single node is shown in Figure 9.23. Again, when the strategy is turned off, the CPU essentially remains idle in the first 23 seconds (corresponding to the observed BT), during which a majority of time has been dedicated to I/O operations for creating the log files at the NFS server over the network. At the end of the simulation, the logged data is flushed to the physical files, resulting in intensive I/O operations again. As expected, the CPU rests idle in the last 12 or so seconds. On the other hand, the computation is condensed when the strategy is applied to the simulator. As a result, the CPU is utilized much more efficiently with the one log file per node strategy. A similar pattern was observed in simulations running on multiple nodes.

Other observations are described below:

1. The bootstrap time tends to increase when more nodes are used to do the simulation. For example, the BT increased from 0.2121 seconds on one node to 0.8614 seconds on four nodes in our experiments. The reason is that the number of log files increases with the number of nodes, causing higher delays in communication and file I/O operations at the NFS server.
2. The bootstrap time also tends to increase somewhat along with the size of the model because of the additional operations for memory allocation and object initialization in the main memory. However, this is a relatively moderate increase when compared with the previous case.
3. Even though the bootstrap time is reduced significantly with the one log file per node strategy, it still constitutes an overhead that cannot be ignored when we measure the real effect of the parallel algorithms. In the experiments, it accounts for 3.9% and even 24.84% of the total execution time on 1 and 4 nodes, respectively.

9.6.2.2 Effect of the Message Type-Based State-Saving Strategy

The fire propagation model was used to evaluate the performance improvement derived from the MTSS strategy. Besides the standard Time Warp algorithms, the one log file per node strategy is also applied to the simulator in the following experiments.

The model was executed on 1 and 4 nodes (respectively) with and without the MTSS strategy (respectively). The number of states saved in the simulation (SS) and the time spent on state-saving operations (ST) are shown in Figure 9.24. Here, the data for four nodes is the average of the corresponding values collected on the nodes. Owing to the MTSS strategy, the number of states saved during the simulation is reduced by 49.29% and 47.74% on 1 and 4 nodes, respectively. Accordingly, the time spent on state-saving operations is decreased by 29.9% and 27.76%.

The corresponding running time and bootstrap times are shown in Figure 9.25. While the bootstrap time remains nearly unchanged, the actual running time is reduced by 17.64% and 7.63% on 1 and 4 nodes, respectively, because fewer states are saved and potentially removed from the queues.

The most noticeable effect of the MTSS strategy is the decrease in memory consumption. Figure 9.26 shows the time-weighted average and maximum memory consumption with and without the strategy for the fire propagation model on 1 and

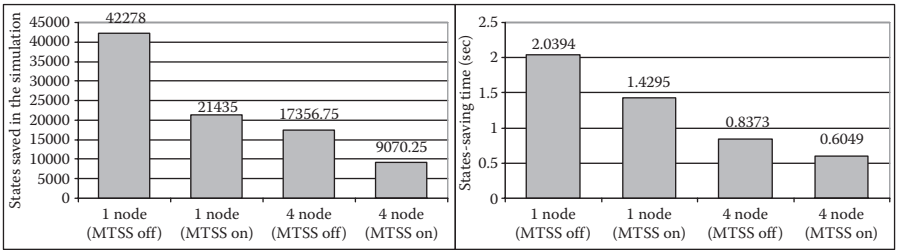


FIGURE 9.24 States saved and state-saving time before and after MTSS strategy on 1 and 4 nodes.

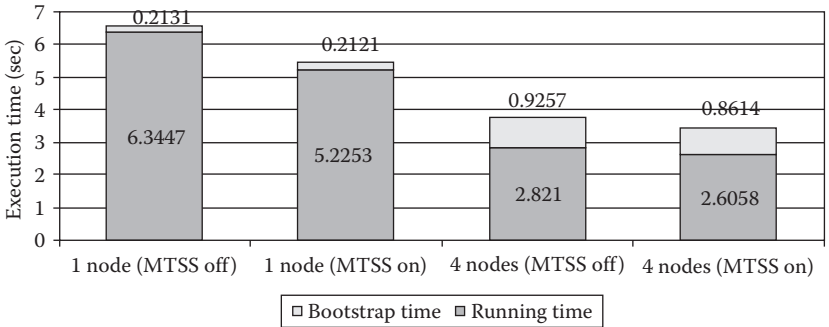


FIGURE 9.25 Running and bootstrap time before and after MTSS strategy on 1 and 4 nodes.

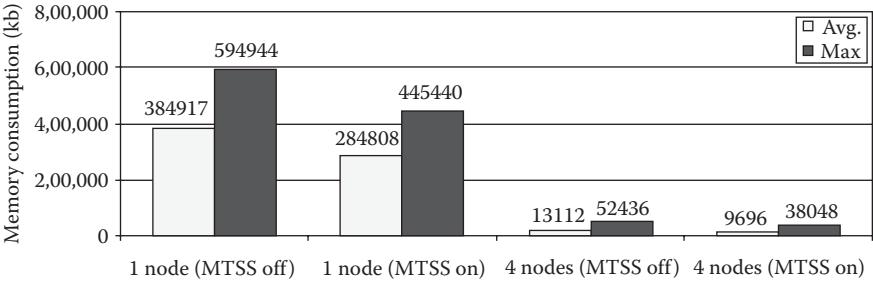


FIGURE 9.26 Average and maximum memory consumption before and after MTSS strategy.

four nodes. The time-weighted average was calculated using an interval of 1 second. For four nodes, the data was also averaged over the nodes. The average memory consumption declines by 26% in both cases, while the peak memory consumption decreases by 25.13% and 27.44% on 1 and 4 nodes, respectively.

9.6.2.3 Performance of the Optimistic PCD++

The metrics used for evaluating the performance of the optimistic PCD++ simulator included the execution time and speedup. We analyze the execution results of the Cell-DEVS models with the standard Time Warp algorithms. In addition, the one log file per node and MTSS strategies were applied to the simulator in the experiments.

We compare the performance of conservative and optimistic PCD++ using the fire propagation model executed on a set of compute nodes. Figure 9.27 shows the total execution time for the fire propagation simulation with different sizes, including 20×20 (400 cells), 25×25 (625 cells), 30×30 (900 cells), and 35×35 (1225 cells).

In all cases, the optimistic simulator markedly outperforms the conservative one. There are three major contributing factors:

1. Optimistic PCD++ has been optimized with the one log file per node strategy. Hence, its bootstrap time is substantially less than that of the conservative. Although the data logged during the simulations is the same for both simulators, the number of log files generated by PCD++ is only a small fraction of that created by the conservative simulator. This factor accounts for much of the difference in the execution time on a single node.
2. Time Warp avoids, for the most part, the serialization of execution that is inherent in the conservative algorithms, and hence, exploit a higher degree of concurrency.
3. The nonhierarchical approach adopted in optimistic PCD++ outperforms the hierarchical one of the conservative simulator. The flattened structure reduces the communication overhead and allows more efficient message exchanges between the PCD++ processors.

The total execution time and running time of the fire model executed on 1 up to 4 nodes are listed in Table 9.1.

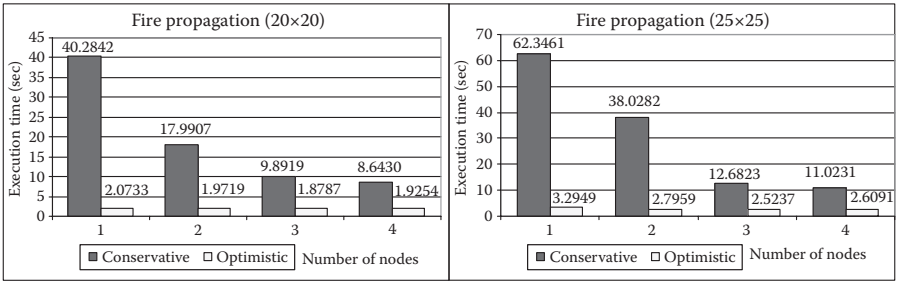


FIGURE 9.27 A comparison of the conservative and optimistic PCD++ using the fire model.

TABLE 9.1
Execution Time and Running Time for Fire Model of Various Sizes on a Set of Nodes

Total Execution Time (sec)					Running Time (sec)				
Number of nodes	20 × 20	25 × 25	30 × 30	35 × 35	Number of Nodes	20 × 20	25 × 25	30 × 30	35 × 35
1	2.0733	3.2949	5.0442	7.8702	1	1.9515	3.1273	4.3566	7.6428
2	1.9719	2.7959	3.5232	4.7138	2	1.4232	2.1225	2.8838	3.9952
3	1.8787	2.5237	3.1573	3.9667	3	1.3574	1.8953	2.5237	3.2959
4	1.9254	2.6091	3.0922	3.8138	4	1.4296	1.8656	2.3314	3.0224

For any given number of nodes, the execution time always increases with the size of the model. Moreover, the execution time rises less steeply when more nodes are used to do the simulation. For example, as the model size increases from 400 to 1225 cells, the execution time ascends sharply by nearly 280% (from 2.0733 to 7.8702 seconds) on one node, whereas it merely rises by 98% (from 1.9254 to 3.8138 seconds) on four nodes. On the other hand, for a fixed model size, the execution time tends to, but not always, decrease when more nodes are utilized. The execution time for the 20×20 model decreases from 2.0733 to 1.8787 seconds when the number of nodes climbs from 1 to 3. However, when the number of nodes increases further, the downward trend in execution time is reversed. It increases slightly from 1.8787 to 1.9254 seconds as the number of nodes rises from 3 to 4. When a model, especially a small one, is partitioned onto more nodes, the increasing overhead involved in inter-LP communication and potential rollbacks may eventually degrade performance. Hence, a trade-off between the benefits of a higher degree of parallelism and the concomitant overhead costs needs to be reached when we consider different partition strategies.

From the table, we can also find that better performance can be achieved on a larger number of nodes as the model size increases. The shortest execution time is achieved on three nodes for the 20×20 and 25×25 models, while it is obtained on

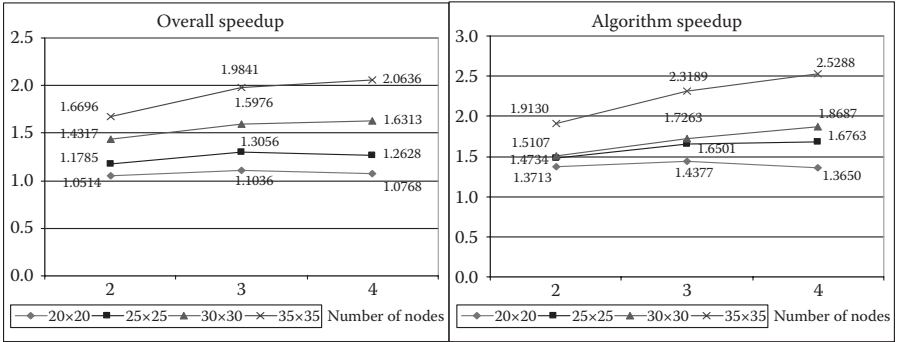


FIGURE 9.28 Overall and algorithm speedups for fire model of various sizes on a set of nodes.

TABLE 9.2
Execution Time and Running Time for the $15 \times 15 \times 2$ Watershed Model on a Set of Nodes

Number of nodes	1	2	3	4	5
Total execution time (sec)	16.8036	11.7930	8.3285	7.3205	6.1538
Running time (sec)	16.6668	11.1522	7.7191	6.8140	5.6743

four nodes for the other two larger models. It is clear that we should use more nodes to simulate larger and more complex models where intensive computation is the dominant factor in determining the system performance.

Based on the collected execution and running time, the overall and algorithm speedups are shown in Figure 9.28. As we can see, higher speedups can be obtained with larger models. In addition, the algorithm speedup is always higher than its counterpart overall speedup, an evidence showing that the Time Warp optimistic algorithms are major contributors to the overall performance improvement.

A more computation-intensive 3-D watershed model of size $15 \times 15 \times 2$ (450 cells) was tested to evaluate the performance of optimistic PCD++ for simulating complex physical systems. Table 9.2 shows the total execution time and running time. The best performance is achieved on 5 nodes with execution and running time of 6.1538 and 5.6743 seconds, respectively.

The resulting speedup is illustrated in Figure 9.29. The best overall and algorithm speedups achieved for the $15 \times 15 \times 2$ watershed model are 2.7306 and 2.9373, respectively, higher than those obtained with the fire model.

9.6.3 EFFECT OF HYBRID OPTIMISTIC ALGORITHMS

To analyze the impact of the LRFM and GRFM Time Warp-based protocols on the performance of the optimistic PCD++ simulator, the same models were also run using LRFM- and GRFM-based optimistic simulators. Different models with

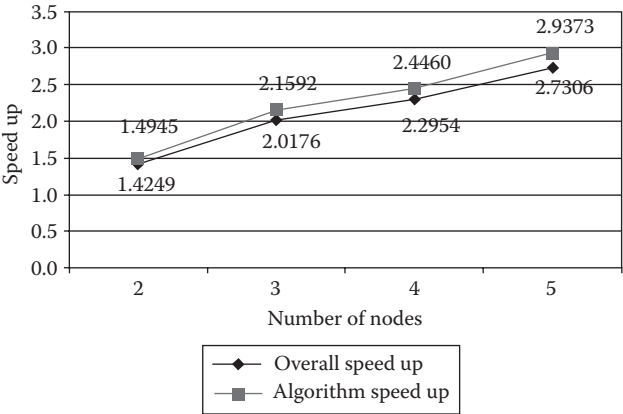


FIGURE 9.29 Overall and algorithm speedups for the $15 \times 15 \times 2$ watershed model.

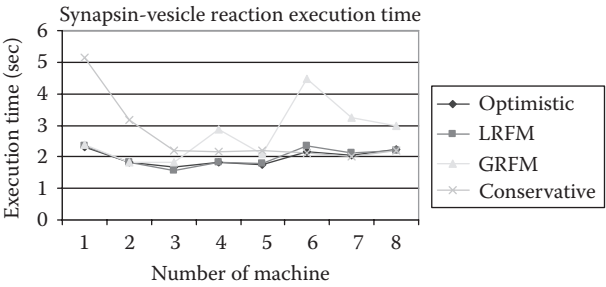


FIGURE 9.30 Synapsin-vesicle model execution time on 4 different simulators.

distinguishable functionality, complexity, and size were selected for the experiments to better judge the capability of the simulators. The *Synapsin-Vesicle Reaction* model presented in Section 9.6.1 is presented in Figure 9.30 that represents the execution time resulting from running the model with four different simulators on 1–8 nodes.

We can see that the optimistic and LRFM-based simulators produce very close results on 1–8 nodes. Also, the GRFM-based simulator has similar results for 1, 2, 3, and 5 nodes. However, the performance is degraded when 4, 6, 7, and 8 nodes are participating due to the increase of number of remote messages that are sent back and forth. On the other hand, the conservative simulator shows a different behavior as the number of nodes increases. As the number of computing nodes increases, the GRFM-based simulator has the lowest performance among the other ones. The main reason is communication overhead among the participating LPs that leads to a noticeable time added to the duration of the model execution.

Figure 9.31 represents the speedups of the model execution times with respect to execution on one node for each particular simulator.

9.6.3.1 Fire Propagation Model

This model uses 900 cells arranged in a 30×30 mesh with a total execution time of 6.2145 seconds when run on a stand-alone CD++. Figure 9.32 represents the

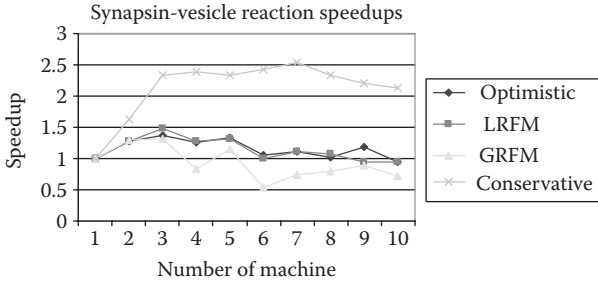


FIGURE 9.31 Synapsin-vesicle model speedups with regard to execution on one node.

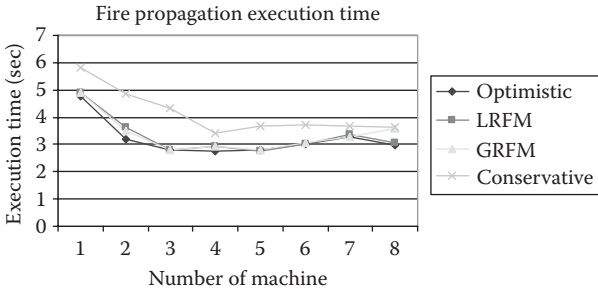


FIGURE 9.32 Fire propagation model execution time on four different simulators.

execution time resulting from running the model with four different simulators on 1–8 nodes. As seen on the graph, the parallel simulator significantly improved the execution time of the simulation. The three optimistic simulators produced very similar results on 1–7 nodes and the optimistic simulators outperforms the conservative one. For the optimistic simulators the best results were achieved on five nodes, while the conservative one had its lowest execution time on four nodes since the number of remote messages and the suspension duration of the simulation objects were at their best on four nodes.

9.6.4 COMPARING TIME WARP AND LIGHTWEIGHT TIME WARP

As analyzed in Section 9.5, LTW can effectively improve system performance in various ways (including reduced memory consumption, lowered operational overhead, and accelerated queue operations). In this section, we present a comparison of the optimistic PCD++ performance under the standard Time Warp and LTW protocols.

9.6.4.1 Experiment Platform and Metrics

Both TW and LTW protocols have been implemented in optimistic PCD++. A stress test was carried out on a cluster of 28 HP Proliant DL140 servers (dual 3.2 GHz Intel Xeon processors, 1 GB 266 MHz RAM with 2 GB disk swap space) running on Linux WS 2.4.21 and communicating over Gigabit Ethernet using MPICH 1.2.7. Note that severe memory swapping may occur if memory usage approaches the

TABLE 9.3
Performance Metrics

Metrics	Description
T	Total execution time of the simulation (sec)
MEM	Maximum memory consumption (MB)
PEE	Number of events executed in persistent queue
VEE	Number of events executed in volatile queue
PQLen	Average length of the persistent input queue
VQLen	Average length of the volatile input queue
SS	Total number of states saved
OPT-SK	Number of states reduced by the optimal strategy
FCT	Average time spent on a single fossil collection (ms)
PriRB	Number of primary rollbacks
SecRB	Number of secondary rollbacks
RB	Total number of rollbacks (i.e., PriRB + SecRB)
EI	Number of events imploded in persistent queue
ER	Number of events unprocessed in persistent queue

upper limit of 1 GB on a node. Table 9.3 lists the metrics collected in the experiments through extensive instrumentation and measurement. The experimental results for each test case were averaged over 10 independent runs to strike a balance between data reliability and testing effort. For those cases executed on multiple nodes, the results were averaged over the participating nodes to obtain a per-node evaluation. The queue lengths (i.e., PQLen and VQLen) were averaged over samples collected every 20 event insertions.

9.6.4.2 Environmental Models

Three Cell-DEVS models with varied characteristics were validated and tested. Two of them simulate stationary wildfire propagation over 50 hours in a 2-D cell space based on the Rothermel model. However, they differ in the way the spread rates are calculated. The first fire model, referred to as *Fire1*, uses predetermined rates at reduced runtime computation cost. The second fire model, referred to as *Fire2*, invokes the `fireLib` library [44] to calculate spread rates dynamically, with higher runtime computation density, based on a set of parameters such as fuel type, moisture, wind direction, and speed. The time for executing a $(*, t)$ message at the Simulators, which reflects the computation intensity of the state transitions, was calibrated at 112 and 748 μ s for *Fire1* and *Fire2*, respectively.

The other model, called the *Watershed*, simulates the environmental influence on hydrological dynamics of water accumulation over 30 minutes in a 3-D cell space. Though it is not as compute-intensive as *Fire2* (577 μ s state transition time), a larger neighborhood of 10 cells on different layers of the cell space is defined with increased communication intensity.

Unlike cellular automata models, which evaluate all the cells synchronously at discrete time steps, these Cell-DEVS models define the cell spaces as discrete-event

models where each cell is an independent atomic model executed by a Simulator allowing efficient asynchronous execution without losing accuracy. These Cell-DEVS models execute a great number of simultaneous events at each virtual time, increasing the operational cost of TW simulation. In the next section, we will show that LTW is well-suited for improving simulation performance in such situations.

9.6.4.3 Test Results and Analysis

The comparative evaluation was conducted under the same configurations. Both protocols used aggressive cancellation and CSS optimized with MTSS. In addition, the optimal risk-free state-saving strategy introduced in Section 9.5 was enabled for LTW. In all test cases, message logging activities were turned off to minimize the impact of file I/O operations on system performance. Also, the corresponding test cases used the same partition scheme to divide the cell space (horizontal rectangles evenly distributed among nodes).

In the following tables, a “×” mark indicates a failed test case due to memory exhaustion, while a shaded entry attributes the poor performance to severe memory swapping activities. A “–” mark stands for a case that was not tested because either the performance trend is already clear in the series, or the model cannot be divided further with the given partition scheme. The best execution time is highlighted in each series. The results (T and MEM) of a sequential simulator are also provided as a reference for evaluating the absolute performance of both protocols.

Table 9.4 gives the resulting total execution time and maximum memory usage (T and MEM) for *FireI* of varied sizes on a different number of nodes. It is clearly shown that an LTW outperforms a TW counterpart in all successful cases. First, the maximum memory usage on each node is reduced by 45% up to 92%, making it possible to execute the model using a smaller number of nodes, with significantly lower simulation cost. Secondly, the total execution time is decreased by 24% up to 60% among those test cases with sufficient memory, and this improvement is achieved with a much smaller memory footprint at the same time.

To find out the reason that causes the differences, other metrics were compared. As an example, in Table 9.5 we present a comparison of the 100×100 *FireI* on 14 nodes. The introduction of the volatile input queue reduces the average length of the persistent input queue by 89.37%, reducing the overhead of queue operations and memory consumption. On the other hand, the volatile queue is kept short throughout the simulation (average length of 121.89 events), despite the fact that 86.38% input events executed on each node have been turned into volatiles under the LTW.

Owing to the optimal risk-free state-saving strategy (which reduces the number of state-saving by 44.86% on top of MTSS), the total number of states saved in LTW is 57.07% fewer than in TW, resulting in less memory usage as well. As expected, the time for each fossil collection decreased from 488.14 to 84.15 ms (a reduction of 82.76%).

When comparing rollback performance, an LTW shows a big advantage over a TW. The number of secondary rollbacks is reduced by 91.77%, showing that rollback propagation is effectively contained within the TW domain on each node. Moreover,

TABLE 9.4
Total Execution Time and Maximum Memory Usage for *Fire1*

Size	Sequential	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
50 × 50	5.54 (T) 29.11 (MEM)	TW	T	×	9.08	5.87	5.26	5.01	5.39	5.49	5.55	5.95	–	–	–	–	–	–
			MEM	×	813.57	220.42	109.94	61.79	43.73	34.84	26.37	22.22	–	–	–	–	–	–
		LTW	T	5.78	3.61	3.02	2.98	2.78	3.01	3.23	3.25	3.54	–	–	–	–	–	–
			MEM	63.53	65.83	27.42	20.58	14.25	13.24	11.98	9.95	9.31	–	–	–	–	–	–
100 × 100	56.07 (T) 110.59 (MEM)	TW	T	×	×	2749.13	484.91	40.09	35.66	34.46	32.35	33.51	32.53	32.44	33.4	35.0	35.19	35.96
			MEM	×	×	2279.42	1492.31	882.82	576.61	410.19	307.79	244.6	197.97	162.92	137.77	121.47	103.03	91.75
		LTW	T	78.21	43.84	31.62	24.35	23.58	22.61	22.26	21.62	21.86	21.88	22.03	22.2	22.0	22.46	21.76
			MEM	405.5	373.25	271.62	160.26	110.94	82.65	66.75	55.65	48.18	43.55	38.92	36.22	34.05	32.3	29.94
150 × 150	260.65 (T) 242.69 (MEM)	TW	T	×	×	×	×	×	1516.48	893.43	572.83	314.03	202.71	141.46	140.98	142.63	142.01	143.18
			MEM	×	×	×	×	×	2309.12	1935.02	1449.83	1131.65	906.07	744.91	623.9	527.05	460.76	404.44
		LTW	T	1489.77	517.92	394.56	122.44	112.93	110.63	111.7	109.67	107.02	107.23	105.27	107.1	106.75	104.88	104.74
			MEM	1418.85	1294.08	986.62	660.31	415.01	296.96	230.4	186.68	161.7	137.22	123.85	105.07	96.8	90.88	85.09
200 × 200	815.43 (T) 432.13 (MEM)	TW	T	×	×	×	×	×	×	×	×	×	×	4324.31	1236.26	1065.79	881.61	737.14
			MEM	×	×	×	×	×	×	×	×	×	×	1848.93	1560.7	1528.73	1188.06	1058.7
		LTW	T	12571.7	6894.36	1425.16	920.86	646.56	350.58	334.77	331.2	333.12	326.7	327.56	327.46	322.93	330.03	327.24
			MEM	1679.36	1644.54	1393.66	1229.82	1145.6	805.17	582.49	431.18	393.15	291.49	244.01	209.52	235.47	186.1	188.68

TABLE 9.5
100 × 100 *Fire1* on 14 Nodes

Metrics	TW	LTW	LTW vs. TW
PEE	96,685.07	10,597.71	
VEE	0	67,214.07	
PQLen	24,798.12	2636.95	↓ 89.37%
VQLen	0	121.89	
SS	52,819.64	22,675.14	↓ 57.07%
OPT-SK	0	18,445.36	
FCT	488.14	84.15	↓ 82.76%
PriRB	613.14	604.00	↓ 1.49%
SecRB	11,922.07	981.14	↓ 91.77%
RB	12,535.21	1585.14	↓ 87.35%
EI	61,751.93	5826.36	↓ 90.56%
ER	48,118.79	5790.93	↓ 87.97%

the number of primary rollbacks is reduced by 1.49%. This combined with the fact that the total number of events executed on each node (i.e., PPE + VEE) decreased by 19.52%, suggests a more stable system with less speculative computation. Consequently, the numbers of events imploded and unprocessed in the persistent queue also declined around 90%, further accelerating the rollback operations.

The experimental results for the *Fire2* and *Watershed* models are shown in Tables 9.6 and 9.7, respectively. Again, an LTW reduces maximum memory consumption by approximately 34% up to 92% for *Fire2* and by 73% up to 93% for *Watershed*. The reduction in memory usage is more prominent for *Watershed* largely because, with a higher number of simultaneous events exchanged between the LPs at each virtual time, a larger percentage of states are reduced with the optimal state-saving strategy.

For those cases with sufficient memory, the total execution time decreased by 13% up to 32% for *Fire2* and by 5% up to 91% for *Watershed*. A general trend reflected in the experimental results is that the reduction in execution time and memory usage is greater for models with larger sizes, indicating an improved scalability.

Other metric values for the 100 × 100 *Fire2* on 20 nodes and 20 × 20 × 2 *Watershed* on 18 nodes are given in Tables 9.8 and 9.9, respectively. As we can see, a similar pattern can be observed regarding the improvement of the metrics, suggesting that an LTW is suitable for simulating models with varied computation and communication characteristics.

In terms of absolute performance, LTWs attain higher and more consistent speedup than TWs. In some scenarios, the performance of a TW simulation is even worse than the sequential execution (e.g., 50 × 50 *Fire1* on 2 and 4 nodes; 20 × 20 × 2 *Watershed* on 14, 16, and 18 nodes), mainly due to the excessive communication and operational overhead. However, such scenarios do not arise in the LTW cases tested.

TABLE 9.6
Total Execution Time and Maximum Memory Usage for *Fire2*

Size	Sequential	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
50 × 50	19.29 (T) 29.52 (MEM)	TW	T	×	20.89	13.93	12.19	10.91	10.41	10.8	10.64	10.84	10.55	11.31	12.51	12.76	13.39	13.44
			MEM	×	800.26	226.82	108.41	65.37	46.29	34.54	28.13	23.23	20.19	18.19	16.31	14.81	13.72	12.85
		LTW	T	20.26	14.23	10.38	9.69	9.46	8.84	9.01	8.51	8.64	8.4	8.32	9.28	9.34	9.51	10.27
			MEM	81.24	66.92	34.99	22.6	17.77	14.65	13.02	11.76	10.83	10.17	9.63	9.29	8.99	8.73	8.49
100 × 100	119.95 (T) 109.57 (MEM)	TW	T	×	×	3284.37	460.32	68.67	54.63	52.03	48.92	48.58	46.96	46.37	47.53	48.69	49.39	49.97
			MEM	×	×	2159.1	1319.08	658.14	576.72	411.14	310.95	240.42	198.4	163.47	149.65	112.23	99.94	83.78
		LTW	T	206.16	114.98	60.09	54.37	51.22	44.11	41.61	40.37	38.87	37.55	35.54	36.83	36.23	36.46	36.48
			MEM	314.37	285.18	248.32	137.73	102.24	81.63	65.57	54.35	48.91	45.62	42.6	38.42	35.75	33.84	32.03
150 × 150	414.25 (T) 243.71 (MEM)	TW	T	×	×	×	×	×	4448.08	2487.95	651.06	394.92	244.97	167.25	164.79	167.42	165.64	168.88
			MEM	×	×	×	×	×	1817.71	1375.23	1399.3	1086.72	905.96	744.91	562.55	532.14	425.91	399.51
		LTW	T	1592.43	493.61	223.65	178.2	174.63	165.84	168.66	167.14	140.67	140.21	137.0	134.3	136.11	133.1	134.01
			MEM	1210.37	924.16	641.79	586.92	385.41	269.62	205.4	172.18	139.44	122.16	112.93	104.22	94.47	89.39	85.79
200 × 200	1033.61 (T) 424.96 (MEM)	TW	T	×	×	×	×	×	×	×	×	×	12112.7	3206.02	1501.28	1202.48	900.05	764.21
			MEM	×	×	×	×	×	×	×	×	×	1943.55	1785.9	1618.94	1522.69	1475.58	1243.95
		LTW	T	11707.5	3363.07	1339.92	1173.69	562.68	414.52	412.92	412.89	381.1	376.58	417.44	373.11	372.6	370.04	371.56
			MEM	1661.95	1562.62	1267.71	1292.97	885.61	438.81	363.5	313.96	289.68	274.55	240.98	227.23	208.62	192.61	173.08

TABLE 9.7

Total Execution Time and Maximum Memory Usage for *Watershed*

Size	Sequential	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
15 × 15 × 2	258.27 (T)	TW	T	×	×	2059.62	899.49	84.97	87.06	86.59	88.76	–	–	–	–	–	–	–
			MEM	×	×	1718.02	997.21	691.2	536.37	422.49	333.53	–	–	–	–	–	–	–
	43.99 (MEM)	LTW	T	262.99	171.18	112.69	100.54	79.45	82.27	82.08	82.59	–	–	–	–	–	–	–
			MEM	45.66	27.91	148.48	121.54	128.96	113.14	101.29	90.39	–	–	–	–	–	–	–
			T	×	×	×	×	2451.7	857.3	757.65	724.55	638.97	676.42	–	–	–	–	–
			MEM	×	×	×	×	1618.94	1180.67	967.51	778.53	643.52	535.21	–	–	–	–	–
20 × 20 × 2	471.86 (T)	TW	T	×	×	×	×	2451.7	857.3	757.65	724.55	638.97	676.42	–	–	–	–	–
			MEM	×	×	×	×	1618.94	1180.67	967.51	778.53	643.52	535.21	–	–	–	–	–
	72.67 (MEM)	LTW	T	473.81	268.87	181.94	155.09	140.14	104.77	108.52	109.58	110.35	112.87	–	–	–	–	–
			MEM	76.02	40.04	164.35	136.36	130.82	149.81	137.24	129.85	115.87	111.99	–	–	–	–	–
			T	×	×	×	×	×	×	×	2002.73	1948.95	1922.21	1705.19	1597.08	1585.6	–	–
			MEM	×	×	×	×	×	×	×	1519.54	1434.77	1262.59	1063.03	774.38	663.21	–	–
25 × 25 × 2	735.39 (T)	TW	T	×	×	×	×	×	×	×	2002.73	1948.95	1922.21	1705.19	1597.08	1585.6	–	–
			MEM	×	×	×	×	×	×	×	1519.54	1434.77	1262.59	1063.03	774.38	663.21	–	–
	115.48 (MEM)	LTW	T	748.49	469.65	306.25	257.18	195.16	176.19	172.39	136.18	136.37	142.69	143.86	139.54	141.85	–	–
			MEM	119.8	70.46	164.86	128.68	131.07	132.81	132.27	153.82	141.87	128.25	114.39	113.95	103.44	–	–
			T	×	×	×	×	×	×	×	×	5381.55	4475.37	3133.72	3130.89	2920.06	2765.2	2784.83
			MEM	×	×	×	×	×	×	×	×	2192.96	1867.83	1602.25	1388.87	1206.87	1055.31	924.49
30 × 30 × 2	1041.39 (T)	TW	T	×	×	×	×	×	×	×	×	5381.55	4475.37	3133.72	3130.89	2920.06	2765.2	2784.83
			MEM	×	×	×	×	×	×	×	×	2192.96	1867.83	1602.25	1388.87	1206.87	1055.31	924.49
	168.46 (MEM)	LTW	T	1098.11	616.28	390.68	293.33	237.82	208.26	204.82	198.27	169.12	168.45	168.01	165.54	165.64	166.55	162.43
			MEM	174.08	89.69	163.07	164.18	151.55	171.62	148.91	138.31	117.57	139.45	156.5	149.91	130.2	122.69	114.69
			T	×	×	×	×	×	×	×	×	5381.55	4475.37	3133.72	3130.89	2920.06	2765.2	2784.83
			MEM	×	×	×	×	×	×	×	×	2192.96	1867.83	1602.25	1388.87	1206.87	1055.31	924.49

TABLE 9.8
100 × 100 *Fire2* on 20 Nodes

Metrics	TW	LTW	LTW vs. TW
PEE	68,346.55	11,658.75	
VEE	0	56,057.00	
PQLen	17,533.37	2149.91	↓ 87.74%
VQLen	0	75.31	
SS	33,833.00	17,565.40	↓ 48.08%
OPT-SK	0	15,591.10	
FCT	245.12	58.36	↓ 76.19%
PriRB	769.95	740.55	↓ 3.82%
SecRB	12,794.35	2036.45	↓ 84.08%
RB	13,564.30	2777.00	↓ 79.53%
EI	46,877.55	7197.90	↓ 84.65%
ER	29,512.45	6651.60	↓ 77.46%

TABLE 9.9
20 × 20 × 2 *Watershed* on 18 Nodes

Metrics	TW	LTW	LTW vs. TW
PEE	12,53641.94	3,61,457.78	
VEE	0	8,56,256.00	
PQLen	33,4016.67	77,790.62	↓ 76.71%
VQLen	0	26.04	
SS	37,1273.33	73,186.94	↓ 80.29%
OPT-SK	0	2,88,247.50	
FCT	61,313.67	395.63	↓ 99.35%
PriRB	173.50	159.94	↓ 7.81%
SecRB	22,816.67	2165.33	↓ 90.51%
RB	22,990.17	2325.28	↓ 89.89%
EI	6,25,210.33	1,75,521.11	↓ 71.93%
ER	5,69,337.94	1,72,280.33	↓ 69.74%

9.7 SUMMARY

In this chapter we presented and analyzed the performance of our two existing parallel CD++ simulators, namely the Conservative PCD++ simulator and Optimistic PCD++ simulator with a focus on the last one. We looked at the design and implementation of these PCD++ simulators and compared their structures as well as functionalities in parallel and distributed simulations. The hierarchical structure of the conservative PCD++ simulator was compared against the flattened structure of the optimistic PCD++ simulator. We illustrated the migration from a hierarchical structure to a flattened structure as two major modifications; the departure from

a conservative-based simulator to an optimistic-based simulator, and flattening the structure. Then we showed how the optimistic PCD++ simulator deals with the communication overhead by a flat structure.

Aiming at improving the performance of the optimistic simulator, we modified the WARPED kernel to handle rollbacks in a more efficient way. We presented new algorithms that we have implemented in the WARPED kernel: The NPSI protocol and the LRFM, and the GFRM. We ran a variety of tests to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative, as well as our LRFM and GRFM Time Warp-based protocols. The main goal of these tests was to show the maximum capability of the two mentioned PCD++ simulators in terms of handling the number of nodes driving the simulation, complexity of the model, and the size of the model.

A novel optimistic synchronization protocol, referred to as LTW, has been proposed and its performance compared to the standard Time Warp. LTW offers a novel approach that systematically addresses several important issues of TW-based systems (especially for DEVS-based simulations that require a large number of simultaneous events to be executed at each virtual time). It allows purely optimistic simulations to be driven by only a few full-fledged TW LPs, preserving the dynamics of the TW mechanism, while at the same time, accelerating the execution in each local simulation space significantly. Both TW and LTW have been implemented in optimistic PCD++. The experimental results demonstrated that LTW outperforms TW in various aspects, including shortened execution time, reduced memory usage, lowered operational cost, and enhanced system stability and scalability. We are currently working on integrating LTW with other TW optimizations to further improve performance. By taking advantage of the LTW protocol, we are also investigating dynamic process creation, deletion, and migration schemes to support more efficient load balancing as well as runtime structure changes in optimistic DEVS systems.

REFERENCES

1. Fujimoto, R. M. *Parallel and Distributed Simulation Systems*. New York: Wiley, 2000.
2. Wainer, G. "CD++: A Toolkit to Develop DEVS Models." *Software—Practice and Experience* 32 (2002): 1261–1306.
3. Wainer, G. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton, FL: CRC Press/Taylor and Francis, 2009.
4. Wainer, G. "Applying Cell-DEVS Methodology for Modeling the Environment." *Simulation: Transactions of the Society for Modeling and Simulation International* 82, no. 10 (2006): 635–60.
5. Farooq, U., G. Wainer, and B. Balya. "DEVS Modeling of Mobile Wireless Ad Hoc Networks." *Simulation Modeling Practice and Theory* 15, no. 3 (2006): 285–314.
6. Yu, J., and G. Wainer. "eCD++: An Engine for Executing DEVS Models in Embedded Platforms." In *Proceedings of the 2007 Summer Computer Simulation Conference*, San Diego, CA, 2007.
7. Troccoli, A., and G. Wainer. "Implementing Parallel Cell-DEVS." In *Proceedings of the 36th IEEE Annual Simulation Symposium (ANSS'03)*, 273–80, Orlando, FL. New York: IEEE, 2003.

8. Wainer, G., and E. Glinsky. "Advanced Parallel Simulation Techniques for Cell-DEVS Models." *Simulation News Europe*. EUROSIM 16, no. 2 (2006): 25–36.
9. Liu, Q., and G. Wainer. "Parallel Environment for DEVS and Cell-DEVS Models." *Simulation, Transactions of the SCS* 83, no. 6 (2007): 449–71.
10. Wainer, G., K. Al-Zoubi, and R. Madhoun. "Distributed Simulation of DEVS and Cell-DEVS Models in CD++ Using Web-Services." *Simulation Modeling, Practice and Theory* 16, no. 9 (2008): 1266–92.
11. Chow, A. C., and B. Zeigler. "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism." In *Proceedings of the Winter Computer Simulation Conference*, Orlando, FL, 1994.
12. Jefferson, D. R. "Virtual Time." *ACM Trans. Program. Lang. Syst.* 7, no. 3 (1985): 404–25.
13. Fleischmann, J., and P. A. Wilsey. "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators." In *Proceedings of PADS '95*, 50–58, Washington, DC, 1995.
14. Steinman, J. S. "Breathing Time Warp." *SIGSIM Simul. Dig.* 23, no. 1 (1993): 109–18.
15. Lin, Y. B., and E. D. Lazowska. "A Study of Time Warp Rollback Mechanisms." *ACM Trans. Model. Comput. Simul.* 1, no. 1 (1991): 51–72.
16. Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. "Efficient Optimistic Parallel Simulations Using Reverse Computation." *ACM Trans. Model. Comput. Simul.* 9, no. 3 (1999): 224–53.
17. Radhakrishnan, R., D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. "An Object-Oriented Time Warp Simulation Kernel." In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, LNCS 1505, 13–23, 1998.
18. Zeigler, B. P., G. Ball, H. Cho, J. Lee, and H. Sarjoughian. "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions." In *1999 Fall Simulation Interoperability Workshop*, 1999.
19. Nutaro, J. "Risk-Free Optimistic Simulation of DEVS Models." In *Proceedings of the Advanced Simulation Technologies Conference*, Arlington, VA, 2004.
20. Nutaro, J. "On Constructing Optimistic Simulation Algorithms for the Discrete Event System Specification." *ACM Transactions on Modeling and Computer Simulation* 19(1), no. 1, 2008.
21. Srinivasan, S., and P. F. Reynolds. "Elastic Time." *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, no.2 (1998): 103–39.
22. Szulcsztein, E., and Wainer, G. "New Simulation Techniques in WARPED Kernel" (in Spanish). *Proceedings of JAIIO*, Buenos Aires, Argentina, 2000.
23. Reynolds, P. F. "A Spectrum of Options for Parallel Simulation." *Proceedings of the 20th Winter Simulation Conference*, 325–32, San Diego, CA, 1988.
24. Preiss, B. R., and W. M. Loucks. "Memory Management Techniques for Time Warp on a Distributed Memory Machine." *SIGSIM Simul. Dig.* 25, no.1 (1995): 30–39.
25. Jefferson, D. "Virtual Time II: Storage Management in Conservative and Optimistic Systems." In *PODC '90: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, 75–89, New York, 1990.
26. Lin, Y.-B., and B. R. Preiss. "Optimal Memory Management for Time Warp Parallel Simulation." *ACM Trans. Model. Comput. Simul.* 1, no. 4 (1991): 283–307.
27. Brown, R. "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem." *Commun. ACM* 31, no. 10 (1988): 1220–27.
28. Rönngren, R., R. Ayani, R. M. Fujimoto, and S. R. Das. "Efficient Implementation of Event Sets in Time Warp." *SIGSIM Simul. Dig.* 23, no. 1 (1993): 101–8.
29. Oh, S., and J. Ahn. "Dynamic Calendar Queue." In *Proceedings 32nd Annual Simulation Symposium*, 1999, 20–25, 1999.

30. Schof, S. "Efficient Data Structures for Time Warp Simulation Queues." *Journal of Systems Architecture* 44, no. 6 (1998): 497–517.
31. El-Khatib, K., and C. Tropper. "On Metrics for the Dynamic Load Balancing of Optimistic Simulations." In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS-32)*, Maui, HI, 1999.
32. Wilson, L. F., and W. Shen. "Experiments in Load Migration and Dynamic Load Balancing in SPEEDS." In *WSC'98: Proceedings of the 30th Conference on Winter Simulation*, 483–90, Los Alamitos, CA, 1998.
33. Carothers, C. D., and R. M. Fujimoto. "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms." *IEEE Transactions on Parallel and Distributed Systems* 11, no. 3 (2000): 299–317.
34. Peschlow, P., T. Honecker, and P. Martini. "A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation." In *PADS '07: Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, 219–28, Washington, DC, 2007.
35. Reiher, P. L., and D. Jefferson. "Dynamic Load Management in the Time Warp Operating System." *Trans. Soc. Comput. Simul. Int.* 7, no. 2 (1990): 91–120.
36. Li, L., and C. Tropper. "Event Reconstruction in Time Warp." In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS 2004)*, 37–44, 2004.
37. Liu, Q., and G. Wainer. "Lightweight Time Warp—A Novel Protocol for Parallel Optimistic Simulation of Large-Scale DEVS and Cell-DEVS Models." In *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2008)*, 131–38, Vancouver, BC, Canada, 2008.
38. Rajaei, H. "Local Time Warp: An implementation and Performance Analysis." *Proceedings of PADS*, 163–70, 2007.
39. Rothermel, R. *A Mathematical Model for Predicting Fire Spread in Wild-Land Fuels*. Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1972.
40. Moon, Y., B. Zeigler, G. Ball, and D. P. Guertin. "DEVS Representation of Spatially Distributed Systems: Validity, Complexity Reduction." *IEEE Transactions on Systems, Man and Cybernetics*, 1996, 288–96.
41. Al-aubidy, B., A. Dias, R. Bain, S. Jafer, M. Dumontier, G. Wainer, and J. Cheetham. "Advanced DEVS Models with Applications to Biomedicine." *Artificial Intelligence, Simulation and Planning*, Buenos Aires, Argentina. AIS, 2007.
42. Jafer, S., and G. Wainer. "An Environment for Advanced Parallel Simulation of Cellular Models." *Proceedings of the International Conference on Unconventional Computation '07*, 2007.
43. Ameghino, J., A. Troccoli, and G. Wainer. "Models of Complex Physical Systems Using Cell-DEVS." *The 34th IEEE/SCS Annual Simulation Symposium*, 2001.
44. Bevins, C. D. *fireLib User Manual and Technical Reference*, <http://www.fire.org/>, accessed in December 2008.

10 From Virtual to Real—A Progressive Simulation-Based Design Framework

Xiaolin Hu

CONTENTS

10.1	Introduction	271
10.2	Background and Related Work	273
10.2.1	DEVS for Real-Time System Modeling and Development.....	273
10.2.2	Related Work on Simulation-Based Design.....	275
10.3	The Progressive Simulation-Based Design Framework.....	276
10.3.1	Model Continuity	276
10.3.2	Simulation-Based Virtual Environment	278
10.3.3	The Progressive Simulation-Based Design Process	280
10.4	A Dynamical Team Formation Multirobot System	281
10.4.1	Dynamical Team Formation	281
10.4.2	The Multirobot Test Bed	284
10.5	Development of the Multirobot System and Experiment Results.....	285
10.5.1	Overview of the Development Process	285
10.5.2	Experimental Results.....	286
10.6	Conclusion.....	289
	References.....	289

10.1 INTRODUCTION

System design is a process evolving a system from an initial conceptual formulation, to model development and verification, to physical prototyping and experimentation, and to the final system realization in the field. For large scale distributed real-time systems, such a process is not easily achieved. This is because of factors such as distributed processing, temporal requirements, limited resources, concurrent environmental entities, and high-reliability requirements. The increasing complexity of these systems calls for systematic methods as well as supporting environments to design, test, and measure them. To serve this purpose, modeling and simulation (M&S) are frequently employed. With M&S, the system under development and its environment can be modeled. This allows designers to test and evaluate the models on computers.

While the role of simulation has been generally recognized in system design, it is not treated in systematic ways. In particular, there exist two common deficiencies for many simulation-based methods. First, the simulation models, which are developed in the early stages of a development, are usually discarded as unusable by the implementation stage. Second, simulation is typically viewed as an activity only in the model world. When physical system components are brought into development, the exercise of simulation stops and the real system experiment takes over. This transition from simulation to a real system experiment is usually carried out in ad-hoc ways. Indeed, we notice a dichotomy: the model-based development happens in the model world, where analytic techniques and simulation methods are frequently applied. Real system-based experiments happen in the real world, where all involved entities are physical system components, and real experiments can be performed. Between these two end points of system evolution, there is little to support design and measurement in the intermediary stages characterized by *combinations* of models and real system components. This results in a gap between simulation and real system experimentation. For large-scale complex systems, such a gap is significant as the transition from simulation to real system experimentation cannot be easily achieved.

This chapter presents a progressive simulation-based design framework that treats simulation as a driving force for all stages of a design process evolving from the model world to the real world. The design process starts from all models that are simulated on computers, and proceeds by bringing real system components into the simulation to replace their virtual counterparts (models), and ends when all components are real and the final system is tested in a physical environment. Throughout this process, model continuity is emphasized so the same simulation model (or transforms of it) is maintained at different stages and eventually acts as the control software of the distributed real-time system. Carrying out this design process in a progressive way is motivated by two important needs of simulation-based development of large-scale distributed real-time systems. First, there is a need to handle the complexity in a controlled manner when transitioning a design from the model world to the real world. The progressive simulation-based design achieves this by supporting a design stage where real system components are (gradually) included into the simulation to work together with simulation models. This allows designers to work on a partially implemented system while still being able to carry out system-wide tests, where the rest of the system is fulfilled by simulation models. Doing this incrementally allows designers to focus on specific design aspects of a system and to gain confidence about how the final system will behave. Secondly, simulation-based design and measurement rely on high-fidelity models. There is a need to continuously and systematically improve the models as a design proceeds. When real system components are brought into a simulation-based study, new design details overlooked before will be revealed. The progressive design framework utilizes the incremental transition from virtual to real to provide feedback for designers to validate their design assumptions and to update simulation models. Because of the need to gradually replace models with real system components, the progressive design framework is best applied to distributed real-time systems where different subsystems are modularly and/or loosely coupled. In particular, this chapter focuses on multirobot systems as the application context.

The simulation-based design is based on the Discrete Event System Specification (DEVS) [1,2] formalism that is derived from dynamic systems theory. DEVS has well-defined concepts of the coupling of components; hierarchical, modular model construction; support for discrete-event approximation of continuous systems; and an object-oriented substrate supporting repository reuse. It benefits from the property of closure under coupling, which justifies treating coupled models as components and enables hierarchical model composition constructs. DEVS supports a formal M&S framework that treats system, model, simulator, and their relationships in a well-defined manner. This formal foundation supports developing systematic methods such as the model continuity method described later, where the same model is tested incrementally by different simulators. Built on this formal foundation, this work applies DEVS to simulation-based system development.

The remainder of this chapter is organized as follows. Section 10.2 presents the background of DEVS for real-time system modeling and development, and the related work on simulation-based design. Section 10.3 presents the progressive simulation-based design framework that includes model continuity, robot-in-the-loop simulation, and the progressive simulation-based design process. This framework is illustrated by a multirobot example. Section 10.4 describes this multirobot dynamical team formation example and the test bed where the system was developed. Section 10.5 documents the development process of this example and shows experiments results to demonstrate the effectiveness of the design framework. Section 10.6 concludes this work.

10.2 BACKGROUND AND RELATED WORK

This section reviews the background of DEVS for real-time system modeling and development and some related work on simulation-based design.

10.2.1 DEVS FOR REAL-TIME SYSTEM MODELING AND DEVELOPMENT

Real-time systems are computer systems that monitor, respond to, or control an external environment. This environment is connected to the computer system through sensors, actuators, and other input–output interfaces [3]. The one common feature of all real-time systems is timeliness. For these systems, a systematic time handling and time modeling approach will benefit both system design and verification. It also follows from the very nature of most real-time applications that there is a stringent requirement for high reliability. Furthermore, real-time systems generally have a concurrent event-driven nature and are characterized by their continuous interaction with the environment. This demands effective test methods and verification techniques during system development. Formal computation models, such as finite state machine [4], timed automata [5], Petri Net [6], and temporal logic [7,8,9] have been employed for the design, analysis, and implementation of real-time software. A formal model is an essential ingredient of a sound system-level design methodology because it makes it possible to capture the required functionality, verify the correctness of the functional specification, and synthesize the specification independent of the tool [10,11].

DEVS is a formal model that provides systematic modeling capability to capture the structure, behavior, and timeliness of a system under development. The structure of a system identifies the entities and the relationships between them (e.g., communication relationships and containment relationships). This can be modeled by DEVS coupled models. Corresponding to the different subsystems or entities in a system, a DEVS coupled model contains several component models (DEVS atomic model or coupled model), each of which defines its own input and output ports. DEVS couplings are established between the output/input ports to enable intercommunications between models. Within this framework, a system that exhibits an intercommunication relationship as well as a hierarchical containment relationship between its entities can be naturally modeled. This highly modular modeling approach also greatly promotes repository reuse. The behavior of a system can be modeled by DEVS atomic models, which have a well-defined state, state transition function (triggered by external or internal events), time advance function, and output function. From the design point of view, we can view a DEVS atomic model as a timed state machine. The transition from one state to another is triggered by external or internal events. The model can generate output and send it out through its output ports.

With timeliness being an essential property of any real-time system, DEVS models handle time explicitly. This is accomplished by the time advance function in DEVS atomic models. Whenever an atomic model transits to a new state, its time advance function specifies how long the model will stay at that state. During this period of time, if there are external events, a model responds to them in its external transition function, which may change the model to a new state with a new time period; otherwise if an “internal” time-out event is generated, the model responds to it in its internal transition function. Figure 10.1 illustrates several time patterns used in real-time systems that are easily modeled in DEVS. In the figure, the Periodic Model maintains an active state at every time period; the Delay Model holds in a state until the delay time elapses; the Condition Model stays in the wait state until some

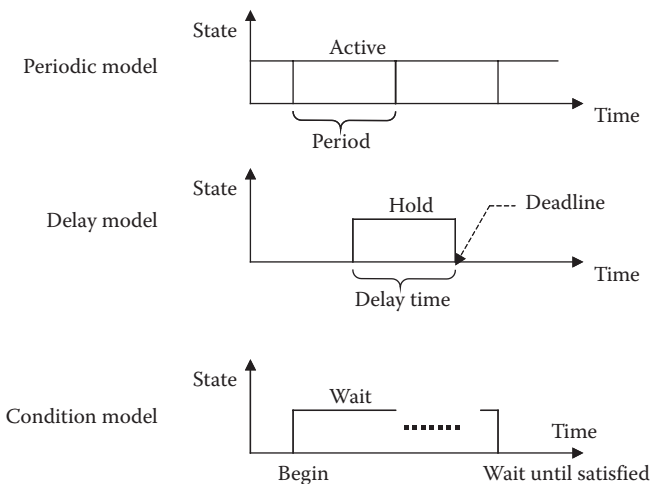


FIGURE 10.1 Time patterns modeled in DEVS.

condition is satisfied, for example, triggered by an external event. Some DEVJSJAVA code for these time patterns can be found in [12].

The formal foundation of DEVS supports systematic simulation-based methods to evaluate the behavior and/or performance of a system under development. Based on the simulation infrastructures and techniques that have been developed, DEVS has the ability to test and evaluate a model's correctness as the model is iteratively refined on different abstraction levels. This allows early discovery of high-level defects as well as validation of the finalized model in detail. In DEVS, simulation-based test and evaluation are conducted using appropriate experimental frames derived from the design objectives. An experimental frame is a specification of the conditions under which the system is observed or experimented with [2]. With experimental frames, not only the correctness of a model can be tested and validated, but also the performance of the model, such as average response time, can be measured. Moreover, different perspectives of the system can be captured through specialized experimental frames and tested in the various phases of development. The combination of formal modeling capability and systematic simulation-based methods makes DEVS a competitive candidate for real-time systems development. Over the years, DEVS has been applied to different aspects of real-time system development such as model-based development of real-time embedded systems [13,14], real-time system specification [15], and real-time system analysis [16].

10.2.2 RELATED WORK ON SIMULATION-BASED DESIGN

Using simulation to support system design has been studied under the research umbrella of *simulation-based design*. Simulation-based design is a design approach in which simulation is the primary means of design evaluation and verification [17]. The motivation is that many design decisions affecting the final configurations of a system under development can be taken as a result of computer simulations before real physical experiments. A major application area of simulation-based design is the manufacturing industry, which pioneers simulation-based design with the paradigm of virtual prototyping. Using simulations instead of physical experiments, a virtual prototype makes design verification faster and less expensive. It provides the designer with immediate feedback on design decisions. This in turn promises a more comprehensive exploration of design alternatives and a better performing final design [18]. A well-publicized example of virtual prototyping is the design of the Boeing 777 airplane [19]. Similar approaches are also adopted by car manufacturers (e.g., Bullinger et al. [20]). Simulation-based design is also applied to the design of electronic systems such as hardware/software codesign of embedded systems [21,22]. Our research shares common ground with the above work; that is, developing models for systems to be designed and evaluating them in simulation-based virtual environments. However, we focus on the progressive transition of a design from the model world to the real world. To support that, a major step is design/evaluation with combined models and real system components. Note that focusing on the transition from virtual to real is different from work that is concerned with the transitions from coarse, high-level simulation models to more detailed models (e.g., [18]). Furthermore, our work focuses on the design of (control) software for real-time

systems. This differs from simulation-based design methods that deal with virtual prototypes of physical components.

Using simulation to support software design is adopted by several projects and tools for designing embedded software. The Ptolemy project [23,24] provides a component-based framework to express various computational models related to the embedded systems. It supports modeling of interactions between sets of components that are represented by different computational models. Simulink® [25] is a commercial tool that provides an extensive graphical interface to MATLAB® [26] for interactive modeling and functional simulation. It promotes a paradigm of Simulation and Model-Based Design [27] for designing and testing embedded control and signal processing software. Both Ptolemy and Simulink support implementation of embedded software using code generation from simulation models. However, a systematic process that progressively transitions a design from virtual to real is not well studied. For large-scale distributed real-time systems that have complex software, hardware, and physical operating environments, there is a need for systematic transitions from simulation models to real system implementation and measurement.

10.3 THE PROGRESSIVE SIMULATION-BASED DESIGN FRAMEWORK

The progressive simulation-based design framework includes a *model continuity* development methodology, a *simulation-based virtual environment* that supports simulation-based tests with combined virtual and real system components, and a *progressive simulation-based design process* for distributed real-time system development.

10.3.1 MODEL CONTINUITY

Model continuity refers to the ability to transition as much as possible of a model specification through the stages of a development process. This is opposite to the discontinuity problem where artifacts of different design stages are disjointed and thus cannot be effectively consumed by each other. For distributed real-time systems, we restrict model continuity to the models that implement the system's real-time control. This means the control models of a distributed real-time system can be designed, analyzed, and tested by simulation methods, and then smoothly transitioned from simulation to distributed execution [12,28].

To support model continuity it is essential to develop system models and run simulation-based tests in a systematic way. Figure 10.2 displays the major activities involved in the model continuity methodology for developing distributed real-time systems. The top portion of the figure is about the modeling stage that develops the system model. The bottom portion of the figure is about the simulation stage that carries out stepwise simulation-based tests. In the modeling stage, we distinguish the system from its environment and develop models for both of them. To develop the system model, the subsystems (e.g., the individual robots) and their communication network structure are identified. This allows designers to develop the network

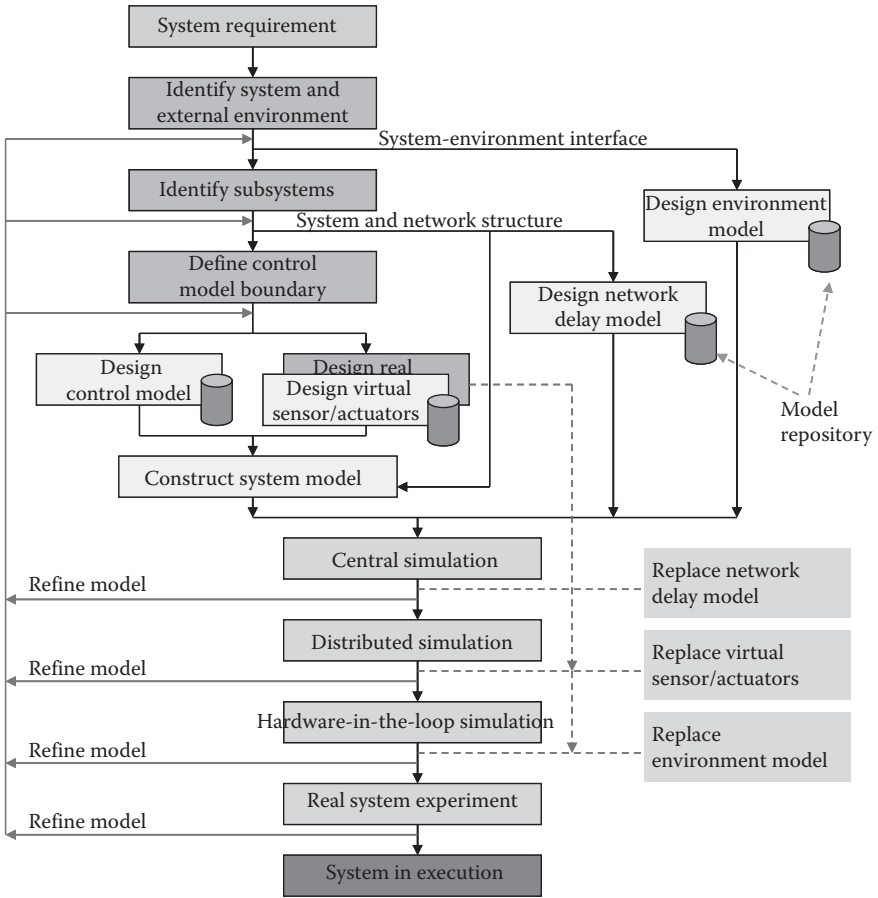


FIGURE 10.2 The model continuity methodology. (From Hu, X., *A Simulation-Based Software Development Methodology for Distributed Real-Time Systems*, PhD Dissertation, Electrical and Computer Engineering Department, University of Arizona, Tucson, May 2004.)

model (e.g., to model the communication latency between subsystems) for supporting simulation-based tests.

For each subsystem, we distinguish its control model (also referred to as the decision-making model) from the sensors/actuators. Models of sensors/actuators (also referred to as virtual sensors/actuators) are developed. In this approach, the control model acts as the decision-making component to process data and to make decisions. Virtual sensors/actuators act as the interface to provide functions for the control model to interact with the environment. It is important to note that the interfaces between the control model and the virtual sensors/actuators should be well defined and consistent with how the real sensors/actuators will be used. This allows designers to replace the virtual sensors/actuators with real sensors/actuators in later design stages without modifying the control model.

To illustrate this modeling approach, we consider a simple example where a mobile robot moves randomly in an area and avoids obstacles using its infrared (IR) sensors. The physical area and obstacles are modeled by the environment model. The robot model includes models for the IR sensors (including their layout on the robot if multiple sensors exist), a model for the motor, and a decision-making model that receives input from the IR sensor models and issues commands to the motor model. Examples of the decision-making model can be a DEVS atomic model realizing a finite state machine, or a DEVS coupled model realizing a multilayer hierarchical control structure. The network delay model is not needed in this example because there is only one robot.

The modular modeling approach described above allows the decision-making model of a distributed real-time system to be designed and tested incrementally, and then deployed to a distributed environment for execution. As shown in the bottom portion of Figure 10.2, the model continuity methodology supports a design and test process having four steps: (1) *Centralized simulation* to analyze the system under a test within a model of the environment linked by virtual sensors/actuators on a single computer. (2) *Distributed simulation* in which models are deployed to the real network that the system will be executed on and simulated by distributed real-time simulators. (3) *Hardware-in-the-loop (HIL) simulation* in which the environment model is simulated by a DEVS real-time simulator on one computer while the control model under test is executed by a DEVS real-time execution engine on the real hardware. (4) *Real system experiment* in which a DEVS real-time execution engine executes the control model that interacts with the real environment through the sensor/actuator interfaces that drive the real sensors/actuators. More technical details about these stepwise simulation-based tests can be found in Hu [12] and Hu and Zeigler [28].

The model continuity methodology reduces the possibility of design discrepancy along the development process, and increases the confidence that the final system realizes the specification as desired. Furthermore, it makes the design process easier to manage since continuity between models of different design stages is retained.

10.3.2 SIMULATION-BASED VIRTUAL ENVIRONMENT

To smooth the transition from conventional simulation to real system experiments and to bridge the gap between them, we developed, in the context of distributed robotic systems, a simulation-based virtual environment that allows combined real robots and virtual robot models to work together [29,30]. We use the term Robot-In-the-Loop (RIL) simulation to refer to this capability of experimenting with several real robots in a simulation-based virtual environment containing many robot models.

We note that the term “robot-in-the-loop simulation” is directly related to “hardware-in-the-loop simulation.” HIL simulation is a technique frequently applied in computer system design, especially for embedded systems and control systems design. It refers to a system in which parts of a pure simulation have been replaced with actual physical components. This is based on the belief that once physical

components are added into the loop, unmodeled characteristics can be investigated, and controls can be further refined. RIL simulation, by sharing the same belief that higher simulation fidelity will be reached when real system components are brought into study, provides several advantages that are especially useful for complex and large-scale robotic systems. For example, when a physical environment is not available to test a robot, RIL simulation allows the robot to be tested and evaluated in a virtual environment. For distributed robotic systems that include a large number of robots, RIL simulation also allows one or two real robots to be tested and evaluated together with many other robot models simulated on computers.

RIL simulation is carried out by replacing some robot models in a distributed robotic system with real robots. These real robots use a combination of virtual and real sensors/actuators (this is feasible because the model continuity methodology allows a robot's control model to work with virtual or real sensor/actuators in the same manner). For example, a real mobile robot may use its virtual IR sensors to get sensory inputs from the virtual environment and use its real motors to move the robot. As a result, this real robot moves in a physical space based on the sensory input from a virtual environment. Meanwhile, this real robot can sense other robot models simulated on computers and communicate/coordinate with them.

Figure 10.3 shows a snapshot of a RIL simulation, which demonstrates the coordination between real robots and robot models. The figure shows a robot convoy system with four robots (denoted by $R0$, $R1$, $R2$, and $R3$), among which the second and third ones ($R1$ and $R2$) are real robots. $R1$ uses virtual IR sensors to receive sensory input from the environment model. $R2$ uses real IR sensors to sense its front robot ($R1$) and the real environment. There are two windows in the figure. The right window shows how two real robots move in the real world. The left window is the simulation window. It shows the movements of the entire convoy system, among which the second and third robots are the counterparts of the two real robots ($R1$ and $R2$). Thus

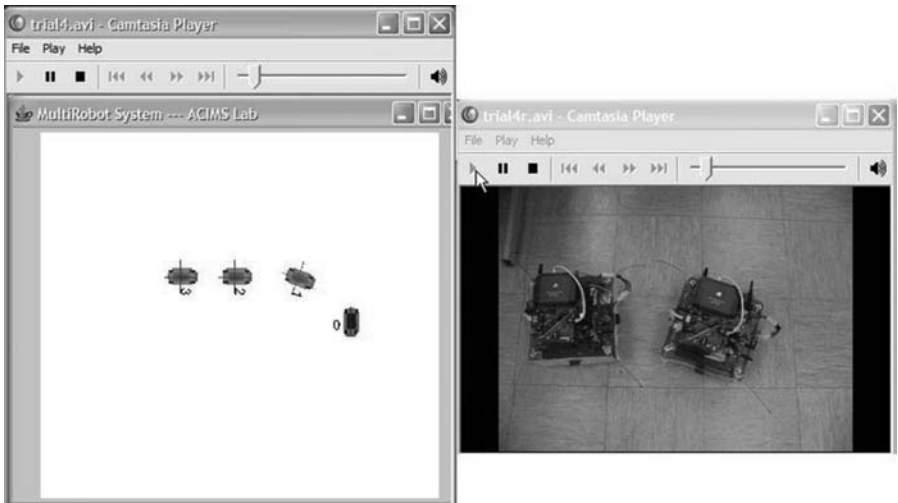


FIGURE 10.3 Snapshot of an RIL simulation.

the second and third robots' movements in the simulation window are synchronized with the two real robots' movements in the real world. This synchronization is necessary to provide robots with the correct sensory information.

The capability of RIL simulation brings simulation-based study one-step closer to reality, and provides the flexibility to allow experimentation with real robots in a virtual environment. Measurement results from RIL simulations can be compared with those from conventional simulations. Such comparisons will provide useful feedback to the designers; that is, indicating that a robot model may not model the real robot's movement very well. Meanwhile, by using real robots, RIL simulation increases designers' confidence about how the final system is going to behave. Note that in both of these two cases, RIL simulation allows the designers to use only a few, instead of all, of the real robots to gain the above knowledge. Because of this, the capability of RIL simulation is especially useful for large-scale cooperative robotic systems whose complexity and scalability severely limit experimentations in a physical environment using real robots.

10.3.3 THE PROGRESSIVE SIMULATION-BASED DESIGN PROCESS

Built on the model continuity methodology and the simulation-based virtual environment described above, a progressive simulation-based design process is developed. This design process views simulation as the driving force for designing and testing distributed real-time systems. It explicitly focuses on systematic transitions from simulation models to real system realization.

As shown in Figure 10.4, the design process consists of three stages, each of which is characterized by the types of entities (virtual or real) that are involved. The first stage is *conventional simulation*, where simulation is carried out using all models. A major task of this stage is to develop the system model based on knowledge (and assumptions) about the real system's hardware and operating environment. Often discrepancies between simulation models and real system components exist. These discrepancies cause the control models tested in conventional simulations to result in undesirable behavior when applied to a real system in a physical environment. To reveal such design discrepancies, the next stage of the design process is *robot-in-the-loop simulation*, where simulation-based study is carried out using combined real

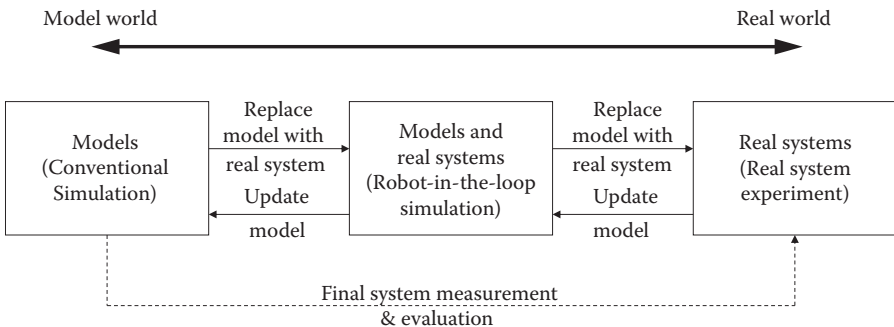


FIGURE 10.4 The progressive simulation-based design process.

robots and robot models. This stage brings simulation-based study one step closer to reality by including real robots into the simulation. The motivation is to use some (not all) real robots to reveal any overlooked design details, thus helping to test and improve the decision-making models of robots. The final stage is *real system experiment*, where all real robots are tested in a physical environment.

The progressive design process emphasizes two parallel activities: *replace models with real system components*, and *update models* along the process. As the design moves forward, real system components are gradually brought into the simulation to replace models. Simulations with these real system components allow designers to validate their design assumptions and to reveal new design details overlooked before. Such information is fed back to the previous stages to update the models if needed. The updated models will then be used for follow-up designs and testing. This activity of model update is important in order to maintain a coherent model of the system under development. Thus at the end of the design, not only the system is realized and tested, but also a system model that faithfully represents the system is developed. This system model can support final system measurement and evaluation (shown by the dashed line in Figure 10.4), as well as serve other purposes such as system maintenance and future development. It is important to note that each design stage is a dynamical evolving process by itself. For example, during the conventional simulation stage, it is common for designers to start from high-level models and then refine them to more detailed models. Similarly, the robot-in-the-loop simulation stage may start from including one real robot and then gradually add more.

10.4 A DYNAMICAL TEAM FORMATION MULTIROBOT SYSTEM

A dynamical team formation multirobot system was developed based on the progressive simulation-based design framework. The development of this system started from a simulation model initially developed in Hu and Edwards [31], which tests a bio-inspired, behavior-based control mechanism. This section presents the dynamical team formation multirobot system and the robot test bed where the multirobot system was developed.

10.4.1 DYNAMICAL TEAM FORMATION

The dynamical team formation system includes a group of autonomous mobile agents, which search for partners, form subteams dynamically, and eventually convoy as a single large team. To help the team formation, agents are assigned IDs consecutively from 1 to T , where T is the total number of agents. The agent that has the ID k is denoted as R_k . The team formation process starts with all agents being randomly deployed in a rectangular field. The goal of each agent is to search for its immediate front or back agent (based on agent IDs, for example, R_{k-1} is the front agent of R_k) and form a subteam with it. In order to form larger teams, an agent needs to maintain its existing subteam while searching for other agents. This team formation process continues until all agents (or subteams of agents) merge into a single team, and then convoy in a line formation. Figure 10.5 shows this process with three snapshots from a simulation with 15 agents. In this simulation, agents were initially

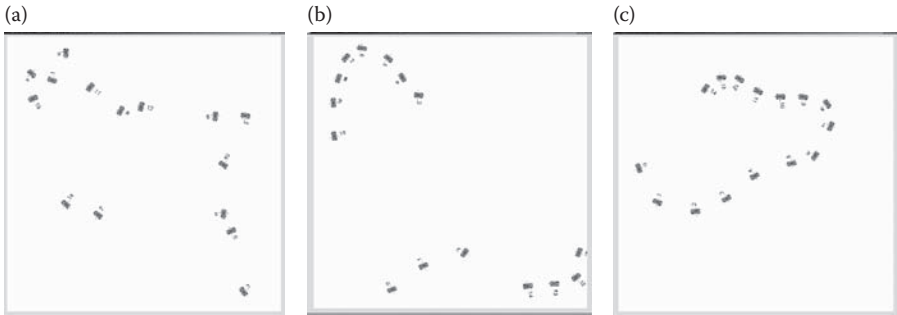


FIGURE 10.5 A simulation of dynamical team formation. (a) Initial locations, (b) subteams formed, and (c) convoy in a single team.

randomly distributed in a rectangle space as shown in Figure 10.5a. They started to form subteams as shown in Figure 10.5b. Finally Figure 10.5c shows that they succeeded in forming a single team and convoyed in a line formation.

To achieve the adaptive behavior needed during the process of dynamical team formation, the decision-making model of each robot employs a mutual inhibition behavior network architecture. This architecture uses mutual inhibition as the major mechanism to achieve adaptive behavior selection among competitive behaviors. The set of inhibitory coefficients that define the strength of inhibition between pairs of behaviors are context-dependent; that is, they are modulated by a higher layer called Behavioral Context. This allows the relative priorities of the behaviors to be dynamically changed, thus making the agent exhibit different behavior patterns when it switches to different operating conditions (behavioral contexts).

In the dynamical team formation system, a robot has four behaviors: *Avoid*, *Follow*, *Wait*, and *Search*. The *Avoid* behavior controls the robot to avoid collisions with obstacles (including other robots); the *Follow* behavior controls the robot to follow its front robot; the *Wait* behavior controls the robot to wait for its back robot; and the *Search* behavior controls the robot to move in a particular pattern to search for its front or back robot. Each robot has two behavioral contexts: *team_forming* and *convoy*. The *team_forming* behavioral context favors movement behaviors such as *Search* and *Follow* and thus allows a robot to move/search “aggressively” without waiting for other robots. After the robot finds its back robot and forms a subteam with it, it switches to the *convoy* behavioral context. This behavioral context favors behavior *Wait* so the robot moves more “conservatively” as it should wait (stop moving) to maintain the subteam whenever the back robot falls back. Specific values of the inhibitory coefficients of these two behavioral contexts can be found in Hu and Edwards [31] and are omitted here.

Each of the four behaviors is defined by how it is excited by sensory inputs and what actions to carry out if it is selected. Considering agent R_k as an example, the definitions of these behaviors, including the excitation formula and the actions of the behaviors are given in Tables 10.1 and 10.2, respectively. In Table 10.2, a notation $[n1, n2]$ means a random number between $n1$ and $n2$ and an arrow $\rightarrow$ indicates the sequence of tasks. The tasks of a behavior are stored in a task queue

TABLE 10.1
Excitation Formulas of the Four Behaviors

	Excitation	Explanation
Avoid	$E_{avoid} = \min\left(e^{-\frac{d-10}{10}}, 1\right)$	d = Front IR Distance
Follow	$E_{follow} = \begin{cases} 0.9 & \text{if } k=1 \\ \min\left(0.4 + 0.6 \times e^{-\frac{d-100}{10}}, 1\right) & \text{o.w.} \end{cases}$	d = Distance to R_{k-1}
Search	$E_{search} = \begin{cases} 0.8 & \text{if } d > 150 \text{ and } k \neq 1 \\ 0 & \text{o.w.} \end{cases}$	d = Distance to R_{k-1}
Wait	$E_{wait} = \begin{cases} 0 & \text{if } k = T \\ \min\left(e^{-\frac{d-120}{10}}, 1\right) & \text{o.w.} \end{cases}$	d = Distance to R_{k-1}

TABLE 10.2
Action Tasks of the Four Behaviors

	Action
Avoid	Move backward for [10, 20] → rotate clockwise for [20, 30] degrees. Using non resumable task queue.
Follow	Turn to agent R_{k-1} → move for forward or backward to keep desired distance ($d_{desired}$) of 20. Using nonresumable task queue.
Search	Move forward for [30, 50] → rotate counter clockwise for [70, 80] → move forward for [30, 50] → rotate clockwise [70, 80]. Using resumable task queue.
Wait	Null

(resumable or nonresumable) and are sequentially executed as long as the behavior continues to be active. For the first agent R_1 , the action of the *Follow* behavior (not shown in Table 10.2) is to move forward if there is no obstacle ahead and to turn clockwise by 90° if there is an obstacle ahead. More details can be found in Hu and Edwards [31].

We choose the dynamical team formation multirobot system as the case study example for the following two reasons. First, the original model of this system was developed as a high-level simulation model with little consideration of how it will be realized by real robots. This represents a typical design case that starts from simulation of high-level conceptual models and then becomes more and more embodied as the design proceeds. Second, the emergent behavior of dynamical team formation in this example and the connectionist model of the robots' behavior network make this example complex and interesting enough to demonstrate the progressive design framework.

10.4.2 THE MULTIROBOT TEST BED

This section presents the robot test bed where the dynamical team formation system was realized. Figure 10.6a shows the hardware setup of this robot test bed. It includes a rectangle field table (1.2×1.5 m) surrounded by 10 cm high edges and four Khepra II robots. Each robot is equipped with a radio turret that can communicate with a radio base connected to a desktop computer. For localization purposes, an overhead camera is set up above the field table (at about 1.5 m high) so that the images grabbed by the camera cover the entire field of the table. These images are processed by the desktop computer for detecting the locations and headings of the robots in the field.

Figure 10.6b shows the functional components of the software that were developed to interact with this robot test bed. The Control Panel GUI is the main interface through which a user can set up parameters, start/stop simulations, and so on. It also includes a monitoring window that displays live video taken by the overhead camera. The Localization component is responsible for computing the positions and headings of the real robots using image processing techniques based on the grabbed images. To help the image processing, the top face of each real robot is covered by a specially designed six angle circular pattern. This pattern helps the localization component to distinguish a robot from obstacles, and to detect the position, heading, and the unique ID of the robot. The *AgentSys* component represents the simulation system, including models of robots and the environment, and a DEVS simulation engine that drives the simulation. The Communication and Navigation component is responsible for communicating with real robots as well as supporting manual navigations of real robots.

During the robot-in-the-loop simulation and real robot experiments, an image is taken every 100 ms and processed by the localization component. The results of image processing; that is, the positions/headings of real robots, are sent to *AgentSys*,

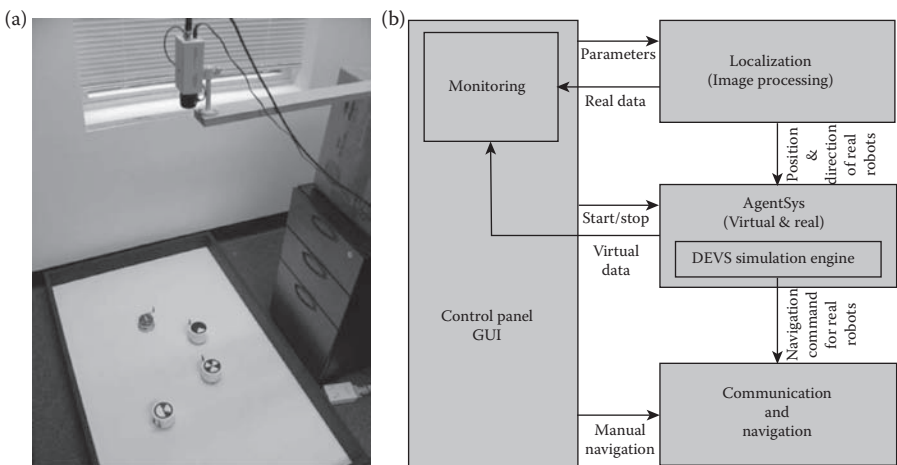


FIGURE 10.6 The multirobot Test Bed. (a) Test bed hardware and (b) test bed software.

which combines them with the positions/headings of virtual robots (if there exist virtual robots) and then dispatches the information to all robots (virtual or real). This allows each robot (virtual or real) to know the positions and headings of the other robots. This information is used by robots to calculate the excitations for their behavior as well as to carry out the action of the selected behavior in every simulation/execution step. More details about this multirobot test bed can be found in Azarnasab and Hu [32].

10.5 DEVELOPMENT OF THE MULTIROBOT SYSTEM AND EXPERIMENT RESULTS

The dynamical team formation multi-robot system described above has been developed using the progressive design process. Below we present this process and some experimental results.

10.5.1 OVERVIEW OF THE DEVELOPMENT PROCESS

The progressive design process of the multirobot system started from the simulation model presented in Section 10.4.1. It goes through the stages of conventional simulation, robot-in-the-loop simulation, and real system experiment. In the design stage of conventional simulation, a major task is to restructure the robot model according to the model continuity methodology (e.g., explicit separation between the control model and the sensor/actuator models of a robot), and add more details into the model to make the simulation embodied (e.g., model the shape and dimensions of the robots). We call this process *embodied modeling*, which is an indispensable step in the transition from a high-level simulation model to a physical system realization. The process of embodied modeling resulted in an environment model and a set of robot models that include control models and sensor/actuator models, such as virtual IR sensors and virtual motors. Simulations using these embodied models allow us to test the correctness and effectiveness of the behavior-based control model and to tune the model parameters.

The design stage of robot-in-the-loop simulation tests the decision-making model using one or more real robots together with robot models. A major task in this stage is to set up the real robots that are included in the robot-in-the-loop simulation. Following the model continuity methodology, this includes transforming the behavior-based control model into the C code that is required by the Khepera II robot and configuring the real robots to use a combination of real and virtual sensors/actuators.

Figure 10.7a shows an example of robot-in-the-loop simulation where two real robots (robot 2 and robot 3) convoy with two virtual robots (robot 1 and robot 4) as a single team. One advantage of robot-in-the-loop simulation is that it can reveal some overlooked design details. To give an example, Figure 10.7b illustrates a collision scenario, where a real robot collided with a virtual one during a convoy process. Note that this kind of collision does not happen in the conventional simulations using all robot models. Further analysis of this problem showed that the collisions

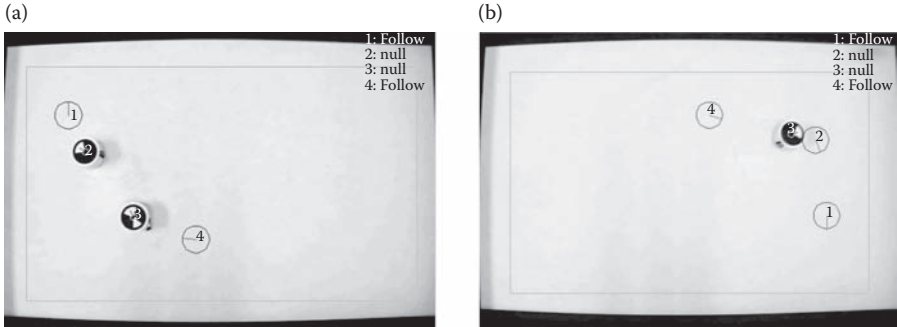


FIGURE 10.7 Robot-in-the-loop simulation. (a) Robot-in-the-loop simulation and (b) collision caused by communication delay.

happened because a real robot did not have the up-to-date information about the robots' positions/headings (including its own position/heading) because of the wireless communication latency of position update.

To solve this problem, we implemented a “position estimation” scheme using a local dead reckoning method, which allows a robot to estimate its current position and heading based on the updated position and the robot's most recent action. The improved decision-making model was retested in a conventional simulation with an updated communication delay model and then in a robot-in-the-loop simulation and showed improved results. The final design stage is a real system experiment that tests the entire system (with all real robots) in a physical environment. In a real system experiment, a real robot uses the same decision-making model refined and tested in the preceding stages and uses the real sensors/actuators.

10.5.2 EXPERIMENTAL RESULTS

The developed control models were deployed to four real Khepera II robots and showed good results. The robots were able to dynamically form a team and then maintain the team in a convoy. Both qualitative and quantitative results were collected. This section shows these results based on two sets of experiments after the design was finished. In both cases, we set up experiments of both simulations (using models) and real executions (using real robots) with the same configurations and then compare the results collected from them.

The first set of experiments compares conventional simulation, robot-in-the-loop simulation, and real execution of four robots in a convoy scenario. In this experiment, we first set up a robot-in-the-loop simulation with two real robots (robot 2 and robot 3) and two virtual robots (robot 1 and robot 4). The initial positions of these robots (real and virtual) were manually set so they already form a team in a line formation as shown in Figure 10.8a. The green lines denote the virtual boundary of the field. With this initial setting, the robots convoy in a clockwise direction inside the virtual boundary with an approximate rectangle trajectory. The simulation was run until the robots complete one round and roughly come back to their original positions. After the robot-in-the-loop simulation, a

conventional simulation with the same initial positions and a real robot execution with four real robots in roughly the same initial positions were run. In each case, a movie was recorded and all robots' time-indexed positions/headings were saved.* These movies allow us to visually see how robots behave and thus qualitatively compare the three experiments.

To quantitatively compare the results, we define a variable *convergence* to indicate how well robots maintain the team in the convoy based on their desired distances to their neighboring robots. We define $convergence(t) = 1/n \sum_{i=1}^n \min(e^{dist_i(t)-D/\sigma}, 1)$ in which n is the total number of robots ($n = 4$ in this example), $dist_i(t)$ is the distance of the i th robot to the following robot at time t , D is the desired distance and is chosen to be $1.5 * R$ (R is the diameter of the robot, $R = 7.5$ cm), and σ is the tolerable difference and is set to 0.2 cm. Figure 10.8b through d shows the convergence graph of a conventional simulation, a robot-in-the-loop simulation, and a real robot execution in this experiment, respectively. In the graphs, the horizontal axis represents the time (in seconds) and the vertical axis represents the convergence value. It can be seen that in all three cases, about four convergence peaks were reached that correspond to the four corners of the rectangular field in one round. The result from conventional simulation and that from robot-in-the-loop simulation are more similar to each other because they both used the same virtual robot as the first robot, which plays the most important role in defining the convoy process (such as the trajectory and pace of the convoy).

The second set of experiments concerns a dynamical team formation scenario with four robots. In this experiment, we first run a real robot execution with the

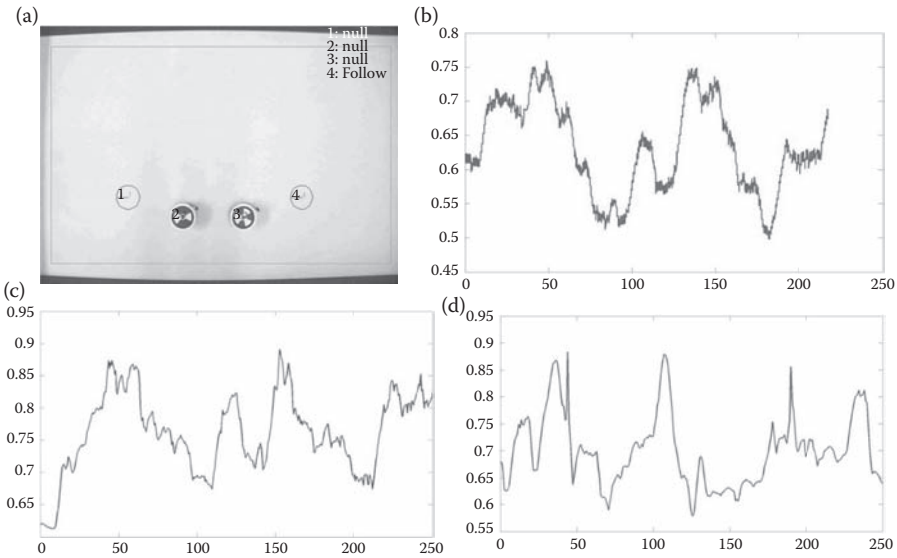


FIGURE 10.8 Experiment 1—Convergence of four robots in a convoy. (a) Initial positions/headings of robots, (b) conventional simulation, (c) robot-in-the-loop simulation, and (d) real robot execution.

* The three movies can be downloaded and viewed from <http://www.cs.gsu.edu/xhu/robotMovie.htm>.

initial positions/headings of the four robots shown in Figure 10.9a. Then a conventional simulation was run using the same initial positions/headings of the four robots. With this setup, during the team formation process the robots first moved toward the center of the field and formed a team, and then convoyed in a small half-circle trajectory, and finally convoyed roughly in a rectangular trajectory. We name this process a “butterfly” team forming process. Similarly, in both cases, movies were recorded and the robots’ positions/headings were saved.* The trajectories of the four robots in conventional simulation and in real robot execution are shown in Figures 10.9b and 10.9c, respectively. These trajectories clearly show the similarity between the simulations using all models and execution using all real robots.

We note that it is impossible to set up a simulation and real robot execution with exactly the same initial conditions. The differences of initial conditions and other factors (such as noise) result in different behaviors and trajectories of the robots in a simulation and in a real execution. These experimental results and the comparisons between simulation and execution demonstrate that: (1) the simulation model resulted from the progressive design process is a good representation of the system under development; and (2) the control model tested in the simulation-based design process can effectively control the real robots to display the expected behaviors in a physical environment. As mentioned before, after the design is finished, the

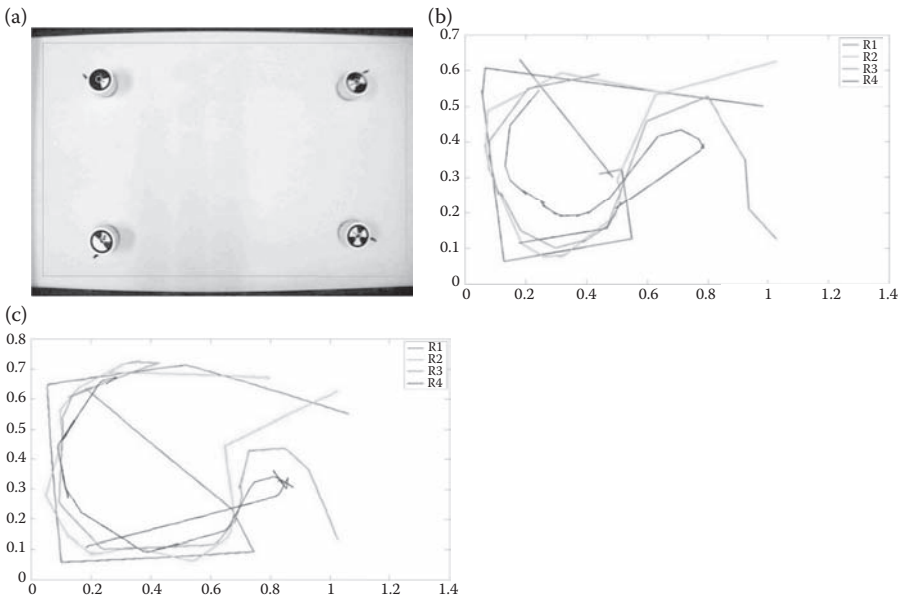


FIGURE 10.9 Experiment 2—Trajectories of the four robots in a dynamical team formation. (a) Initial position/direction of the four robots, (b) trajectory of the four virtual robots, and (c) trajectory of the four real robots.

* The two movies can be downloaded and viewed from <http://www.cs.gsu.edu/xhu/robotMovie.htm>.

simulation model can be used to support simulation-based measurement and evaluation of the final system.

10.6 CONCLUSION

We present a progressive simulation-based design framework for distributed real-time systems, in particular, multirobot systems. The design process of this framework includes multiple design stages and emphasizes two ongoing activities: gradually including real system components into the simulations and constantly using the feedback from these simulations to update the system model. The framework was applied to the development of a multirobot dynamical team formation example and showed good results. Although this paper focuses on design in the context of real-time systems, the concept of progressive simulation-based design and model continuity can be extended or adapted to the development of other types of software intensive systems, such as complex manufacturing systems and defense command and control systems. The applicability of a simulation-based design to these systems is increasingly supported by the high-fidelity simulation environments developed in these fields, as well as the continuous advances of modeling and simulation technologies.

REFERENCES

1. Zeigler, B. P. *Theory of Modelling and Simulation*, New York: Wiley, 1976.
2. Zeigler, B. P., Kim, T. G., and Praehofer, H. *Theory of Modeling and Simulation*, 2nd ed. New York: Academic Press, 2000.
3. Shaw, A. C. *Real-Time Systems and Software*. New York: John Wiley, 2001.
4. Gill, A. *Introduction to the Theory of Finite-State Machines*. New York: McGraw-Hill, 1962.
5. Alur, R., and Dill, D. L. A theory of timed automata, *Theoretical Computer Science* 126 (1994): 183–235.
6. Petri, C. A. *Kommunikation mit Automaten*. Ph.D. Thesis. Bonn, Germany: University of Bonn, 1962.
7. Prior, A. N. *Time and Modality*. Oxford: Clarendon Press, 1957.
8. Prior, A. N. *Past, Present and Future*. Oxford: Clarendon Press, 1967.
9. Prior, A. N. *Papers on Time and Tense*. Oxford: Clarendon Press, 1969.
10. Sgroi, M., Lavagno, L., and Sangiovanni-Vincentelli, A. Formal models for embedded system design, *Design & Test of Computers, IEEE* 17, no. 2 (2000).
11. Martin, G., Lavagno, L., and Louis-Guerin, J. Embedded UML: A merger of real-time UML and co-design, *Hardware/Software Codesign, 2001. Proceedings of the Ninth International Symposium on CODES 2001*, April 25–27, Copenhagen, Denmark, 2001.
12. Hu, X. *A Simulation-Based Software Development Methodology for Distributed Real-Time Systems*, Ph.D. Dissertation, Electrical and Computer Engineering Department, Tucson, AZ: University of Arizona, May 2004.
13. Wainer, G. A., and Glinsky, E. Model-Based Development of Embedded Systems with RT-CD++, *Proceedings of the WIP Session, IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, 2004.
14. Shang, H., and Wainer G. A. Dynamic Structure DEVS: Improving the Real-Time Embedded Systems Simulation and Design, *Proceedings of the IEEE 40th Annual Simulation Symposium (ANSS)*. New York: IEEE, 2008.

15. Paillet, J., and Giambiasi N. DECM, A User Oriented Formalism for High Level Discrete Event Specifications of Real-Time Systems, *Journal of Intelligent and Robotic Systems* 34, no. 1 (2002): 27–81.
16. Kim, T. G., Cho, S. M., and Lee, W. B. DEVS Framework for Systems Development: Unified Specification for Logical Analysis, Performance Evaluation and Implementation, *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*. New York: Springer-Verlag, 2001.
17. Shephard, M.S., Beall, M. W., O'Bara, R. M., and Webster, B. E. Toward Simulation-Based Design, *Finite Elements in Analysis and Design* 40 no. 12 (2004): 1575–98.
18. Paredis, C. J. J., Diaz-Calderon, A., Sinha, R., and Khosla, P. K. Composable Models for Simulation-Based Design, *Engineering with Computers* 17, no. 2 (2001): 112–28.
19. Upton, J. *Boeing 777. Airlinertech Series*. North Branch, MN: Specialty Press, 1998.
20. Bullinger, H. J., Breining, R., and Bauer, W. Virtual Prototyping—State of the Art in Product Design, *26th International Conference on Computers & Industrial Engineering*, 103–7. Melbourne, Australia, 1999.
21. Balarin, F., Chiodo, M., Hsieh, H., Tabbara, B., Sangiovanni-Vincentelli, A., Jurecska, A., Lavagno, K., Passerone, C., and Suzuki, K. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Boston, MA: Kluwer Academic Publishers, 1997.
22. Bakshi, A., Prasanna, V. K., and Ledeczi, A. MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems, *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, Snow Bird, Utah, 82–93, 2001.
23. Lee, E. A. *Overview of the Ptolemy Project*, Technical Memorandum No. UCB/ERL M03/25. Berkeley, CA: University of California, 2003.
24. The Ptolemy Project, <http://ptolemy.eecs.berkeley.edu>
25. Simulink 6, <http://www.mathworks.com>
26. Matlab, <http://www.mathworks.com/products/matlab/>
27. Smith, P. F., Prabhu, S. M., and Friedman, J. H. *Best Practices for Establishing a Model-Based Design Culture*, SAE World Congress, April 16–19, Detroit, MI. Warrendale, PA: SAE International, 2007.
28. Hu, X., and Zeigler, B. P. Model Continuity in the Design of Dynamic Distributed Real-Time Systems, *IEEE Transactions On Systems, Man And Cybernetics—Part A: Systems And Humans* 35, no. 6 (2005): 867–78.
29. Hu, X., and Zeigler, B. P. A Simulation-Based Virtual Environment to Study Cooperative Robotic Systems, *Integrated Computer-Aided Engineering (ICAE)* 12, no. 4 (2005): 353–67.
30. Hu, X., Ganapathy, N., and Zeigler, B. P. Robots in the Loop: Supporting an Incremental Simulation-Based Design Process, *IEEE International Conference on Systems, Man, and Cybernetics*, October 10–12, Waikoloa, Hawaii. New York: IEEE, 2005.
31. Hu, X., and Edwards, D. Context-Dependent Structure Control for Adaptive Behavior Selection, *Proceedings of the Workshop on Bio-Inspired Cooperative and Adaptive Behaviours in Robots*, in co-operation with *The Ninth International Conference on the Simulation of Adaptive Behavior (SAB'06)*, September 25–30, Roma, Italy. Berlin, Germany: Springer, 2006.
32. Azarnasab, E., and Hu, X. An Integrated Multi-Robot Test Bed to Support Incremental Simulation-Based Design, *Proceedings of the 2007 IEEE International Conference on System of Systems Engineering (SoSE)*, April 16–18. San Antonio, Tx, New York: IEEE, 2007.

Section III

Applications

11 Formalization for Formal Verification of an Optical Network-on-Chip Using DEVS

*Luiza Gheorghe Iugan, Gabriela Nicolescu,
and Ian O'Connor*

CONTENTS

11.1	Introduction.....	293
11.2	Related Work.....	294
11.3	Optical Network-on-Chip.....	296
11.3.1	Presentation.....	297
11.3.2	Electro-Optical and Opto-Electrical Interfaces.....	297
11.3.3	Passive Optical Network: λ -Router.....	297
11.4	Methodology and Application.....	298
11.4.1	Conversion Interfaces	299
11.4.1.1	Transmitter Architecture.....	299
11.4.1.2	Receiver Architecture	300
11.4.2	Passive Photonic Devices.....	301
11.4.2.1	Point-to-Point Optical Connection.....	301
11.4.2.2	4-Port Optical Switch.....	302
11.4.2.3	The 4×4 λ -Router.....	304
11.5	Summary.....	305
	References.....	305
	Appendix.....	307

11.1 INTRODUCTION

Many of the modern Systems-on-Chip (SoC) integrate a high density of heterogeneous components such as different processors and a wide range of hardware components, as well as complex interconnects that use different communication protocols. On-chip physical interconnections represent a limiting factor for performance and energy consumption. Energy and device reliability impose small logic swings and power supplies. Moreover, the growth of the number of components that are integrated on-chip increases the impact of the deep submicron

effects (e.g., electrical noise due to crosstalk, electromagnetic interference that can produce data errors). By consequence, transmitting data on wires may be in some cases unreliable and nondeterministic [13]. New interconnect challenges are added when moving to 65 nm and beyond: interconnect delay becomes larger than gate delay and the interconnect area becomes much larger than the gate area [13]. Designers also face deep submicron effects like voltage isolation and wave reflection. Optical Networks on Chip (ONoC) are promising because of their scalability, simplicity, and low real estate (0.00425 mm^2 for a passive network) [14]. However, the access to physical prototyping for multitechnology SoCs is a major challenge because of its significant cost and the difficulty to influence standard processes. Modeling and simulation become necessary alternatives in the design space exploration for these systems. A multidisciplinary cooperation between physical and system-level designers is required to achieve accurate design and facilitate the verification step that takes more and more time in the design process [4]. Actually and in the past, formal methodologies were used as initial exploration in the processor domain to correct light errors and defects [14]. Today, in many application designs the most costly task in terms of time and human resources is the design verification. Formal methodologies emerge as a more structured verification approach [3]. This implies that the design model is more thoroughly checked and more cases are taken into consideration.

Discrete Event Systems Specifications (DEVS), defined in Ref. [11] is a mathematical formalism for systems representation and simulations where time advances on a continuous time base. This approach is based on the systems theory: a system with a time base, inputs, states, and outputs. Given the current states and the inputs, functions are implemented to determine the next states and the outputs.

The main contributions of this chapter are: (i) the formalization of optical-electrical interfaces using DEVS and (ii) the formalization of basic elements of an Optical Network-on-Chip (ONoC).

In this chapter we propose a novel approach that enables: (i) the possibility to formalize very recent technologies using DEVS formalism and (ii) the use of this formalization to validate and debug complex systems as optical-electrical interfaces.

The chapter is organized as follows: Section 11.2 presents the related work; Section 11.3 provides basic concepts with DEVS definition as well as optical interfaces and photonic devices; Section 11.4 describes the methodology used in this work with optical network examples, and Section 11.5 gives our conclusions.

11.2 RELATED WORK

A novel approach for the representation of heterogeneous systems is the formal-based description where the integration is addressed as a composition of models of computation (MoC), where some of the heterogeneous components may present timed computation models while others may present untimed computation models. These formal representation approaches propose a single main formalism to represent different models and the main concern is building interfaces between different MoC.

In [8] a formal framework for comparing different MoC used in heterogeneous models is presented. The authors propose a formal classification framework that makes it possible to compare and express the differences between them. The framework was used to compare certain features of various MoCs such as dataflow, sequential processes, and concurrent sequential processes with rendezvous, Petri Nets, and discrete-event systems. The intent is “to be able to compare and contrast its notions of concurrency, communication, and time with those of other MoC.”

An important work in the usage of MoCs in abstracting functionalities of complex heterogeneous systems was done and presented in [5]. The author proposes a classification of the MoC from a denotational view, the main models being: untimed MoCs, synchronous MoCs, and timed MoCs. The untimed MoCs can be characterized by the way the processes communicate and synchronize with other processes in the absence of timing information. The synchronous MoCs “divide the time axis into slots.” Moreover, “everything inside a slot occurs at the same time” and “the evaluation cycle of processes lasts exactly one time slot” [5]. The timed MoC is a generalization of the synchronous MoC.

Discrete Event Systems Specifications (DEVS) [11] is a formalism that supports a full range of dynamic systems representation, with hierarchical and modular model development and more recently it integrates object-oriented programming techniques. The abstraction separates modeling from simulation and provides atomic models that can be used to build complex models that allow the integration of heterogeneous models [11]. It also provides all the mechanisms for the definition of operational semantics for high level representation of heterogeneous models. Several toolkits that enable modeling and simulation based on the DEVS formalism have been developed [1,2,6,7,12]. DEVS was successfully used for prototyping and the testing environment for embedded system design, urban traffic models, and intrusion detection systems or for depot operations modeling but it was not used for the formalization of optical components, opto-electrical interfaces, or optical networks.

The existing work on the representation of heterogeneous systems applies for ONoC. This work can be divided into several classes depending on the abstraction levels.

At a lower level we can find models for physical phenomena of optics using a representation through mathematical equation [15,16].

At a higher abstraction level the models describe the photonic devices behavior respecting a given technology. In [17] the authors used VHDL-AMS language while in [18], in order to model the ONoC at a high abstraction level, SystemC [10] and a bottom-up approach were used. In [9], Chatoyant, a simulation environment based on architecture design at a system level, was used for free space optical interconnects.

The work presented in this paper proposes: (i) a formal model for opto-electrical systems and (ii) a generic representation of the opto-electrical interfaces and their application in a specific case of a system on chip integrating an optical network on chip.

The next section will briefly introduce some basic concepts and definitions regarding the components forming the ONoC.

11.3 OPTICAL NETWORK-ON-CHIP

11.3.1 PRESENTATION

The integrated optical communication system studied in this work, also called ONoC [17], is composed of three types of blocks: (i) a transmitter interface circuit (for the electro-optical conversion), (ii) a passive integrated photonic routing structure (named λ -router; waveguides and wavelength routing structures for interconnection between the transmitter and the receiver), and (iii) a receiver interface circuit (for the opto-electrical conversion).

Figure 11.1 presents an overview of these ONoC plugging initiators and targets (also called cores). The ONoC is a heterogeneous structure that can be represented as a combination of passive and active optical devices as well as mixed analog/digital integrated circuits.

11.3.2 ELECTRO-OPTICAL AND OPTO-ELECTRICAL INTERFACES

Each SoC core (initiator and target) requires a transmitter block that enables the electro-optical conversion (cf. Figure 11.2a). This block is mainly composed of a laser to emit light at a given wavelength and optical power, and its driver for the modulation and polarization.

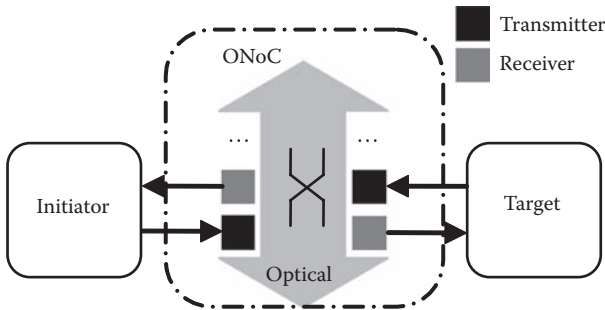


FIGURE 11.1 Optical Network-on-Chip overview.

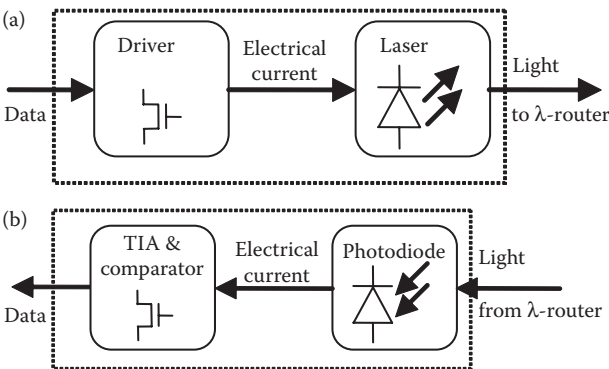


FIGURE 11.2 Transmitter (a) and receiver (b) architecture.

Similar to the transmitter block, each SoC core requires a receiver block that enables the opto-electronic conversion (cf. Figure 11.2b). This block is mainly composed of a photodiode (conversion of flow of photons into a photocurrent), a TransImpedance Amplifier (TIA), and a decision circuit (digital signal regeneration).

11.3.3 PASSIVE OPTICAL NETWORK: λ -ROUTER

The λ -router is a passive optical network (see Figure 11.3a) composed of 4-port optical switches (microresonators) based on add-drop filters (cf. Figure 11.3b) designed to route data through SoC components [14]. The add-drop filters have the capacity to add or subtract a signal from a waveguide, based on its wavelength. The basic function of a microresonator can be seen as a wavelength-controlled switching function. An optical filter is characterized by a specific wavelength, called resonant wavelength (λ_i in Figure 11.3) depending on filter geometry and material. If the wavelength of an optical signal passing through does not correspond to the resonant wavelength, then the electromagnetic field continues to propagate along the waveguide and not through the structure. If, however, the signal wavelength is equal to the resonant wavelength, then the electromagnetic field propagates around the structure and then out along the second waveguide [14].

From any input port, the switching is obtained from one of the two opposite output ports depending on the wavelength value of the optical signal injected into the optical filter (cf. Figure 11.4).

As illustrated in Figure 11.4, there are three possible switch states depending on the input signal:

- The straight state, Figure 11.4a, occurs when resonant wavelengths (λ_i) are injected in the filter and are routed through the optical switch.
- The diagonal state, Figure 11.4b, occurs when other wavelengths (λ_j) are injected in the filter and are not routed through the microresonator.
- The cumulative state, Figure 11.4c, occurs when signals of both resonant and nonresonant wavelengths (λ_i and λ_j) are injected into the filter.

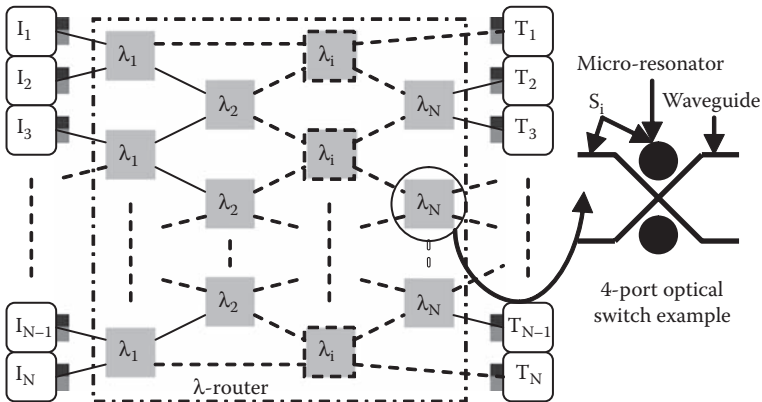


FIGURE 11.3 λ -router overview (a) and optical switch (b).

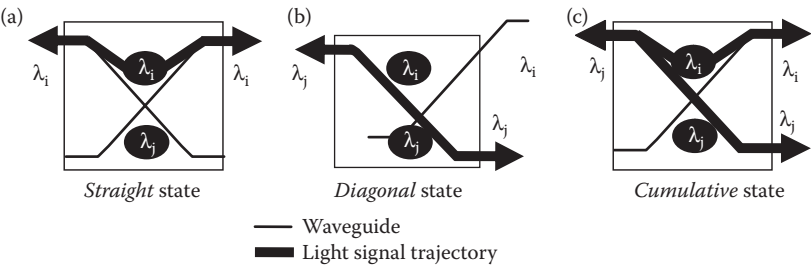


FIGURE 11.4 Functional states of 4-port optical switch.

TABLE 11.1
4 × 4 λ-Router Truth Table

I/T	T ₁	T ₂	T ₃	T ₄
I ₁	λ ₂	λ ₃	λ ₁	λ ₄
I ₂	λ ₃	λ ₄	λ ₂	λ ₁
I ₃	λ ₁	λ ₂	λ ₄	λ ₃
I ₄	λ ₄	λ ₁	λ ₃	λ ₂

In a λ -router, only one physical path associated with one wavelength exists between an initiator I_i (i.e., a processor) and a target T_j (i.e., a memory). The broadcast is also possible with this architecture. Table 11.1 shows the truth table for different operations for a 4×4 network that is a network with four initiators (I_1 to I_4 in Table 11.1) and four targets (T_1 to T_4 in Table 11.1). This system is a fully passive circuit-switching network based on wavelength routing and is a nonblocking network. From I_i to T_j there exists only one physical path associated with one wavelength. For example, if I_2 communicates with T_4 , data must use the wavelength λ_1 to be sent through the λ -router. At the same time I_1 can communicate with T_1 using the wavelength λ_2 .

In order to establish that the formalization is possible, this paper presents only simple optical–electrical interfaces (transmitter and receiver; cf. Section 11.3.1.2) as well as the formalization of the basic elements of a passive photonic device (cf. Section 11.3.1.3).

11.4 METHODOLOGY AND APPLICATION

This section presents the formal representation of opto-electrical, electro-optical interfaces as well as passive photonic devices using DEVS formalism. In order to prove DEVS efficiency for optical components we formally represented only from a behavioral point of view. However, the methodology can be then applied for the design of more complex systems using DEVS coupled models.

- = ('polarization', σ , p , v , $latency\_pol$) if $phase = 'conversion'$ and $p = polarization\_port$ and $v = \{active\}$ or if $phase = 'idle'$ and $p = polarization\_port$ and $v = \{no\_active\}$
- = ('emission', σ , p , v , $latency\_laser$) if $phase = 'conversion'$ and $p = laser\_port$ and $v = wave\_value$ (with power proportional with the modulation current I_m and the polarization current I_p of the laser driver).
- = ('idle', σ , p , v) else.

$$\delta_{ext}(phase, \sigma, e, x): Q \times X \rightarrow S$$

- = ('idle', e , p , v) if $phase = 'idle'$ and $p = activation$ and $v = off$
- = ('busy_active', $proces\_time$, p , v) if $phase = 'conversion'$ and $p = activation$ and $v = on$
- = ('busy_send', $process\_time$, p , v) if $phase = 'conversion'$ and $p = data$ and $v = data\_to\_send$

$$\lambda(phase, \sigma, local\_inport, wave\_value, wave): S \rightarrow Y$$

- = (out , $wave\_value$) if $phase = 'conversion'$ and $local\_inport = laser\_port$
- = (out , $0 \cdot exp(0)$) if $phase = 'idle'$

$$ta(phase, \sigma) = \sigma = latency \mid time\_next\_data$$

$$\text{with } latency = latency\_mod \mid latency\_pol \mid latency\_laser$$

The transmitter presented here receives at the input *data* and *select* and at its outputs the optical signal characterized by a wavelength and its power. Its behavior is characterized by two states: *idle* (no conversion) and *conversion* (data is sent through the interface). There are four possible internal transitions (events): *modulation* (to modulate the laser with the data to convert), *polarization* (to polarize the laser), *light* (for the light emission at a given optical power and wavelength), and *idle* (no light emission). Three external transitions (events) are possible: *idle* (no conversion), *selection* (conversion activation), and *data* (data to convert). The state advancing time is mainly composed of latencies extracted from the physical design (IC) or the datasheet (laser).

11.4.1.2 Receiver Architecture

Figure 11.6 shows the DEVS optical receiver architecture view, including the internal and external events with the Is/Os.

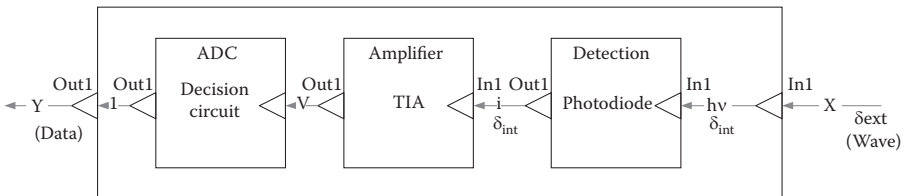


FIGURE 11.6 DEVS optical receiver architecture view.

This system realizes the opto-electrical conversion after the light transport in the optical interconnect (the variations in light intensity are converted to digital signals). This receiver part includes photodetectors that are photodiodes (the “Detection/Photodiode” block in Figure 11.6) to convert the light power variations into a photocurrent, and a TIA (the “Amplifier/TIA” block in Figure 11.6) converts the photocurrent of a few μA into a voltage of a few mV. A comparator subsequently converts this analog signal to a digital rail to rail signal (‘1’ or ‘0’) with a voltage detection threshold. The interface has only one input that is the light named here *wave*. The equations that give the formal representation of the behavior of the optical receiver (opto-electrical conversion) using DEVS formalism are given in Appendix 1. The receiver’s behavior is characterized by two states: *idle* (no conversion) and *conversion* (data is detected through the interface). The behavior of the receiver is not as complicated as the behavior of the transmitter. There are four internal transitions (events): *photoconversion* (for the light conversion in photocurrent), *amplify* (for the amplification of the current and the conversion in voltage), *CAN* (for the analog-to-digital conversion), and *idle* (no light to detect). There are two external transitions (events): *idle* (no conversion) and *data* (light to convert and data sent to the target). The time advancing is mainly composed of latencies extracted from the physical design (IC) or the datasheet (photodiode).

11.4.2 PASSIVE PHOTONIC DEVICES

This section presents two basic photonic devices that form a λ -router: (i) a simple point to point connection (Section 11.4.2.1) and (ii) a basic 4-port optical switch (Section 11.4.2.2). Section 11.4.2.3 shows a 4×4 λ -router using these elementary blocks.

11.4.2.1 Point-to-Point Optical Connection

This section gives the DEVS description of a bidirectional point to point optical connection (see Figure 11.7). A point to point connection can be a straight optical waveguide or a curved one for example. These point-to-point optical links replace electrical global links with optical ones.

The point-to-point connection is bidirectional, therefore there are two inputs. The formal representation of its behavior is simple. The connection is characterized by two states: *idle* (when no light is transported through the optical waveguide) and *communication* (when light is transported through the optical waveguide). There are two internal transitions (events): *busy* (light through the waveguide) and *idle* (no light through the waveguide). There are also two possible external transitions (events):



FIGURE 11.7 DEVS point to point bidirectional optical connection view.

idle (no light at any of the inputs) and *in_light* (light in one of the inputs). The time advancing is due to the light transport in a waveguide depending on its length and its manufacture materials. This DEVS representation must take into account the attenuation in the point to point connection due to its defects (*P2Pdefects*). These defects attenuate the optical power value at the outputs. The equations that give the formal representation of the behavior of the point-to-point connection using DEVS formalism are given in Appendix 2.

11.4.2.2 4-Port Optical Switch

This section shows the DEVS description of a simple 4-port optical switch (called λ -router; shown in Figure 11.8). This component is the most important passive element in the transport component in the network. It is a combination of several basic passive optical devices (like waveguides, add-drop filters) that routes the light through the system on chip.

In Figure 11.8, the add-drop filter (or an optical switch) is made of two microdiscs and two waveguides with a cross. This relatively simple structure is the photonic equivalent of an electronic switch. The operation of this device depends on the wavelength value λ_i of the signal entering at one of the inputs of the add-drop filter (the filter is bidirectional). Signals of wavelength λ_j are routed and the other wavelengths ($\lambda_i \neq \lambda_j$) are not routed through the add-drop but instead continue on through the same waveguide. Given the bidirectionality of the switch, the four ports of the component are inputs and outputs.

The behavior of the 4-port optical switch is characterized by two states: *idle* (when there is no conversion therefore no light is routed through the optical switch) and *communication* (when light is routed through the optical switch, either in the *straight* state or either *diagonal* state as seen in Figure 11.2). There are two possible internal transitions (events): *busy* (when light is present) and *idle* (when there is no light through the switch).

There are two external transitions (events): *idle* (there is no light at any of the input ports) and *in_light* (there is light in at least one of the input ports). The time advancing is due to the light routing in the microresonator and in the waveguide depending on its geometry and its manufacture materials. In the case of the point-

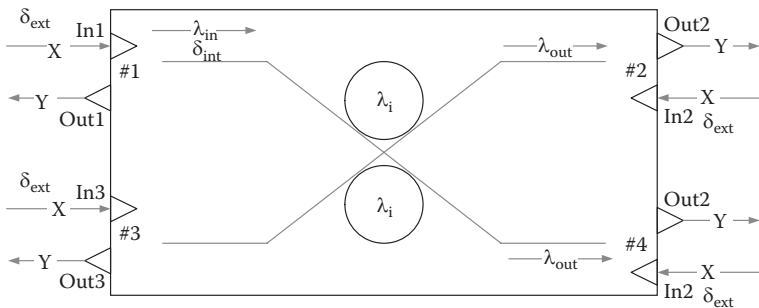


FIGURE 11.8 DEVS optical switch view.

to-point connection, this DEVS representation must take into account the attenuation in the switch due to its defects (*OSdefects*). The following equations give the formal representation of the behavior of the 4-port optical switch using DEVS formalism.

$$DEV_{S_{OS}} = (X, Y, S, \delta_{int}, \lambda, ta)$$

$$\begin{aligned} X &= \{(p,v) | p \in InPorts, v \in X_p\} \text{ with } InPorts = \{'in1', 'in2', 'in3', 'in4'\} \text{ and } \\ &\quad X_p = wave\_value \in \{wavelength, power\} \\ Y &= \{(p,v) | p \in OutPorts, v \in Y_p\} \text{ with } OutPorts = \{'out1', 'out2', 'out3', 'out4'\} \text{ and } \\ &\quad Y_p = wave\_value \in \{wavelength, power\} \\ S &= \{'idle', 'communication'\} \cdot InPorts \end{aligned}$$

$$\delta_{int}(phase, \sigma, inport, wave\_value, wavelength\_OS): S \rightarrow S$$

$$\begin{aligned} &= ('busy', \sigma, p, v) \text{ if } phase = 'communication' \text{ and } p \in InPorts \text{ and } v \in X_p \\ &= ('idle', \sigma, p, v) \text{ else.} \end{aligned}$$

$$\delta_{ext}(phase, \sigma, e, x): Q \times X \rightarrow S$$

$$\begin{aligned} &= ('idle', e, p, v) \text{ if } phase = 'idle' \text{ and } p = wave \text{ and } v = 0 \cdot exp(0) \\ &= ('in\_light', process\_time, p, v) \text{ if } phase = 'communication' \text{ and } p = wave \\ &\quad \text{and } v = wave\_value \cdot OSdefects \end{aligned}$$

$$\lambda(phase, \sigma, inport, wave\_value, wavelength\_OS)$$

$$\begin{aligned} &= (out2, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value = wavelength\_OS \text{ and } inport = in1 \\ &= (out4, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value \neq wavelength\_OS \text{ and } inport = in1 \\ &= (out1, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value = wavelength\_OS \text{ and } inport = in2 \\ &= (out3, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value \neq wavelength\_OS \text{ and } inport = in2 \\ &= (out4, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value = wavelength\_OS \text{ and } inport = in3 \\ &= (out2, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value \neq wavelength\_OS \text{ and } inport = in3 \\ &= (out3, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value = wavelength\_OS \text{ and } inport = in4 \\ &= (out1, wave\_value \cdot OSdefects) \text{ if } phase = 'communication' \text{ and } wave\_ \\ &\quad wavelength\_value \neq wavelength\_OS \text{ and } inport = in4 \\ &= (out, 'X') \text{ if } phase = 'idle' \text{ with } out \in OutPorts \end{aligned}$$

$$ta(\sigma) = \sigma = bit\_propagation\_time$$

Figure 11.9 presents the state flow of a 4-port optical switch. This diagram takes into account the DEVS events seen previously.

11.4.2.3 The 4 × 4 λ-Router

Figure 11.10 shows the representation using DEVS of a 4 × 4 optical network that could be used in a SoC with eight IP blocks: four initiators (ports #1, #3, #5, and #7) and four targets (ports #2, #4, #6, and #8). Considering that the switch is bidirectional, the four initiators can also be targets and the four targets can be initiators. Moreover, this structure verifies the truth table presented in Table 11.1. Figure 11.11 presents the state flow of the 4 × 4 λ-router behavior. *Ci* represents the connection between any input ports with the *i*th output port.

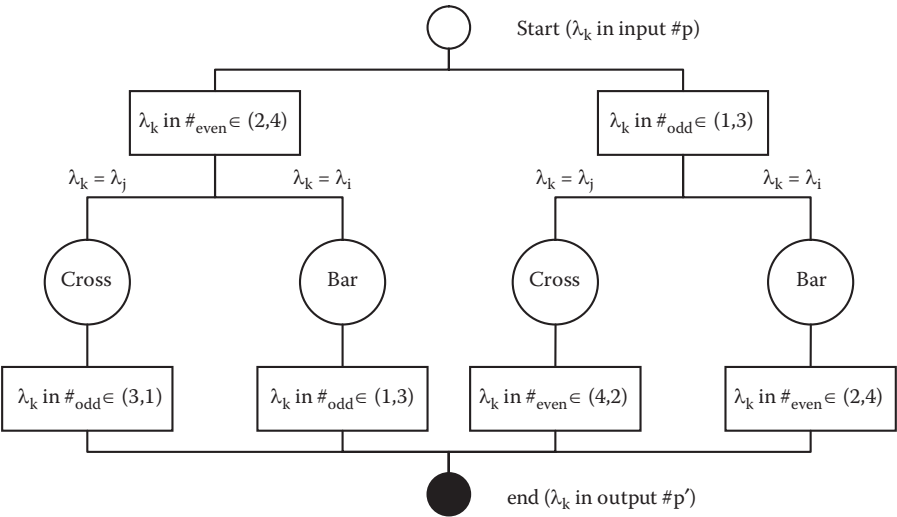


FIGURE 11.9 State diagram of a 4-port optical switch.

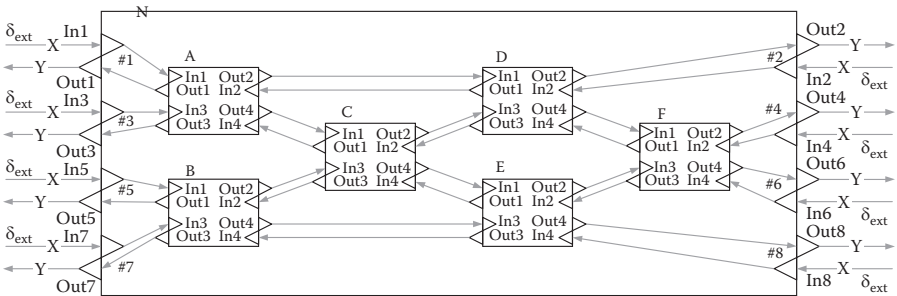
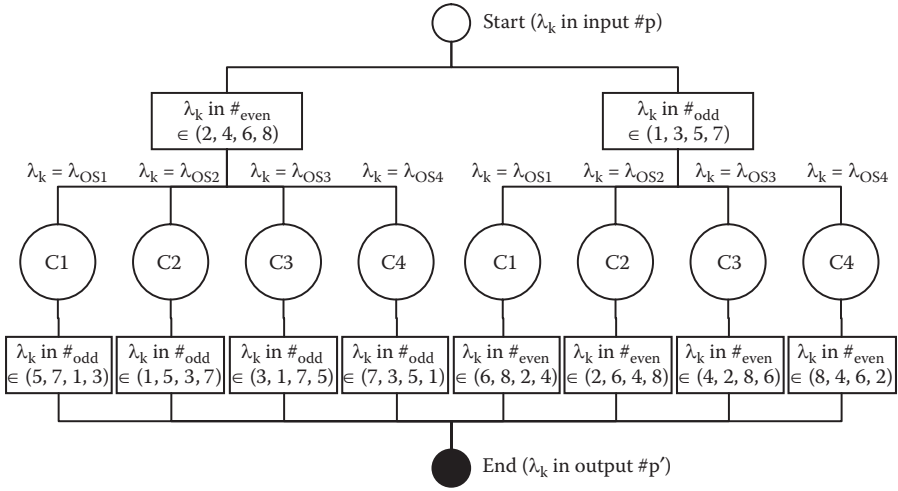


FIGURE 11.10 DEVS 4 × 4 λ-router view.


 FIGURE 11.11 State diagram of a 4×4 λ -router.

11.5 SUMMARY

In this chapter we present the formalization of opto-electrical and electro-optical interfaces and basic passive elements of an ONoC using the DEVS formalism. This formalization allows a quicker design exploration, in terms of network configuration and architecture. This work represents the first step in the formal representation of a complete ONoC. The next step is the formal verification for the validation of the network, using different existing tools.

REFERENCES

1. Bolduc, J.-S., and H. Vangheluwe. "The modelling and simulation package PythonDEVS for classical hierarchical devs." MSDL Technical Report MSDL-TR-2001-01. Montreal, Quebec, Canada: McGill University, 2001.
2. D'Abreu, M., and G. Wainer. "M/CD ++: Modeling continuous systems using Modelica and DEVS," *Proceedings of the IEEE International Symposium of MASCOTS'05*, 2005.
3. Frey, P., and D. O'Riordan. "Verilog-AMS: Mixed-signal simulation and cross domain connect modules," *Proceedings of the BMAS International Workshop*, Orlando, FL, 2000.
4. *International Technology Roadmap for Semiconductor Design*. [Online]. Available at: <http://public.itrs.net/>
5. Jantsch, A. *Modeling Embedded Systems and SoCs—Concurrency and Time in Models of Computation*. Systems on Silicon Series. San Francisco, CA: Morgan Kaufmann Publishers, 2003.
6. Kim, T. G. *DEVSIM ++ User's Manual*, SMSLab, Department of EECS, KAIST, 1994, Available at: <http://smslab.kaist.ac.kr>
7. Kim, Y. J., J. H. Kim, and T. G. Kim. "Heterogeneous simulation Framework using DEVS-BUS," *Simulation, the Society for Modeling and Simulation International* 79, no. 1 (2003): 3–18.

8. Lee, E. A., and A. L. Sangiovanni-Vincentelli. "Comparing models of computation," *IEEE Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 234–41, 1996.
9. Levitan, S., J. Martinez, T. Kurzveg, P. Marchand, and D. Chiarulli. "Multi technology system-level simulation," in *Design Test Integration and Packaging of MEMS/MOEMS (DTIP 2000)*. New York: IEEE Computer Society, 2000.
10. *SystemC LRM* [Online]. Available at : www.systemc.org
11. Zeigler, B. P., H. Praehofer, and T. G. Kim. *Modeling and Simulation—Integrating Discrete Event and Continuous Complex Dynamic Systems*. San Diego, CA: Academic Press, 2000.
12. Wainer, G. "Modeling and simulation of complex systems with Cell-DEVS," *Winter Simulation Conference*, 49–60, 2004.
13. Benini, L., and G. Di Micheli. "Networks on chips: A new SoC paradigm," *Computer* 35, no. 1 (2002): 70–77.
14. O'Connor, I. "Optical solutions for system-level interconnect," *Proceedings of the International Workshop on SLIP*, Paris, France, 2004.
15. *Intel Technology Journal*. Available at: <http://www.intel.com/technology/itj/archive/2004.htm>
16. Senturia, S. *Microsystem Design*. New York: Springer, 2004.
17. M. Brière, L. Carrel, T. Michalke, F. Mieyeville, I. O'Connor, and F. Gaffiot. "Design and behavioral modeling tools for optical network-on-chip," *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2004.
18. Brière, E. Drouard, F. Mieyeville, D. Navarro, I. O'Connor, and F. Gaffiot. "Heterogeneous modeling of an optical network-on-chip with SystemC," *Proceedings of the Rapid System Prototyping (RSP)*, 2005.

APPENDIX

APPENDIX 1: FORMAL REPRESENTATION OF THE BEHAVIOR OF THE OPTICAL RECEIVER (OPTO-ELECTRICAL CONVERSION) USING DEVS FORMALISM

```

DEVSRX = (X, Y, S,  $\delta_{ext}$ ,  $\delta_{int}$ ,  $\lambda$ , ta)
X = {(p,v) | p ∈ InPorts, v ∈ Xp} with InPorts = {'wave'}
and Xp = wave_value ∈ {wavelength, power}
Y = {(p,v) | p ∈ OutPorts, v ∈ Yp} with OutPorts = {'data'}
and Yp = {data_to_receive}
S = {'idle', 'conversion'}
 $\delta_{int}$ (phase,  $\sigma$ , local_inport, local_value, inport,
value)
    = ('detection',  $\sigma$ , p, v, latency_pdiode) if
phase = 'conversion' and p = pdiode_port and v =
wave_value
    = ('amplifier',  $\sigma$ , p, v, latency_TIA) if phase
= 'conversion' and p = TIA_port and v = photocurrent
    = ('ADC',  $\sigma$ , p, v, latency_ADC) if phase =
'conversion' and p = ADC_port and v =
photocurrent.gain
    = ('idle',  $\sigma$ , p, v) else.
 $\delta_{ext}$ (phase,  $\sigma$ , e, x)
    = ('idle', e, p, v) if phase = 'idle' and p =
wave and v = 0.exp(0)
    = ('busy_receive', process_time, p, v) if
phase = 'conversion' and p = wave and v = wave_value
 $\lambda$ (phase,  $\sigma$ , local_inport)
    = (out, data_to_receive) if phase =
'conversion' and data_to_receive = bit_value and
local_inport = ADC_port
    = (out, 'X') if phase = 'idle'
ta(phase,  $\sigma$ ) =  $\sigma$  = latency
    with latency = latency_pdiode | latency_TIA
| latency_ADC

```

APPENDIX 2: FORMAL REPRESENTATION OF THE BEHAVIOR OF A POINT TO POINT OPTICAL CONNECTION USING DEVS FORMALISM

```

DEVSP2P = (X, Y, S,  $\delta_{int}$ ,  $\lambda$ , ta)
X = {(p,v) | p ∈ InPorts, v ∈ Xp} with InPorts = {'in1',
'in2'} and Xp = wave_value ∈ {wavelength, power}
Y = {(p,v) | p ∈ OutPorts, v ∈ Yp} with OutPorts = {'out1',
'out2'} and Yp = wave_value ∈ {wavelength, power}
S = {'idle', 'communication'}
 $\delta_{int}$ (phase,  $\sigma$ , inport, wave_value)
    = ('busy',  $\sigma$ , p, v) if phase = 'communication'
    and p ∈ InPorts and v ∈ Xp
    = ('idle',  $\sigma$ , p, v) else.
 $\delta_{ext}$ (phase,  $\sigma$ , e, x)
    = ('idle', e, p, v) if phase = 'idle' and p =
    wave and v = 0.exp(0)
    = ('in_light', process_time, p, v) if phase =
    'communication' and p = wave and v = wave_
    value.P2Pdefects
 $\lambda$ (phase,  $\sigma$ , inport, wave_value): S → Y
    = (out2, wave_value.P2Pdefects) if phase =
    'communication' and inport = in1
    = (out1, wave_value.P2Pdefects) if phase =
    'communication' and inport = in2
ta( $\sigma$ ) =  $\sigma$  = bit_propagation_time

```

12 Construction Simulation Using Cell-DEVS Modeling

Amin Hammad, Hong Pang, and Cheng Zhang

CONTENTS

12.1	Introduction.....	309
12.2	Procedure of Cell-DEVS Simulation	311
12.2.1	Preprocessing.....	312
12.2.2	Main Processing.....	314
12.2.3	Postprocessing.....	315
12.3	Case Studies	315
12.3.1	Case Study I—Jacques Cartier Bridge Rehabilitation Project	315
12.3.2	Case Study II—Analysis of Effect of Work Zones on Traffic Flow.....	324
12.3.2.1	Developing the Cell-Based Simulation Model.....	325
12.3.2.2	Visualizing and Analyzing the Simulation Results	326
12.4	Conclusions and Discussion.....	328
	Acknowledgments.....	329
	References.....	329

12.1 INTRODUCTION

Construction operations of buildings, bridges, and so on, are inherently complex with many uncertainties where each activity has its own variables, and a large number of factors affect them during the construction processes. Simulation has been used as an effective means to analyze and design construction operations for more than three decades. The objective of simulation is to determine the impact of a change of an input on the entire system or on local parts of the system. CYCLONE [1], STROBOSCOPE [2], and Symphony [3] are three representative systems not only because of their widespread use in the construction industry and their distinct characteristics, but also because they have inspired the development of many other simulation systems. However, these construction simulation tools typically cannot illustrate the modeled processes graphically, but provide results in

the form of numerical or statistical data, which poses difficulty in communicating the results, in debugging models during the development stage, and in validating the simulation results.

Visualization provides valuable insights into subtleties of the simulated construction operations that are otherwise nonquantifiable and nonpresentable [4]. Visualizing simulated construction operations, acting as the postprocessor of construction simulation, is an effective means for checking errors in the coding (verification). On the other hand, visualization can help discover differences between the way expert engineers understand the simulated operation and the way the model developer understands it (validation). Thus, the model, coupled with verification and validation, can be communicated effectively and attains credible utility in the construction industry [4]. As such, visualizing simulated construction operations, or construction animation in 2-D or 3-D, has become a necessary supplement of construction simulation.

Recently, researchers have been more concerned about workspace conflict detection using simulation. Much research addressed the impact of spatial conflicts in construction operations [5]. Ref.[6] mentioned that workspace conflicts are one of the important problems that can delay construction activities, reduce productivity, or cause accidents that threaten the safety of workers. In other work, simulation has been utilized to decide the layout planning of seaports [7] and warehouse layout planning [8,9]. Ref. [10] used simulation to develop a vent simulation model to optimize stockyard layouts for precast concrete products and to evaluate different layout scenarios. Ref. [11] integrated simulation with genetic algorithms for the evaluation and optimization of construction site layouts. Ref. [12] simulated asphalt paving operations under lane closure conditions. Ref. [4] proposed detecting conflicts between any pair of mobile or static objects on a construction site based on collision detection methods implemented within visualization tools of discrete-event simulators. However, because of the complex characteristics of space (e.g., size, shape, dynamic changes, and other uncertainties), it is very difficult to describe space explicitly.

Based on the literature review, we can conclude that available construction simulation tools can provide the logical relationships between the different resources; however, they are not suited to define the spatial relationships between resources on a specific construction site. This could result in spatial constraints being ignored in the simulation, and the simulation result not reflecting the actual situation of the construction site. Therefore, construction simulation models should be built in a way that the space can be represented explicitly, which makes the model more representative of the actual construction environment, resulting in a more realistic simulation.

In this chapter, a new cell-based construction simulation method using the Cell-DEVS [13] simulation methodology is proposed, which allows the consideration of the spatial constraints of a site and to perform a sensitivity analysis with different site layout patterns.

In previous work, we have investigated the applicability of the cell-based simulation approach in construction [14,15]. This chapter builds on the previous research

of using the Cell-DEVS simulation method to explicitly represent, analyze, and visualize spatial resource allocation, and site layouts, as well as the movement of equipment. Spatial conflicts can be detected and resolved using predefined rules. The simulation results can be compared based on different site layout patterns and resource combinations. Sensitivity analysis is applied to further investigate the possibility of using cell-based modeling in resource optimization. Cell-DEVS and MicroCYCLONE models are compared to show the feasibility of the proposed method. Two case studies about bridge deck rehabilitation and road asphalt pavement construction are presented to illustrate the usability of the proposed method.

12.2 PROCEDURE OF CELL-DEVS SIMULATION

Cell-DEVS modeling is a general purpose simulation tool, and its application is not limited to any process or resource-based simulation. In order to apply this modeling to construction operations, a specific procedure is needed to guide the development of cell-based simulation models.

The procedure of cell-based simulation is shown in Figure 12.1. It consists of three phases: Preprocessing, Main-processing, and Postprocessing. Preprocessing focuses on model building. Main-processing focuses on deciding site layout patterns and resource combinations, and running the simulation. Postprocessing includes analyzing, verifying and validating, as well as visualizing the simulation results. Next, the steps of the general procedure are described using a generic example. Two specific case studies are explained in detail in Section 12.3.

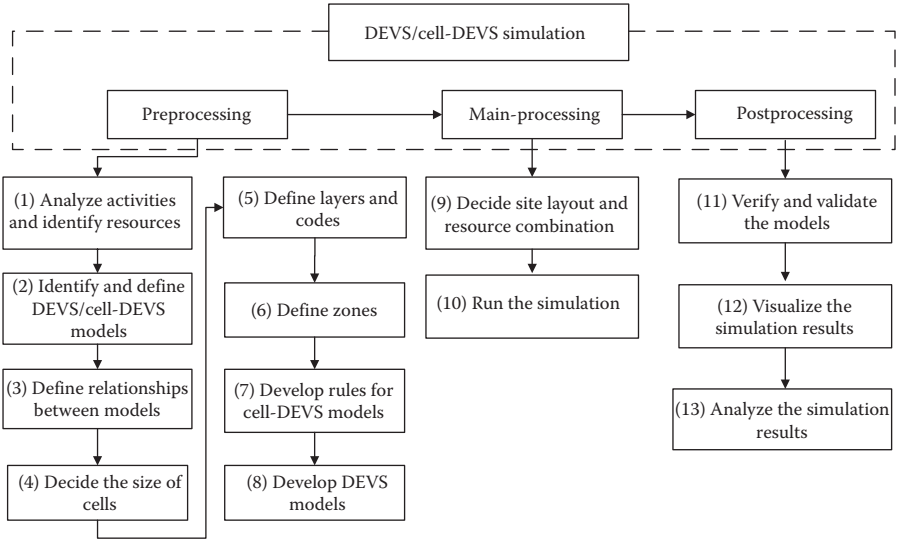


FIGURE 12.1 The process of the cell-based simulation.

12.2.1 PREPROCESSING

Preprocessing includes the following steps:

1. Analyzing Construction Activities and Identifying Involved Resources

As the first step, the development of a construction simulation model requires breaking the construction operations down to recognizable work *tasks* and identifying the construction *resources* involved in the operations.

2. Identifying and Defining DEVS and Cell-DEVS Models

The second step is to define models to represent construction *operations*. Two types of models are used in cell-based simulation—Cell-DEVS and DEVS models. Since Cell-DEVS models provide a virtual cell space to represent the real worksite, they are employed where the spatial representation is important. DEVS models are employed to simulate processes that cannot be represented by cells or when spatial representation is not necessary. For example, suppose that a construction is performed on *Work Area-1* and *Work Area-2* as shown in Figure 12.2, the spatial representation of the work areas is important because most spatial conflicts occur on them and every detail of construction work spaces should be visualized. As a result, these work areas should be defined as Cell-DEVS models. *Control Unit* and *Queues* can only be defined as DEVS models. *Resource Servers* can be defined as either of the two types depending on the severity of the spatial conflicts. Each model might be composed of several submodels, forming different hierarchical levels.

3. Defining Relationships between Models

Input/output ports of models should be defined to exchange messages and build links between models. For example, in Figure 12.2, suppose that each work area represents a road section with three lanes and the vehicles could enter through one of the three lanes, three links can be defined to send and receive messages to indicate the movement of the vehicles into the work area.

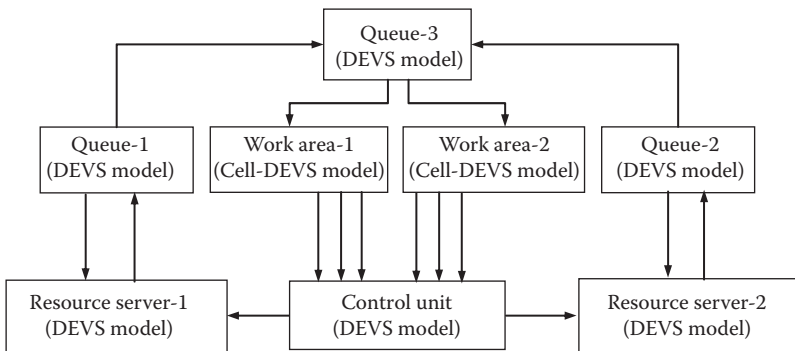


FIGURE 12.2 Abstract example of a cell-based model.

4. Deciding the Suitable Size of Cells and the Dimension of Each Cell-DEVS Model

The size of a cell should be decided considering the following factors:

- The size of the smallest object to be represented: Because the cell-based simulation method can only represent objects with equal cells, it is difficult to represent an object with specific shapes and dimensions, such as a truck that is 12 m long and 3 m wide. In this case, a combination of several cells (called a “block”) could be applied to represent the space occupied by an object.
- The trade-off between the accuracy and the computing time: The smaller the size of the cells, the more accurate the results and the more complex the rules in order to cover the extended neighborhood.
- The time needed for an object to pass through a cell: For instance, if a truck moves at the speed of 10 km/h and the size of the cell is 3 m $\times$ 3 m, it takes about 1 second for the truck to go from one cell to the next.

5. Defining the Layers and Codes for Each Cell-DEVS Model

The cell space can be taken as an N-dimensional space and its attributes are distributed on N layers, where each layer represents one attribute. For example, it may be necessary to indicate if a cell is occupied by a specific type of object (e.g., truck, crane, or other types of equipment), the movement of that object (e.g., static, moving, and the moving direction), and its ID. Therefore, three layers should be defined to describe the above three attributes. Predefined codes are assigned to the cells of each layer to indicate these attributes.

6. Defining the Zones of Each Cell-DEVS Model

A zone defines a region of the cellular space that will use a different local transition function. By default, the rules of a Cell-DEVS model apply to all the cells of the model. However, in some cases, it is necessary to specify a part of the cell as a zone where different local rules apply. For example, in a road pavement project, one lane is closed to perform the construction and the other lanes can still be used for traffic. Zones reserved for the pavement and for the traffic could be defined to apply different sets of rules to simulate the paving operation and traffic flow within one cellular space.

7. Developing Rules for Each Cell-DEVS Model

Rules are developed to check the states of the reference cell's neighbors by the communication between cells and defining the change of the reference cell's state. Each rule consists of three elements: a *condition*, a *delay*, and a *result* with the format {Result} Delay {Condition}. If the {Condition} is satisfied, its {Result}, and its Delay are evaluated, and the {Result} will be assigned to the cell after the Delay.

Rules are also used to control the movement of mobile objects. For example, if the code “5” indicates a “static” object that is not moving throughout the task duration, it can be regarded as an indication of being an obstacle for other moving objects. When a moving object meets a static object with code “5,” it needs to change its direction of movement to get around the

obstacle. In other cases, spatial conflicts among moving objects may occur. For example, when an object is moving, other moving objects may become obstacles and conflicts among moving objects will occur, causing some of them to stop and to assigning priority of movement to the others. Therefore, defining a set of priority rules to detect and resolve these conflicts is necessary. Furthermore, in some cases, multiple cells are required to represent an object for a more accurate description. Thus, more rules have to be defined for those cells since each cell is independent from the others. To reduce the effort of developing rules, the master-slave concept is provided to assign one cell of an object as the master cell and the others as slave cells that follow the same actions as the master cell during simulation. In this way, the model developer can focus on the behavior of the master cell and the slave cells will follow each other one by one. Detailed examples about rules can be found in Case Study 1.

Note that reusability is an important factor in defining rules. Each action is performed by a set of rules and these rules can be grouped into a “macro” that can be called to perform a specific action.

8. Developing DEVS Models

DEVS models are important components in cell-based construction simulation. The external events of a DEVS model are collected through input ports and the *external transition function* defines how to react to such inputs. The results are distributed through output ports by activating the *output function*. These functions are defined for each DEVS model by programming. The programmability makes these DEVS models very flexible, and the validated models can be reused in other models with minor modifications.

12.2.2 MAIN PROCESSING

Main processing includes the following steps.

9. Deciding Site Layout and Resource Combinations

The resource allocation in cell-based simulation consists of specifying the resource quantity and position. The key factor in investigating different site layouts is to develop a mechanism that connects the facilities and moving objects. For example, a truck can be sent to a target position next to its corresponding team. Various resource combinations can be created and observed in order to determine the system response and the interaction between resources. All resources should be placed at their initial positions before running the simulation.

10. Running the Simulation

A cell-based simulation model defines all the submodels, their relationships, and the output ports. Any state changes of these output ports will be recorded in an output file. A log file can also be generated to record all of the messages sent between models.

12.2.3 POSTPROCESSING

Postprocessing includes the following steps:

11. Verification and Validation of the Models

Cell-based simulation provides several output files to verify and test the simulation results. These files can be used to depict the simulation state from different perspectives and at different levels of detail, as well as to debug errors.

12. Visualizing the Simulation Results

The proposed cell-based simulation method integrates an animation tool that can read the output files and visualize the simulation states of Cell-DEVS models in 2-D.

13. Analyzing the Simulation Results

The simulation results can be analyzed from multiple perspectives according to the requirements, productivity, spatiotemporal conflicts, and so on. In addition to the conventional sensitivity analysis realized by changing one or more variables while keeping the others constant, the delays that result from spatial conflicts under different site layout patterns can also be identified and quantified. The first case study will demonstrate how a search for a near-optimal combination can be conducted by generating many resource combinations and applying the cell-based simulation.

12.3 CASE STUDIES

The following two case studies illustrate the application of the procedure of cell-DEVS simulation to a bridge rehabilitation project and road maintenance work zones.

12.3.1 CASE STUDY I—JACQUES CARTIER BRIDGE REHABILITATION PROJECT

The Jacques Cartier Bridge redecking project is used as a case study to develop two simulation models, one using MicroCYCLONE and the other using the Cell-DEVS method. The development of the MicroCYCLONE model is used for comparison and is described in [15]. The deck of the Jacques Cartier Bridge was replaced in 2001–2002 [16]. This case study focuses on the two main activities of removing existing deck sections and installing new panels on the main span of the bridge. The existing deck was removed by saw-cutting the deck into sections similar in dimensions to the new panels. The old sections were removed and replaced with new prefabricated panels by two telescopic cranes. The old sections were transported to a dumping area near the bridge, and the new panels were transported from the plant located at the south end of the bridge. The following paragraphs describe the Cell-DEVS model development according to the steps explained in Figure 12.1.

1. Analyzing Construction Activities and Identifying Involved Resources

The construction activities were abstracted into several sequential processes:

- (1) At the beginning of the simulation, saws are used to cut the deck into small sections and a work team including two cranes was installed beside

each old section; (2) An empty truck for carrying old sections (OS Truck) moves on the bridge, stops to load old sections removed by a work team, and goes off the bridge to the dump area; (3) A truck loaded with a new panel (NP Truck) moves on the bridge, stops to unload at the place where the old section has been removed, and then goes back to the plant; (4) The work team moves to the next position and starts another cycle of activities; (5) The OS truck is unloaded at the dump area and then sent back to the bridge; and (6) The empty NP truck is reloaded with a new panel at the plant and sent back to the bridge. Based on the above analysis, the resources involved in the processes are: Teams, Saws, OS Trucks, and NP Trucks.

2. Identifying and Defining DEVS and Cell-DEVS Models

In this case study, the *Bridge* model is of the most importance for spatial representation because it has space constraints and most of the spatial conflicts are expected to occur on the bridge. Therefore, the *Bridge* is defined as a Cell-DEVS model. Other models, such as *Controller*, *Delay-Queue*, *OS Queue*, *NP Queue*, *Dump Area*, and *Plant*, are defined as DEVS models as shown in Figure 12.3. The functions of each model are briefly explained in the following and more detail can be found in [17]. A summary of these models and task durations are presented in Table 12.1. The duration of each task follows a normal distribution according to the data collected from the project.

- **Controller.** The *Controller* controls the receiving and dispatching of OS trucks and NP trucks. It has three submodels. *Saw* is used to cut the

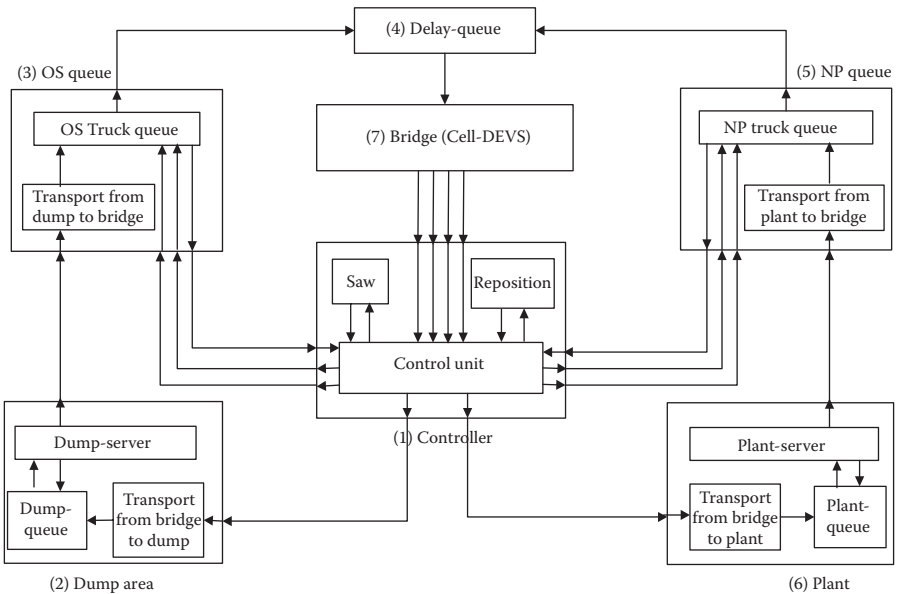


FIGURE 12.3 Cell-based model of the Jacques Cartier Bridge Rehabilitation Project.

TABLE 12.1
Summary of Models of Case Study I

Model Type	Coupled Model	Atomic Model	Model Description	Task Duration	
				Mean (min.)	Std. Deviation (min.)
DEVS	Controller	<i>Saw</i>	Cuts old sections	18	3
		<i>Reposition</i>	Team repositioning	18	3
		<i>Control Unit</i>	Controls and coordinates the operations		N.A.
	Dump Area	<i>Dump-server</i>	Dumps old sections	5	2
		<i>Transport from Bridge to Dump</i>	Loaded OS truck travels from the bridge to the dumping area	7	2
		<i>Dump-queue</i>	Stores loaded OS trucks		N.A.
	OS Queue	<i>Transport from Dump to Bridge</i>	Empty OS truck returns from the dumping area to the bridge	5	2
		<i>OS Truck Queue</i>	Stores OS trucks available to be sent to the bridge		N.A.
	Plant	<i>Plant-server</i>	Loads new panels	14	4
		<i>Transport from Bridge to Plant</i>	Empty NP truck returns from the bridge to the plant	5	1
		<i>Plant-queue</i>	Stores loaded NP trucks		N.A.
	NP Queue	<i>Transport from Plant to Bridge</i>	Loaded NP truck travels from the plant to the bridge	7	1
		<i>NP Truck Queue</i>	Stores NP trucks available to be sent to the bridge		N.A.
	N.A.	<i>Delay-queue</i>	Separates close trucks		N.A.
Cell-DEVS	<i>Bridge</i>	<i>Each cell</i>	Removes old sections	15	5
			Installs new panels	26	3

bridge deck into small old sections. *Reposition* represents a time delay for the reposition of work teams. *Control Unit* is the main part of the system that controls and coordinates the operations of other models.

- **Dump Area.** *Dump Area* is a coupled DEVS model. *Dump-server* is a workstation where OS trucks loaded with old sections are unloaded by a forklift. *Dump-queue* stores loaded OS trucks from the submodel *Transport from Bridge to Dump* in the order of their arrival. *Transport from Bridge to Dump* represents the duration for OS trucks traveling from *Bridge* to *Dump Area*.

- **OS Queue.** *OS Queue* is a coupled DEVS model that consists of two submodels. *Transport from Dump to Bridge* represents the duration for trucks traveling from *Dump Area* to *Bridge*. *OS Truck Queue* receives OS trucks from *Transport from Dump to Bridge* and stores them in a queue. The *Control Unit* determines whether or not this truck can be sent out under the current working conditions on the bridge.
- **Delay-Queue.** *Delay-Queue* is an atomic DEVS model. We assume that there is only one lane open for trucks to access the bridge. *Delay-Queue* is used to check and keep an interval between trucks for safety purpose.
- **NP Queue.** The *NP Queue* model is similar to *OS Queue*. The main difference is that it is used specifically for the dispatching of NP trucks.
- **Plant.** *Plant* is a coupled DEVS model that is similar to *Dump Area*.
- **Bridge.** The *Bridge* model represents the real bridge workspace where the crews remove old sections and install new panels, and where trucks travel and stop for loading/unloading. This model provides a workspace for equipment and shows the location of resources, as well as detects and resolves spatial conflicts.

3. Defining Relationships between Models

The arrows in Figure 12.3 show the directions in which messages are sent. For example, on the bridge there are six lanes. It is assumed that the two side lanes are reserved for an emergency and the other four lanes are used for hauling. Therefore, four output ports in the *Bridge* model and four input ports in the *Controller* should be defined.

4. Deciding the Suitable Size of Cells and the Dimensions of Each Cell-DEVS Model

In this case study, the size of cells is taken as 3×3 m. The deck of the main span of the bridge is approximately represented by 200×6 cells. A crane is represented by 1×3 cells located beside an old section that is also represented by 1×3 cells. A truck is represented by 4×1 cells as shown in Figure 12.5.

5. Defining the Layers and Codes for Each Cell-DEVS Model

This case study adopts the three-layer structure for the *Bridge* model: the *Occupancy Layer*, the *Control Layer*, and the *ID Layer* as shown in Figure 12.4. The *Occupancy Layer* has the attribute of the equipment type occupying a cell. For example, code “8” means that an NP truck carrying new panels is occupying this cell. The *Control Layer* controls the movements of mobile objects. Codes in this layer indicate the moving state of an object (moving or static) and direction of movement (north, south, east, or west). The *ID Layer* contains the ID number of each piece of equipment.

6. Defining the Zones of Each Cell-DEVS Model

The *Bridge* is divided into two zones (zone-1 for the upper three lanes and zone-2 for the lower three lanes). The major difference of the rules applied to these two zones is that a truck moves in different directions when it meets an obstacle. For example, when a spatial conflict between a truck and a

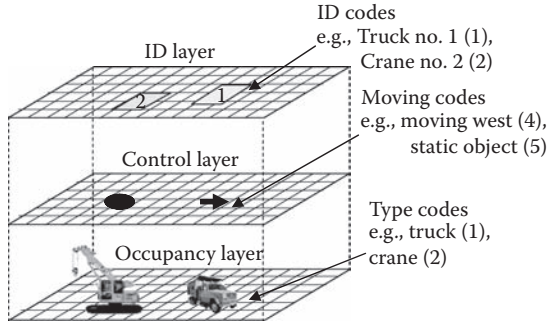


FIGURE 12.4 Representation of layers.

work team occurs, the direction of the truck will change to south or north to avoid the obstacle depending on whether the truck is in zone-1 or zone-2, respectively.

7. Developing Rules for Each Cell-DEVS Model

In this case study, two types of rules are defined. The first type is the regular rules, which control truck movements or turning north and south under regular conditions (no conflicts). Regular rules can be easily reused for other projects. The other type is the priority rules, which regulate the movement of objects when conflicts occur. To resolve conflicts, the following priority rules are defined in this case study: (1) When two trucks are moving on the same lane, the interval between them is at least two cells; otherwise, the latter has to stop temporarily, yielding priority to the former; (2) When two trucks are moving on different lanes, if the truck on the upper lane is behind the truck on the lower lane but they have an overlap, the truck on the upper lane has priority; (3) When two trucks are moving on different lanes, if the truck on the upper lane is behind the truck on the lower lane without any overlap, the truck on the lower lane has priority; and (4) When a truck is taking priority to turn, other mobile objects that may cause spatial conflicts with this truck have to keep their positions until the truck leaves.

8. Developing DEVS Models

These DEVS models are implemented by programming [17].

9. Deciding Site Layout and Resource Combinations

In this case study, three representative patterns of the site layout were chosen as shown in Figure 12.5. In Pattern-A, teams are located on one side of the bridge and their ID numbers are in ascending order. In Pattern-B, teams are on both sides of the bridge in ascending order, while in Pattern-C, teams are on both sides of the bridge in descending order. When a truck is moving from west to east, it checks if it is passing a team. If this team is its corresponding team, it will stop for loading old sections or unloading new panels; otherwise, it will keep moving. In this way, different site layout patterns and the paths of movement of trucks can be considered.

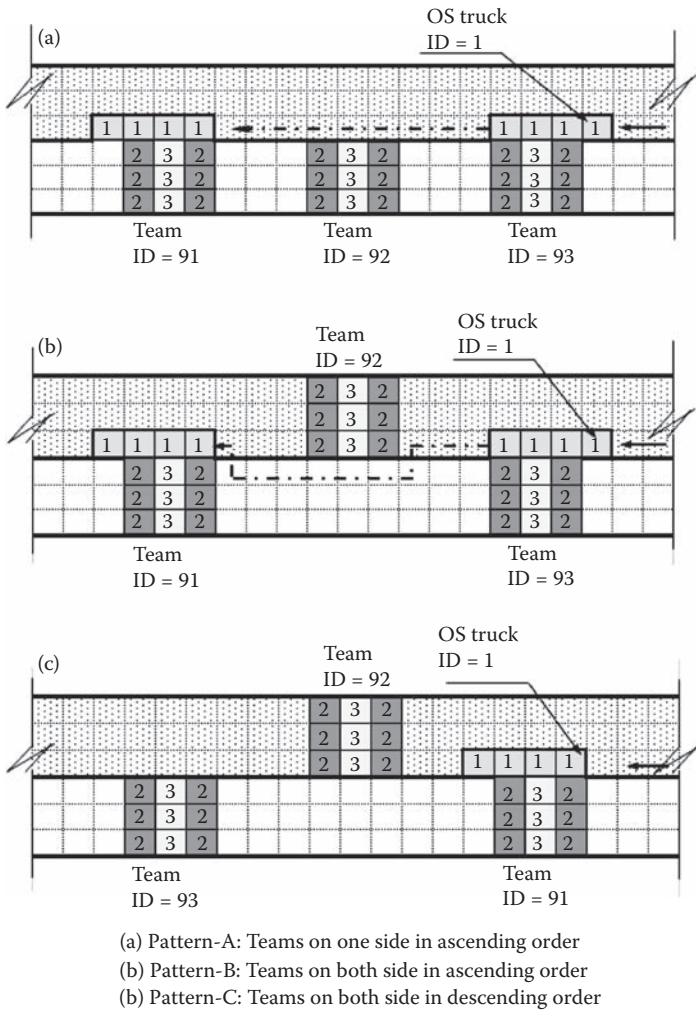


FIGURE 12.5 Patterns of site layout. (a) Pattern-A: teams on one side in ascending order, (b) Pattern-B: teams on both sides in ascending order, and (c) Pattern-C: teams on both sides in descending order.

10. Running the Simulation

A large number of cases has been developed to consider the different resource combinations and site layout patterns as will be discussed in step (13) below. The simulation was done using CD ++ [13].

11. Verification and Validation of the Models

The process of verification is exercised through examining the output files. In order to show the simulation results, a small model with a space of 6×60 cells is used. The resource combination of TSON 5235 (5 teams, 2 saws, 3 OS trucks, and 5 NP trucks) and the site layout Pattern-B are shown in Figure 12.6.

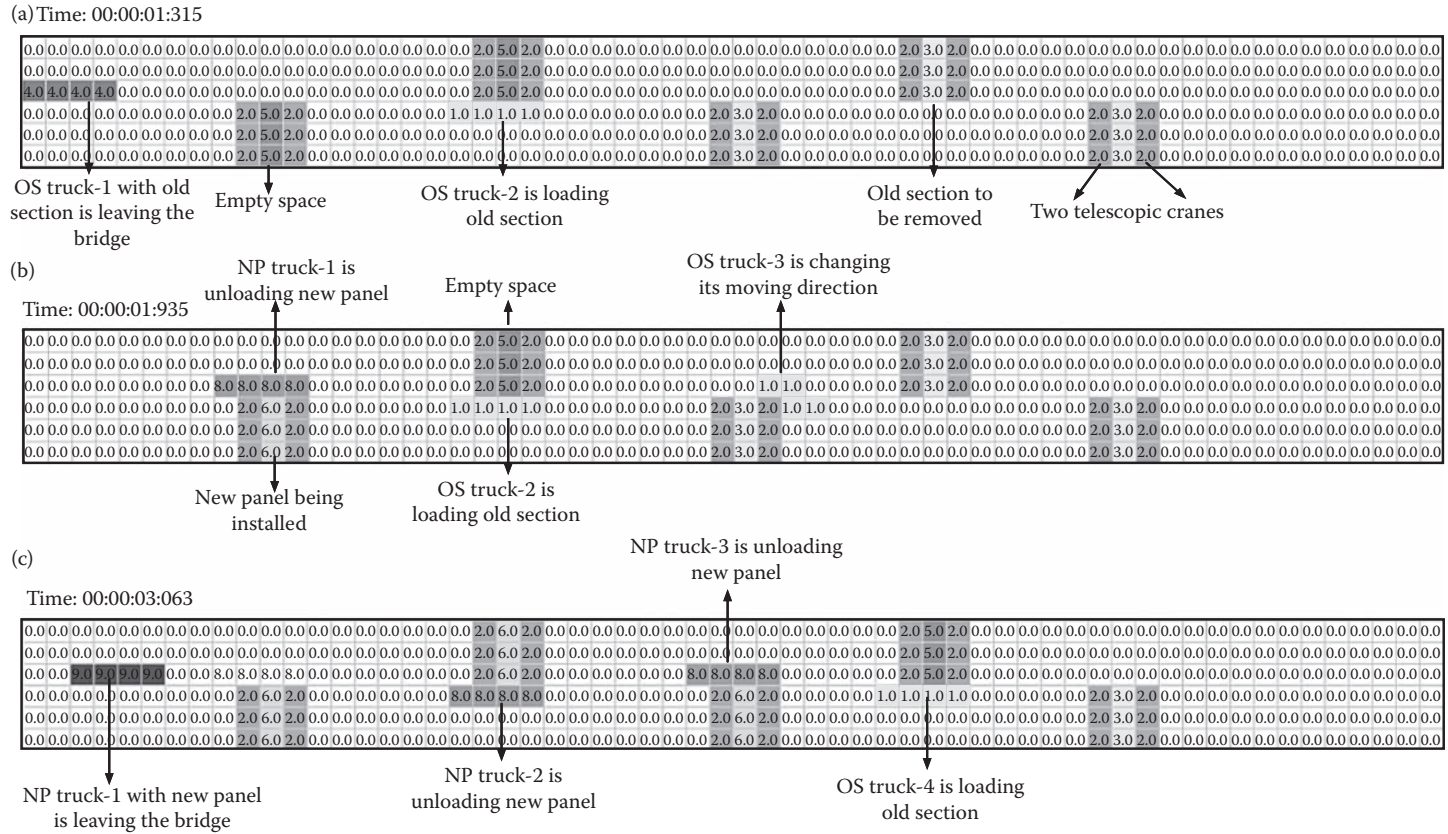


FIGURE 12.6 Snapshots of the *occupancy* layer of the *bridge* model.

12. Visualizing the Simulation Results

As an example, Figure 12.6 shows the animation snapshots of the *Occupancy* layer of the *Bridge* model at different times. With animation, the user can easily identify each object and its state.

13. Analyzing the Simulation Results

Assumptions: The analysis of the simulation results is based on the following assumptions: (1) The simulation time is 9 hours to represent the real operation shift from 8:30 p.m. to 5:30 a.m. the next day. (2) For simplicity, this case study concentrates on the spatial conflicts occurring in the *Bridge* model and ignores the spatial constraints in other models. (3) Both the OS trucks and NP trucks move on the bridge at a constant speed of 10 km/h. The effect of reduced speeds when changing direction is also considered. (4) There is only one lane for the trucks to arrive at the bridge and four lanes for them to leave. (5) For economic analysis, we assigned costs to each resource based on the real project data.

Quantifying Delays Resulting from Spatial Conflicts: The emphasis of this case study is investigating the effects of spatial conflicts on productivity. Therefore, it is important to detect conflicts on the bridge and the delays resulting from these conflicts. Trucks on the bridge may encounter various spatial conflicts. They may change directions to turn around obstacles, which results in changing direction delays. In other cases, a truck might stop temporarily and yield priority of movement to another truck, which results in waiting delays. Occasionally, two or more trucks may get to the bridge at the same time or very close. The *Delay-Queue* model will separate them by a short interval and the delays because of the separation can be recorded. These delays are analyzed as shown in Figure 12.7a. For Pattern-B and Pattern-C, the delays are sensitive to the number of teams. With the increase of the number of teams, the delays increase significantly. The number of trucks also influences the delays. For Pattern-A, the influence of different resource combinations is much less than the other patterns and the delays are significantly less for each resource combination, while the difference between Pattern-B and Pattern-C is subtle. Therefore, from the perspective of less spatial conflicts or delays, Pattern-A is better than the other patterns, which is consistent with our expectations since there are less directional changes in Pattern-A. This result indicates that the spatial delays can be decreased by properly arranging the resources on site and selecting resource combinations.

Sensitivity Analysis Based on Resource Combinations: This analysis is performed by changing the number of each resource for each site layout pattern. Many resource combinations are applied using the cell-based simulation model and the MicroCYCLONE model. As shown in Figure 12.7b, based on 45 resource combinations for each of the three site layout patterns, it is found that: (1) The contractor was able to replace eight panels per

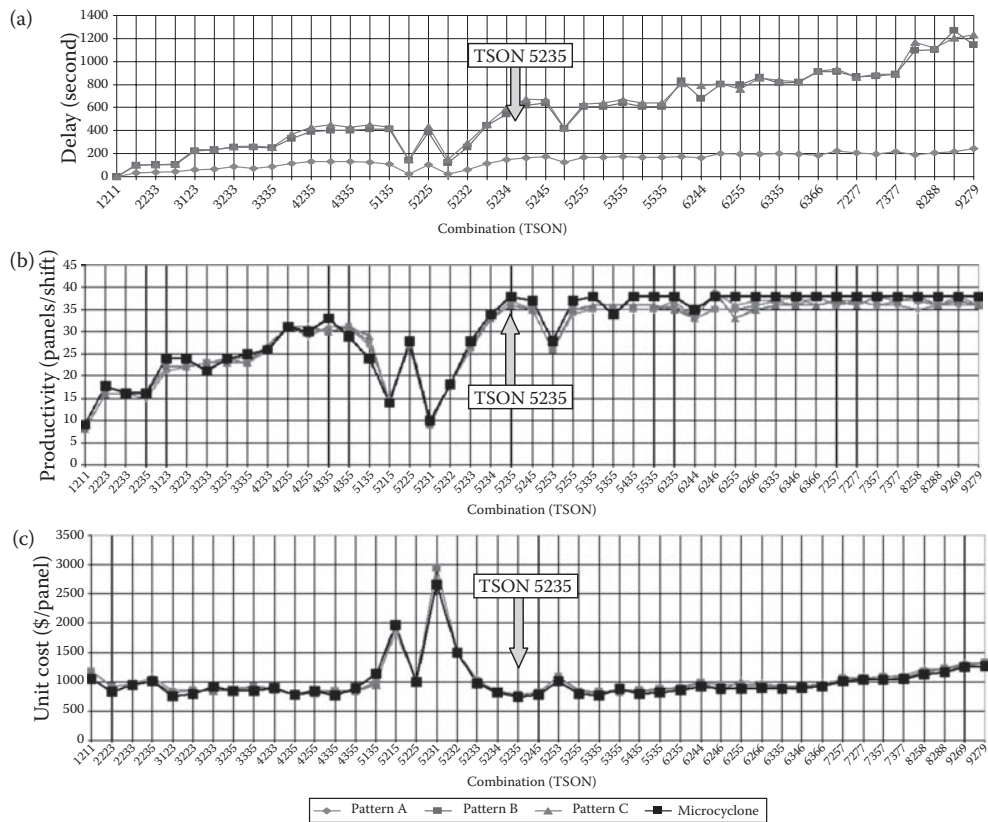


FIGURE 12.7 Comparisons between different site layouts with different resource combinations (a) delays, (b) productivity, and (c) unit cost.

crew per night with the resource combination of TSON 2223. The simulation results show that the total number of installed panels of this resource combination is 16, which matches the productivity of the real project. (2) The results of both modeling techniques (cell-based and MicroCYCLONE) are similar, which indicates that the cell-based model predicts the performance of different resource combinations and site layout patterns properly. (3) In most combinations, the productivity of the MicroCYCLONE model is higher than that of the cell-based model by about 3–5% (1–2 panels/shift). This indicates that the influence of space constraints is small when the speed of the trucks is constant at 10 km/h. (4) In most combinations, the productivity of Pattern-A is a little higher than the other two patterns. This is because Pattern-A has less delays resulting from spatial conflicts as was mentioned above.

Optimal Resource Combination: The lowest unit cost can be found using the MicroCYCLONE model at the combination TSON 5235 with the value of \$746.45 per panel. However, for the cell-based method, different site layout patterns should be considered. In Figure 12.7c, for Pattern-B and Pattern-C, the lowest unit cost is \$783.87 per panel at the combination TSON 4235, while the unit cost of the combination TSON 5235 is \$787.50 per panel. For Pattern-A, the lowest unit cost is \$766.22 per panel at the combination TSON 5235, while the unit cost of the combination TSON 4235 is still \$783.87 per panel. Since Pattern-A has a lower unit cost and less spatial delays, the optimal resource combination can be roughly confirmed as the combination TSON 5235 with Pattern-A.

Effects of Reduced Turning Speed: In the above analysis, the cell-based simulation was based on the assumption that trucks move at a constant speed of one cell per second (10 km/h). In fact, when a truck is changing direction, it has to slow down and longer delays should be taken into account. Nine combinations with Pattern-B were selected for testing the effect of a reduced turning speed of 1 km/h. Figure 12.8 shows the productivity with a reduced turning speed of 1 km/h and a normal speed of 10 km/h. The productivity rate dropped to 87% with TSON 4235 and to 89% with TSON 5235, which indicates that the effect of changing speed should not be ignored.

12.3.2 CASE STUDY II—ANALYSIS OF EFFECT OF WORK ZONES ON TRAFFIC FLOW

In order to demonstrate the applicability of the proposed cell-based simulation to different types of construction projects, another case study about traffic flow analysis considering work zones during asphalt pavement construction is discussed as an example of noncyclic simulation. Through the evaluation of the conditions on the observed road section and the control of the vehicles' movement, the effects of construction operations on the traffic flow under different scenarios can be investigated.

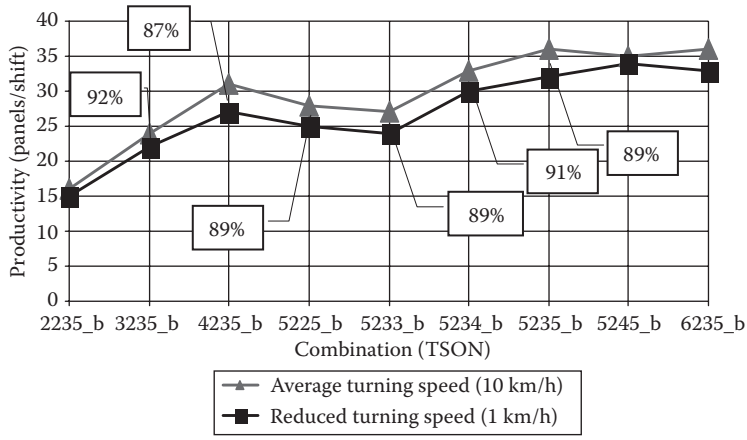


FIGURE 12.8 Effects of reduced turning speed.

In the following paragraphs, only the main steps of developing the cell-based model and analyzing the results are described.

12.3.2.1 Developing the Cell-Based Simulation Model

In this case study, it is assumed that the paving operations are being done on only half of a six-lane road and the closure scenarios influence only one direction of the road. Therefore, the observed section can be simplified as a three-lane one-way road section. A cell-based simulation model is developed as shown in Figure 12.9. The *Generator* generates cars and trucks according to the current traffic volume and the truck/car rate. The *Sender* receives and counts the generated cars or trucks, and then distributes them randomly to the three *Queues* through the three output ports. The *Road* is a Cell-DEVS model where the interferences between construction operations and traffic occur. It represents a section of the road with three lanes. When a vehicle is sent to a lane of the observed section, it has to make a lane-changing maneuver if this lane has been reserved for construction; otherwise, the vehicle can continue moving on the same lane. Three *Control Units* are used to check the first four cells of each lane to determine if a car or a truck can come in. For example, if the first four cells are empty, a truck can enter the road section. Otherwise, the truck cannot enter into the observed section because of the traffic jam and will be stored in one of the three corresponding *Queues*.

The size of the cells is taken as 3 m × 3 m. The observed road section is represented by 60 × 3 cells. A car is represented by one cell and a truck is represented by 3 × 1 cells (equivalent to three vehicles). This case study also adopts the three-layer structure used in Case Study I for the *Road* model as shown in Figure 12.4 but with different codes. The paving operations could reserve one or two lanes and the other lanes could be used for traffic. Accordingly, the *Road* is divided into two zones to adopt different rules for the work zones and traffic lanes. The priority

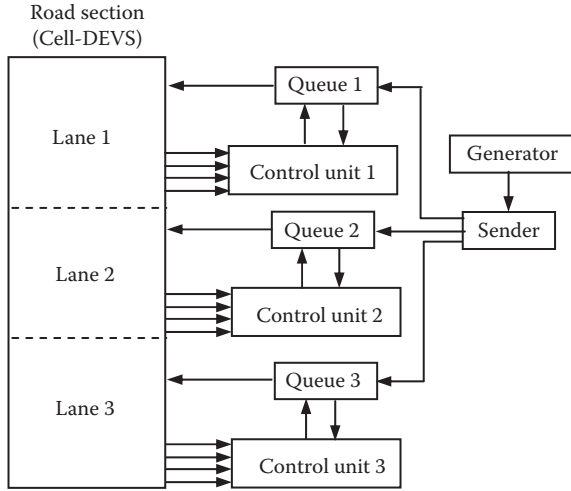


FIGURE 12.9 Cell-based model of pavement project.

rules are defined as the following: (1) When two vehicles are moving on the same lane, the interval between them is at least one cell; otherwise, the latter has to stop temporarily, yielding priority to the former. When vehicles are stopping temporarily, no space is kept between them. (2) When two vehicles are moving on different lanes (Figure 12.10), if the vehicle on the upper lane is ahead of the vehicle on the lower lane without any overlap, the vehicle on the upper lane has priority. (3) When two vehicles are moving on different lanes, if the vehicle on the upper lane is ahead of the vehicle on the lower lane but they have an overlap, the vehicle on the lower lane has priority and the vehicle on the upper lane has to stop temporarily. (4) When a vehicle is taking priority to turn, other vehicles that may cause spatial conflicts with this vehicle have to stop until the vehicle leaves.

12.3.2.2 Visualizing and Analyzing the Simulation Results

Figure 12.10 shows the animation snapshots of the *Occupancy* layer of the *Road* model with different lane-closure scenarios. Through animation, the user can identify the interferences between the operation zone and traffic flow. For comparison, three scenarios are considered as shown in Figure 12.10: traffic flow on the three-lane road section without lane closure, with one-lane closure, and with two-lane closure. Three traffic volumes are considered (5040, 2520, and 1680 equivalent vehicles/h, with a truck rate of 20% and an average speed of 10 km/h). The vehicles are sent to one of the three lanes randomly. Figure 12.11 shows the numbers of vehicles passed through the road section under different lane-closure scenarios. There is an obvious drop in the case of 5040 vehicles/h, from 2513 vehicles/h without lane closure to 2318 and 1530 vehicles/h with one-lane closure and two-lane closure, respectively. The remaining traffic volume is waiting in the queues. This result means that the two-lane closure has a much greater influence on the road traffic with a traffic volume of 5040 vehicles/h. Also, the figure shows that there is not much influence in the other two scenarios where the traffic volumes are 2520 and 1680 vehicles/h. This

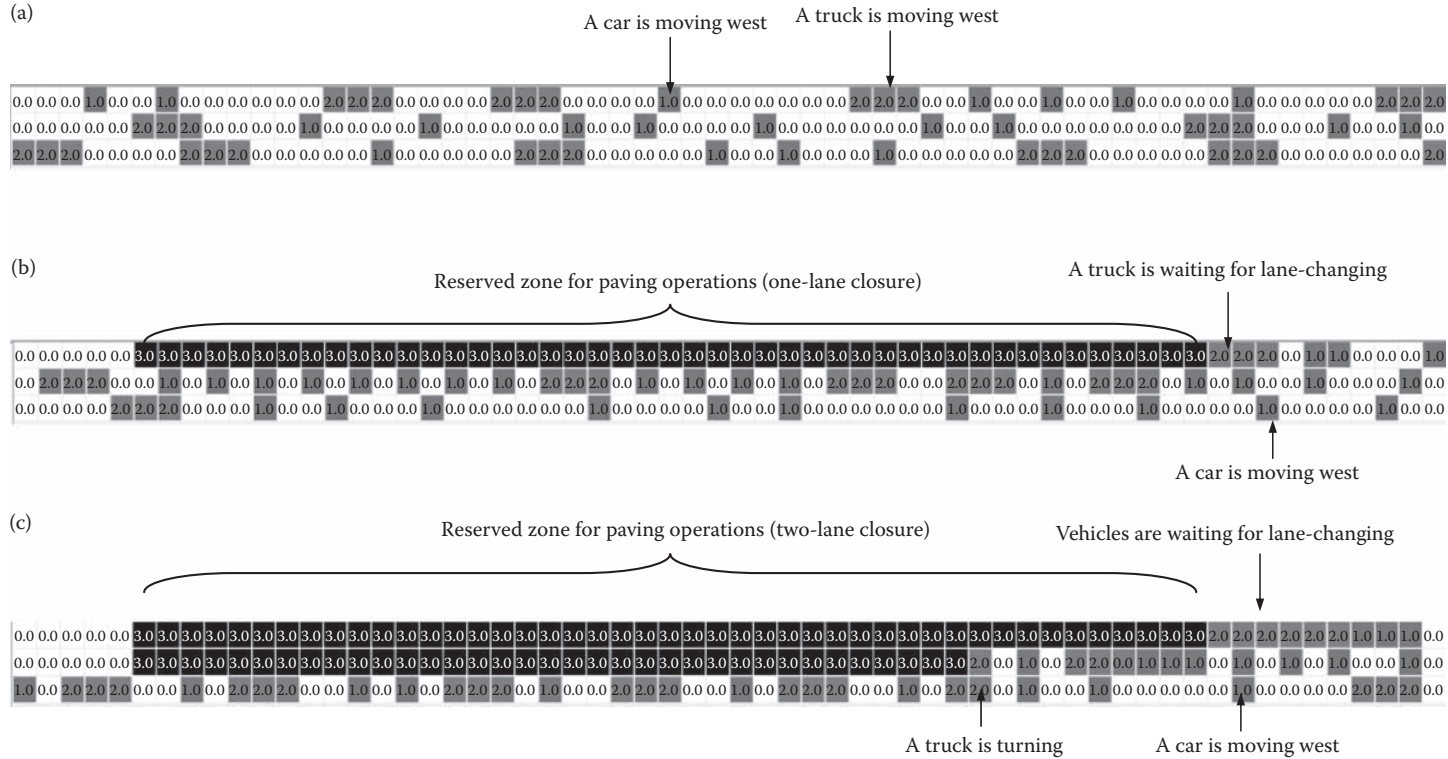


FIGURE 12.10 Snapshots of the animation of the traffic flow under different lane closure scenarios. (a) Traffic flow on a three lane road without lane closure, (b) traffic flow on a threelane road with a one lane closure, and (c) traffic flow on a threelane road with a two lane closure.

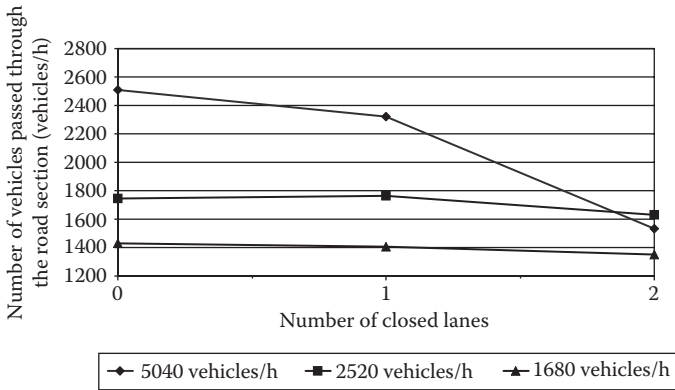


FIGURE 12.11 Comparisons of traffic flow under different lane closure scenarios.

case study helps the project manager to predict the effects of the paving operations on the traffic flow and then to adjust the lane-closure strategies according to the traffic volume.

12.4 CONCLUSIONS AND DISCUSSION

The work presented in this chapter proposed a cell-based construction simulation method using the Cell-DEVS technique. In this method, the spatial constraints, site layout patterns, and resource allocations can be explicitly represented. Consequently, spatial conflicts can be detected and resolved. The effects of spatial constraints can be analyzed based on site layout patterns, resource combinations, and vehicle moving speeds. Two case studies were used to implement and demonstrate the advantages and applicability of the method. In addition, sets of rules and models have been defined that can be reused in other simulation studies.

The simulation results show that: (1) The cell-based simulation method is a general-purpose simulation tool for broad domains and can be used to simulate complex construction operations, especially when spatial constraints are crucial to the project. (2) Space is represented explicitly so that the occupation of the workspace and other spatial information about the construction environment can be understood more easily than the abstract symbols in simulation networks. Furthermore, the influence of different site layouts can be investigated. (3) An uninterrupted conflict-free animation provides a better means for practitioners to check the simulation results. (4) The effect of changing speed could be analyzed in cell-based simulation. In the case study of the Jacques Cartier Bridge rehabilitation project, the simulation results showed that the impact of additional delays because of the slower turning speed may lead to a 13% drop in productivity in some cases.

Through our experience with Cell-DEVS modeling, it was found that developing large scale models could become intractable with a large number of rules. Future research in Cell-DEVS modeling tools may alleviate this problem.

ACKNOWLEDGMENTS

We would like to thank Dr. Gabriel A. Wainer for providing the tools for realizing Cell-DEVS simulation. Also, we would like to express our appreciation to Mr. German Cardinal and Mr. Raymond Coté from SNC-Lavalin for providing the data about the Jacques Cartier Bridge rehabilitation project and providing suggestions about the simulation model.

REFERENCES

1. Halpin, D. W. "CYCLONE: Method for modeling of job site processes." *Journal of the Construction Division, ASCE* 103, no. 3 (1977): 489–99.
2. Martinez, J. C. *STROBOSCOPE: State and Resource Based Simulation of Construction Processes*, PhD Dissertation. Ann Arbor, MI: University of Michigan, 1996.
3. Hajjar, D., and AbouRizk, S. M. (1999). "Symphony: an environment for building special purpose construction simulation tools." *Proceedings of the 1999 Winter Simulation Conference*, 998–1006, 1999.
4. Kamat, V. R., and Martinez, J. C. "Visualizing simulated construction operations in 3D." *Journal of Computing in Civil Engineering, ASCE* 15, no. 4 (2001): 329–37.
5. Tommelein, I. D., Levitt, R. E., and Hayes-Roth, B. "Site-layout modeling: How can artificial intelligence help?" *Journal of Construction Engineering and Management, ASCE* 118, no. 3 (1992): 594–611.
6. Guo, S. "Identification and resolution of workspace conflicts in building construction." *Journal of Construction Engineering and Management, ASCE* 128, no. 4 (2002): 287–95.
7. Wadhwa, L. C. "Optimizing deployment of shiploaders at bulk export terminal." *Journal of Waterway, Port, Coastal and Ocean Engineering*, November/December 2000, 297–304.
8. Randhawa, S. U., and Sharoff, R. "Simulation base design evaluation of unit load automated storage/retrieval systems." *Computers and Industrial Engineering* 28, no. 1 (1995): 71–79.
9. Caron, F., Marchet, G., and Perego, A. "Layout design in manual picking systems: A simulation approach." *Integrated Manufacturing Systems* 11, no. 2 (2000): 94–104.
10. Marasini, R., and Dawood, N. "Simulation modeling and optimization of stockyard layout for precast concrete products." *Proceedings of the 2002 Winter Simulation Conference*, 1731–36, 2002.
11. Tawfik, H. M., and Fernando, T. "A simulation environment for construction site planning." *Proceedings of 5th International Conference of Information Visualization*, 199–204, London, 2001.
12. Nassar, K., Thabet, W., and Beliveau, Y. "Simulation of asphalt paving operations under lane closure conditions." *Automation in Construction* 12 (2003): 527–41.
13. Wainer, G. "CD ++: A toolkit to define discrete-event models." *Software, Practice and Experience* 32, no. 3 (2002): 1261–1306.
14. Zhang, C., A. Hammad, T. M. Zayed, and G. Wainer. "Representation and analysis of spatial resources in construction simulation." *Proceedings of the 2005 Winter Simulation Conference*, Florida, 2005.
15. Zhang, C., Hammad, A., Zayed, T. M., Wainer, G., and Pang, H. "Cell-based representation and analysis of spatial resources in construction simulation." *Journal of Automation in Construction* 16, no. 4 (2007): 436–48.
16. Zaki, A. R., and Mailhot, G. "Deck reconstruction of Jacques Cartier Bridge using precast prestressed high performance concrete panels." *PCI Journal*, September–October 2003, 2–15.
17. Pang, H. *Cell-based Representation and Analysis of Spatial Resources in Construction Simulation*, Master's Thesis. Montreal, Canada: Concordia University, 2007.

13 Stochastic Modeling Strategies for the Simulation of Large (Spatial) Distributed Systems: Application to Fire Spread

Alexandre Muzy and David R.C. Hill

CONTENTS

13.1	Introduction	332
13.2	Design and Performance Balance	333
13.2.1	Background on Worldviews and Simulator Efficiency	333
13.2.2	Modularity and Efficiency	334
13.2.3	Activity Paradigm	335
13.3	Modeling of Stochastic Aspects in (Spatial) Distributed Systems.....	339
13.4	Fire Spread Modeling through a Virtual Laboratory	340
13.4.1	Fire Spread Modeling	340
13.4.2	Experimental Model 1: Near-to-Near Propagation Including Firebrands	343
13.4.3	Experimental Model 2: Pseudorandom Distributions of Firebrands	344
13.4.4	Experimental Model 3: A More Sophisticated Approach.....	345
13.4.5	Efficiency and Activity	347
13.5	DEVS Discussion	350
	References.....	353

13.1 INTRODUCTION

The implementation of large (spatial) distributed systems is usually based on either:

1. Mathematical models, which are generally obtained from modeling discipline hypotheses (e.g., Physics, Economics, and Biology). Mappings are then defined from mathematical structures to computational ones. Modeling capabilities of computers are thus constrained by previous discipline hypothesis of usual scientific disciplines and/or by the discretization methods used to implement the model.
2. Fully computational models, which are often derived from heuristic hypotheses. Heuristics correspond to the modeler or expert knowledge. The latter has to try carefully to avoid usual modeling techniques of disciplines. Using this strategy, the modeler is expected to be in a better position to explore new heuristics finding new modeling hypotheses. Not using usual discipline knowledge allows creating new knowledge and original tools to interact with both system and usual models already developed. However, modeling frameworks are guides through these new unknown modeling paths. This chapter discusses this kind of framework.

Computers have been developed to achieve massively, quickly, and automatically calculus performable by humans by hand. Currently, as a result of the increase of computational and communication capabilities, new scientific achievements can be reached by storage of large amounts of data and distributed computing. Developing new modeling and simulation frameworks able to deal with large spatial data is now common in many high performance computing projects.

However, despite powerful computational capabilities, a high number of interacting subsystems requires the efficient exploitation of computational resources. Therefore, a framework for the modeling and simulation of large (spatial) interacting systems requires: (1) guiding the modeler sufficiently to deal with heuristic modeling hypotheses, and (2) being efficient enough to implement tractable solutions.

A last refinement about framework characteristics concerns its first requirement and its ability to guide the modeler sufficiently to deal with “heuristic modeling hypotheses.” Trying to escape from discipline modeling techniques (usually inherited from analytical techniques) necessitates providing another computation-based technique. Modeling and simulation can be done based on data of one or many systems. The more experimental data are collected, the more “precise” or “certain” the simulation model can be. Human knowledge about large (spatial) distributed systems is far from complete and actually uncertain (notably because human capabilities are limited). Considering uncertainty and the opposite “certainty,” the Monte Carlo technique is a powerful means to make explicit the exploration of model parameters [1]. Probabilities and distribution laws are assigned to parameter values. Investigation of state spaces through parameter changes can be automatically achieved using experimental plans [2,3]. Finally, a confidence interval can be obtained on probability results.

The topic of study in this chapter is large (spatial) interacting systems. In that context, we introduce: (i) efficient design strategies, (ii) stochastic modeling approaches, and (iii) the development of fire spread computational models (vs. physics-based models).

13.2 DESIGN AND PERFORMANCE BALANCE

A good design should lead to a computational model that is as simple and as evolvable as possible. Good performances are, for large distributed systems, inversely proportional to the design quality. Performances require the removing and/or simplification of redundant and nonefficient (even if reusable and more understandable) code or structures. A trade-off has to be made between efficiency and design quality. To optimally choose between corresponding solutions of efficiency and design, the activity paradigm can be used for both design (at the modeling level) and efficiency (at the simulation level) tasks.

13.2.1 BACKGROUND ON WORLDVIEWS AND SIMULATOR EFFICIENCY

There are three common types of strategy to implement the kernels of discrete-event simulations [4]. These strategies, also called worldviews, consist of: *event-scheduling*, *activity-scanning* (including the three phases optimization), and the *process-oriented* strategy introduced by the Simula language [5]. A strategy makes certain forms of model description more naturally expressible than others. In all of these world-views, an event corresponds to an instantaneous change in the state of a system at a particular time. Event scheduling models work with the prescheduling of all events without provisions for activating events. In contrast, in the activity scanning approach, events can be conditioned on a contingency test in addition to being scheduled. A model is said to be active when its scheduling time has occurred and when its contingency test is satisfied. An optimization of the activity scanning strategy is named the three phases approach [6,7]. The interaction process world-view is a combination of both event-scheduling and activity scanning strategies.

An additional classification consists of considering the two kinds of time management: discrete-time and discrete-event simulations. In discrete-time simulations, a clock advances the simulation by a fixed time step. At every time step, states are computed. In discrete-event time based management, events drive the simulation. The simulation time advances from one event time-stamp to another according to the events scheduled.

All these techniques and strategies focus on particular concepts (events, activities, and processes) and simulation time managements. On the other hand, source systems are also usually described through a system-based decomposition. According to interactions and autonomy of systems, systems are identified and connected. Discrete Event System Specification (DEVS); [8] is the soundest framework aiming at tackling computational systems, discrete-events, and simulation time managements. Through its specification hierarchy, depending on the observed behavior of the entire system, the internal structure is progressively described up to the definition of interfaces (or ports), in a modular way. At the final specification level, a submodel is fully modular. At the previous level, the states of submodels can directly influence the transitions of other submodels. The choice of specification level is crucial in terms of reusability and efficiency. Currently, three kinds of approaches discuss both modularity and efficiency of DEVS models:

1. At the simulation level, Ref. [9] propose to flatten the hierarchy of cellular models. Ref. [10] aim at taking advantage of spatially distributed causal events in cellular models. Ref. [11] suggest activity tracking mechanisms

- only focusing on active components, reducing the data structure overheads resulting from discrete-event managements.
2. At the compilation level, [12] only account for active event paths. Subsystems that will not receive events or compute transitions during the simulation are not compiled. This results in a reduction of memory size and a simulation speed-up.
 3. At the modeling level, wrapping and modularity are compared in terms of reusability and efficiency [13–15]. Modularity increase reduces performances (because of discrete-event managements) but improves reusability. A weaker modularity (reducing state encapsulation) reduces the reusability of atomic systems, while improving performance. Obviously, the use of port interfaces induces an increase in the number of discrete-events and overhead of data structure management.

13.2.2 MODULARITY AND EFFICIENCY

As depicted in Figure 13.1, various choices of modularity can be made for describing a spatial system. At the modeling level, models can be implemented in a modular or in a nonmodular way. Modular models consist of cells whose state is encapsulated.

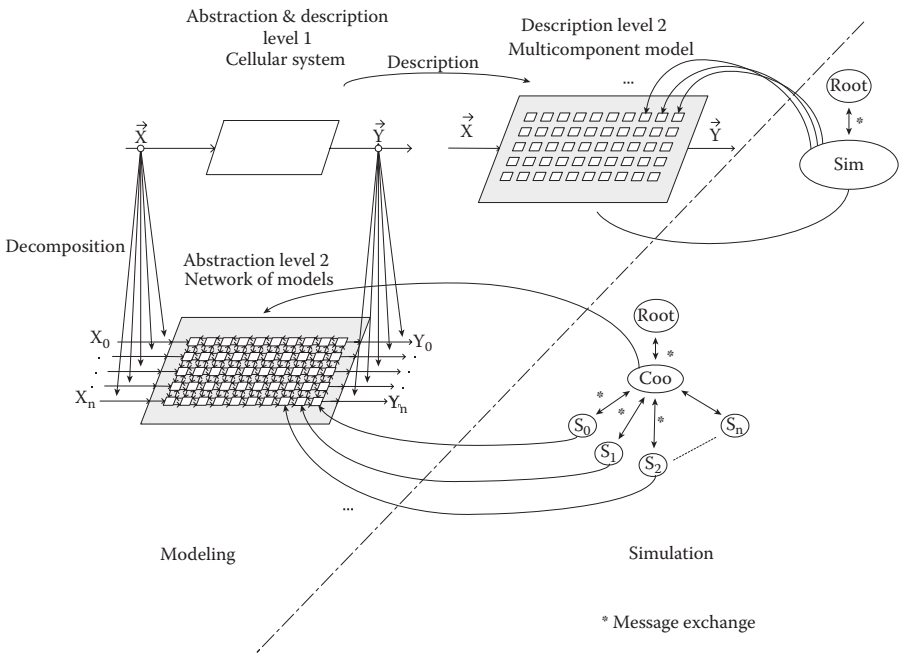


FIGURE 13.1 Comparison between modular and nonmodular modeling and simulation. (Muzy, A., Innocenti, E., Hill, D., and Santucci, J. F., *36th Annual Simulation Symposium, IEEE/SCS/ACM*, Orlando, FL, 289–96, 2003.)

Cells interact through interface ports and every cell is designed as receiving and sending events. Nonmodular cells do not have ports. They directly access the state of influencing neighbors.

At the simulation level, a full aggregation (corresponding to a nonmodular implementation) corresponds to a single simulator driven by the root coordinator. A distributed solution consists in assigning one simulator to every cell. This solution increases the number of messages exchanged (decreasing efficiency) but enhances reusability.

A first full modular implementation at both model and simulator levels has been implemented through the Cell-DEVS formalism [9] and the corresponding CD++ environment [16]. To illustrate some performance aspects, consider a laboratory experiment that consists of a combustion table of 30 cm long and 60 cm wide for a line-ignition, the prediction of spread rate (2.96 mm/s). For a real propagation of 150 s of a 100×100 cell space, the execution time of the Cell-DEVS simulation was about 21 hours and 20 minutes (with an old *Pentium III* at 500 Mhz). Even if this modeling and simulation experiment is elegant and grounded formally, the exchange of many messages produces significant overhead because of data structure management (of schedulers). This results in an execution time much greater than the actual propagation time.

On the other hand, a fully aggregated simulator and nonmodular model has been implemented [13]. An ignition point has been simulated by initializing center cells with a temperature gradient. For a real propagation of 200 s, execution times decreased to 160 s. The drawback, however, is that the entire modeling and simulation model is less reusable.

Many choices can be considered at the modeling and simulation levels for enhancing efficiency. According to the degree of reusability requested, modularity can be achieved at the modeling and/or simulation levels. Reducing the degree of modularity reduces the degree of reusability of components. Full modular components are designed in an autonomous way. They react to and send external discrete-events. Take the example of a cellular automaton. In a nonmodular cellular automaton, simple neighboring rules can be implemented for every cell as: "If my neighboring cell is alive, then I become alive." In a modular cellular automaton, the specification is different. It would be: "If the message received from my neighboring cell indicates that it is alive, then I become alive." The internal structure of nonmodular models depends more directly on the state of its influencing models.

13.2.3 ACTIVITY PARADIGM

Beyond choosing the modularity of a model, the modeler can use a new technique to enhance efficiency: The activity paradigm. Using the latter, models and simulators are designed to automatically track activity (state changes of components). Activity detection rules are added to models. Conversely, this technique increases explicit modularity and nonmodularity aspects. Indeed, DEVS is a powerful framework, however, its model and simulator structures have to be exploited to map activity of components. Subsystems can be modeled as active or inactive,

the entire simulation only directly tracking and computing activity while ignoring inactivity. Simulation resources can be characterized through activity comparing resource usages of modular and nonmodular models. Let's take a simple example of fire spread in a 100×100 cell space (cf. Figure 13.2). Inactive cells (burnt in black or unburnt away ahead of the fire front) and active cells (of the fire front) can easily be identified.

Activity tracking consists in only including burning or heated cells in the main simulation loop. A mean relative activity measure can be studied:

$$\mathring{A} = \frac{Activity_{int} + Activity_{ext}}{\max(Activity_{int}).T_{sim}} .$$

Where $\mathring{A}$ is the percentage of mean activity measure relative to $\max(Activity_{int})$, which represents the maximum number of internal transitions (accounting for all active as well as inactive internal transitions of cells over the entire propagation domain), $Activity_{ext}$ is the number of external transitions, and T_{sim} is the simulation time period.

Then, as depicted in Figure 13.3, applying this simple mean relative activity metrics (counting the number of external and internal transitions of components over a total simulation time of 221 s, which corresponds to the end of fire spread propagation when using a discrete-time step of 0.01 s) allows investigating modularity and activity tracking combinations—experimentally and theoretically:

- (No activity tracking, no modularity): A total of 10,000 internal transitions for all cells (active and inactive) are achieved at every time step. This corresponds to a maximum of $\mathring{A} = 100\%$.
- (Activity tracking, no modularity): The average number of active cells is experimentally determined (Muzy et al. 2008): $\mathring{A} = 23.29\%$.

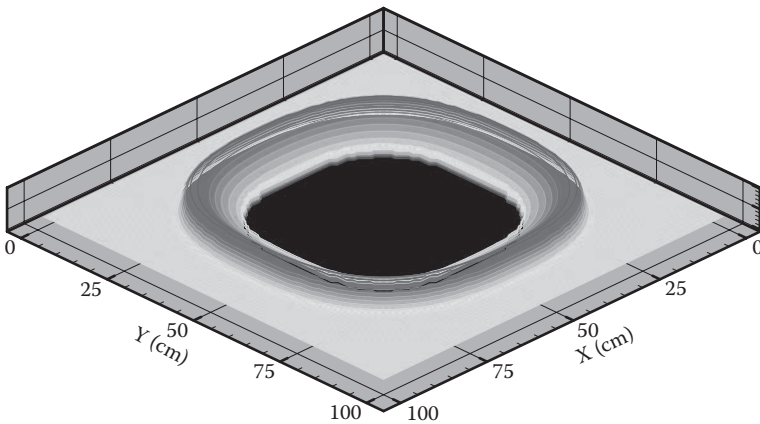


FIGURE 13.2 Fire spread example with an ignition point at the center of the square.

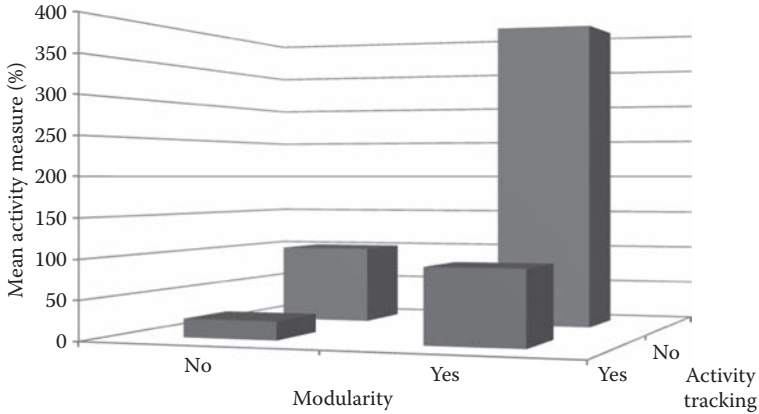


FIGURE 13.3 Activity and modularity comparison of efficiency.

- (Activity tracking, modularity): $\bar{A} = 93.14\%$, simply multiplying the (activity tracking, no modularity) case by four (corresponding to the number of neighbors per cell).
- (No activity tracking, no modularity): $\bar{A} = 400\%$, simply multiplying the (no activity tracking, no modularity) case by four (corresponding to the number of neighbors per cell).

This simple comparison metrics pinpoints the efficiency of activity tracking approaches through modularity. One can notice that metrics for both (activity tracking, modularity) and (no activity tracking, no modularity) are very close. Notice that the efficiency of the (activity tracking, modularity) case will improve linearly with the number of inactive components. According to the superiority of the (activity tracking, no modularity) case, no doubts arise concerning the efficiency supremacy of this solution. Conversely, the (no activity tracking, modularity) case seems to not have any benefits that would merit an implementation.

Here are the lessons that can be learned from this simple example:

1. Do not use explicit discrete-event exchanges for massively parallel discrete-time implementation. Discrete-events should be used with a discrete-event (continuous) time base in mind.
2. Use activity tracking when the mean number of active components is low compared to the total number of components.
3. Discrete-event overhead can be compensated for by a high number of inactive components.

More recently the activity tracking paradigm has been introduced [17]. This paradigm describes and proposes structures focusing on active subsystems. Usual time flows and world-views are embedded. The formal description of DEVS models is discussed through activity-based simulation algorithms and new structures accounting

for dynamic structures. Using the activity-tracking paradigm, components can be modeled and simulated in two steps:

1. The propagation activity is tracked. Information exchanged between components is routed and computed. The current set of active components is scanned. Events are routed and output transitions are computed. Final receivers are detected in the hierarchy using a recursive routing function [11]. The active set is then updated including imminent components for external transitions. The order of the active set depends on a tie-breaking function of imminent components [8].
2. According to current states and to new inputs, new states are computed. External and internal transitions (because of external and internal events) of active components are computed. Components changing state significantly are marked to be added to the new (ordered) active set. In a discrete-event driven simulation, the new active set corresponds to a scheduler and active components are marked to execute further their internal transition function (corresponding to an internal event occurrence). In this case, the current active (ordered) set is a subset of the scheduler, which corresponds to components active at the current simulation time.

Models can be designed for tracking activity during the entire simulation loop. Metaphorically, distributed submodels (constituting the entire model) can be considered as filtering gates orienting and computing activity. As depicted in Figure 13.4, models are activated or not by the flows of external events or by scheduled internal events (bottom right corner). This activation depends on the adjustment of a sensitivity parameter determining the activation of the model. Activity in a network of models depends on: (i) state transitions or internal events and (ii) external events received and sent.

This algorithm makes explicit the distinction between the activity tracking of: (i) events and (ii) components. Using only the second phase corresponds simply to

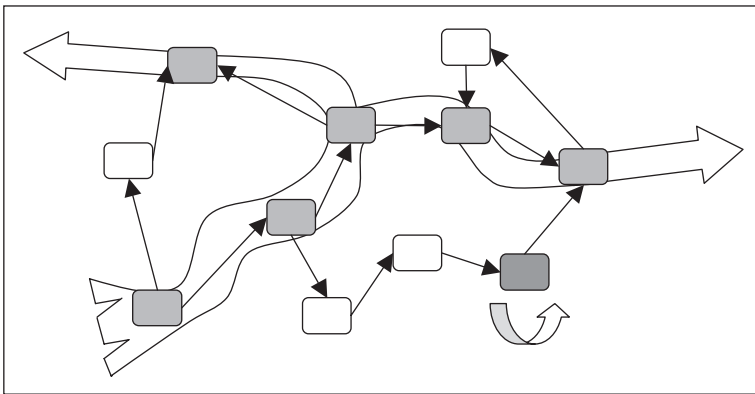


FIGURE 13.4 Active components.

a three-phase algorithm: The three-phase approach executes sequentially [6]: (1) the time-scanning (A Phase); (2) the bound-to-occur or book-keeping activities that represent the unconditional state changes (unconditional events) that can be scheduled in advance (B phase); and (3) the conditional or cooperative activities that represent the state changes that are conditional upon the cooperation of different objects or the satisfaction of specific (compound) conditions (C phase).

The activity paradigm renders explicit the many strategies offered to the modeller for the design of large (spatial) distributed systems. According to his own design and efficiency constraints, he can alternatively opt for tracking activity or not, as well as choosing to use modularity or not.

13.3 MODELING OF STOCHASTIC ASPECTS IN (SPATIAL) DISTRIBUTED SYSTEMS

In many domains, stochastic aspects of spatial spreading have to be considered. At the beginning of the millennium, colleagues, like Lewis, were surprised that despite the recognized importance of stochastic factors, recent models for ecological invasions were almost exclusively formulated using deterministic equations [18]. In forestry, the forest growth can rely, for instance, on the distant spread of seeds. It has been shown that pioneering trees, whose seeds were propagated by heavy winds, do have an impact on the colonization of landscapes by particular species [19,20]. In oceanography, the spreading of a tropical alga, *Caulerpa taxifolia*, introduced by mistake in the Mediterranean Sea is based on the spatial spreading of cuttings. This spreading is in essence stochastic, and occurs at a stochastic and discrete distance from an original point of settlement. Only stochastic models under spatial constraints were able to reproduce maps of the spreading of this alga [21].

In fundamental domains such as physics, stochastic and spatial modeling enables the understanding of phenomena in which determinism does not offer the same precision (e.g., the spreading of spiral waves); [22]. Here, the effects observed can be understood as a generalization of the concept of stochastic resonance in spatially extended systems. In [23], a study is documented of intracellular spreading of calcium-induced calcium release with the stochastic DeYoung–Keizer-model. The system under study presents a state characterized by backfiring. The backfiring occurs because the steadily propagating pulse solution undergoes a heteroclinic bifurcation (in *dynamical systems* this type of bifurcation is a *global bifurcation* involving a *heteroclinic cycle* that is an invariant set in the phase space of a *dynamical system*). The use of spatial stochastic modeling was also chosen to argue that quantum-gravitational fluctuations in the space-time background give the vacuum nontrivial optical properties that include diffusion and consequent uncertainties in the arrival times of photons, causing stochastic fluctuations in the velocity of light in vacuo [24]. In nuclear medicine, we have shown that the precise detection of small tumors with an error less than 10% can currently only be achieved by spatial Monte Carlo simulations [25].

Spatial simulations are generally displayed with computer graphics. Visualization and animation is beneficial because of the human ability to capture spatial relationships. However, in the context of stochastic simulations, when we observe the results of a single replication, it is very important to avoid concluding something with only a small sample of a complex random variable. This does not mean that the study of transient behavior is not useful, but this implies more complex computing when we want to observe the results of many replications. For instance, in the *Caulerpa taxifolia* spreading simulation, the spatial auto-correlation is strong since *Caulerpa taxifolia* contaminated zones tend to form aggregated spots. On the contrary, from one replicate to the other, peripheral spots distribution is completely different without apparent correlation. Thus, it can be interesting to sum up the results of a large number of replicates in a single representation of the frequencies. This constitutes a Discrete Spectral Analysis we already defined [21,26]. Spectral analysis helps us to point out areas that have a high probability of invasion by *Caulerpa taxifolia*. However, this leads to difficulties in result interpretation, since the visualized result is a sum in the space of possible solutions. Furthermore, the existence of a peak is not easy to analyze and sometimes needs deeper study to discover possible spatial attractors [27]. Indeed, this technique itself is not able to detect some multiple spatial attractors because of spatial correlations. Consequently, in some cases the system could significantly diverge from the predicted results. Unfortunately, no reliable mathematical or statistical techniques are available at this time to detect such circumstances. Accurate knowledge and the functional analysis of the ecosystems seem to be the only ways to avoid incorrect forecasts and interpretations.

13.4 FIRE SPREAD MODELING THROUGH A VIRTUAL LABORATORY

In physics, equation-based structures are used to represent fundamental identified physical mechanisms involved in both fire and diffusion processes. Parameters of equations have been (or are) identified through laboratory experiments. These parameters do not correspond directly to the parameters intuitively identified as influencing the fire spread (slope, wind, biomass, humidity). These parameters are split into many other parameters, which aim at describing very finely physicochemical mechanisms (gas dynamics, temperature radiation, etc.) These models are more adapted to fine-grain physics-based laboratory experiments than actual fire spread.

Computer scientists have a different approach. First, their laboratory consists only of one or many computers. They directly build models and simulate them. Then, they verify if the model behavior fits their expectations describing consistent dynamics. They calibrate their model by adjusting parameters and exploring parameter values through experimental plans. Afterward, the first constructive experiments designed by computer scientists are described.

13.4.1 FIRE SPREAD MODELING

Fire spread models are usually categorized into forest and urban fire spread models. Forest fires are considered here. Forest fire models can be separated into stochastic and deterministic models. Stochastic models aim at predicting the most probable fire behavior in average conditions. In contrast, in analytical models, the fire behavior is usually deduced from the deterministic physical laws driving the evolution of the system. Recently, several sophisticated models were proposed [28–31] and successfully validated by comparison with real fires. All these models use either simple or Dynamical Structure Cellular Automata (DSCA).

With reference to deterministic models, based on Weber's classification [32], three kinds of mathematical models for fire propagation can be identified according to the methods used in their construction. The first type of models are statistical models [33], which make no attempt at including specific physical mechanisms, being only a statistical description of test fires. The results can be very successful in predicting the outcome of fires similar to the test fires. However, the lack of a physical basis means that the statistical models must be used cautiously outside the test conditions. The second category of models is composed of semiempirical models [34] based on the principle of energy conservation but which do not distinguish between the different mechanisms of heat transfer. Rothermel's stationary model is a one-dimensional model, in which a second dimension can be obtained using propagation algorithms [35] integrating wind and slope. Finally, physics models [36] integrate wind and slope effects in a more robust manner by describing the various mechanisms of heat transfer and production. Physical mechanisms are described using a chemical, thermal, and mechanical definition of basic fire phenomena. Hence, physical and semiempirical models use the definition of basic fire phenomena to physically describe fire propagation.

With reference to stochastic models, few works are available. In [37], authors explore the results of various stochastic experimental designs (according to various classes of moisture and ignition probabilities based on percolation thresholds) of a replicated fire spread simulation. No fuel biomass is taken into account. Besides, authors argue that "each simulation was replicated five times," without substantiating the choice of the number of replications although serious studies of the stochastic variability of results have to be performed [38]. In [39], a sensitivity analysis of Rothermel's model using Monte Carlo simulation is provided. In [40], Monte Carlo simulation is used to evaluate the error of fire spread data acquisition during a fire spread simulation. This error is then integrated into the fire spread simulation. On stochastic fire spread modeling, two major approaches have emerged that must be mentioned. The first is a very precise mathematical framework for stochastic fire spread modeling [41]. In the latter a very clear presentation of an ellipse based stochastic model is presented. An interesting concept of random spread process is introduced. However, here too, no discussions on the stochastic variability of results or biomass modeling are provided. More recently, in the reference journal *Combustion and Flame*, a very interesting and promising article [42] has been introduced (and apparently discussed with the Nobel Prize winner P. G. de Gennes). Small world networks, percolation, and stochastic simulations are designed. This demonstrates a

new interest in the physicist community for computer-based techniques. Long range spotting is modeled and analytic physics-based equations are used to predict fire-brand distances and numbers.

Concerning the discrete-event design of fire spreading models, much work can be cited:

- In [43], a first discrete-event design of Rothermel's model is proposed. Discrete-events correspond to the patch burning times provided by Rothermel's model. An experimental frame is defined. This frame embeds ecological data of the usual fuel classes provided by Rothermel's model.
- In [28], dynamic structure cellular automata are used to store in memory only the burning active cells obtained through Rothermel's model. A dynamic structure specification is provided. Memory reductions are discussed.
- In [44], a high-level specification of [43] is presented. The specification exemplifies the delay-focused and cell-focused macroinstructions of the Cell-DEVS formalism. As macrospecifications and instructions are automatically embedded in Cell-DEVS, a reduction in code size is obtained.
- In [45], an intensive fine-grained semiphysical model of fire spread has been investigated through discrete-event design. Optimizations (at the implementation and specification levels) are investigated in [13] to account for dynamic structures and discrete-time modeling. Recently, in [54], the quantization technique has been investigated and a parallelization of the model has also been considered [46].
- In [47], previous work in [43] has been worked out in detail and extended to many scenarios including wind, slope, and fire fighting. Dynamic structures have been investigated in [48]. A stochastic implementation comparing experimental and simulation data (obtained through Rothermel's model) is described.

The complexity of fire spread modeling introduces bottlenecks at both physical and computer modeling levels. At the physical level, physicists use concepts (convection, radiation, diffusion) and corresponding usual mathematical models (partial differential equations and ordinary differential equations) to describe fire spread models. After a discretization through usual numerical methods (Euler, Runge-Kunta, etc.), computer-scientists then use design techniques (object-oriented, discrete-events, meta-modeling, dynamic structure, etc.). They discuss design and execution time advantages of the techniques used. However, many scientific disadvantages emerge from this approach. The design techniques may evidence issues in an attempt to follow the evolutions of discretized physics-based models. A first drawback is that the computer design is totally dependent on discretization techniques (except perhaps now with quantization techniques from [8]). A second drawback is that computer scientists usually do not master physics-based modeling techniques and physicists do not exploit the full advantages of computer-based modeling techniques that,

moreover, they sometimes do not even know. A third drawback is that the filters constituted by both physics-based concepts and discretization techniques result in multidimensional parameters, which do not directly represent physical, topological, and biological properties of fire fronts and fuels. Modeling levels are not grasped in a single consistent and complete top-down approach (from problem to implementation). Using a consistent stochastic framework, embedding experimental plans, to model (spatial) distributed systems, scientists are able to master and experiment with the full modeling and simulation process (even if they usually need expert advice).

13.4.2 EXPERIMENTAL MODEL 1: NEAR-TO-NEAR PROPAGATION INCLUDING FIREBRANDS

Because of the many (spatially distributed) phenomena occurring uniformly and in parallel with a fire spread, cellular automata constitute an appropriate paradigm to model and simulate them. A very simple program with a textual interface is sufficient to design a first fire spread model.

Cells of the cellular automaton hold a discrete state constituted of symbols as follows:

- ‘.’: The cell is unburned
- ‘*’: The cell is burning
- ‘x’: The cell is burned

Fire propagation is twofold:

1. Near-to-near propagation: Every burning cell gets one chance over eight (for the eight neighbors) to ignite a neighbor. In case of wind, neighbors are selected according to the strength and direction of the wind.
2. Firebrand propagations: Firebrands are the sparks, burning wood expelled from burning trees during a fire spread, and igniting other trees. Every burning cell produces one firebrand in the direction of the wind. The distance depends on the wind strength. Distances of the firebrand are calculated as follows:

$$\begin{cases} dx = i + w_x[0(\epsilon + \vartheta)] \\ dy = j + w_y[0(\epsilon + \vartheta)] \end{cases}$$

Where, dx and dy are the distances of firebrand projection in directions x and y , $\theta \in [0,1]$ is a random real number, (i,j) are the coordinates of cells, w_x and w_y are the wind directions, ϑ is the wind strength, and ϵ is a parameter adjusted to observe modifications of fire spread.

This very simple model obtains consistent results as represented in Figure 13.5.

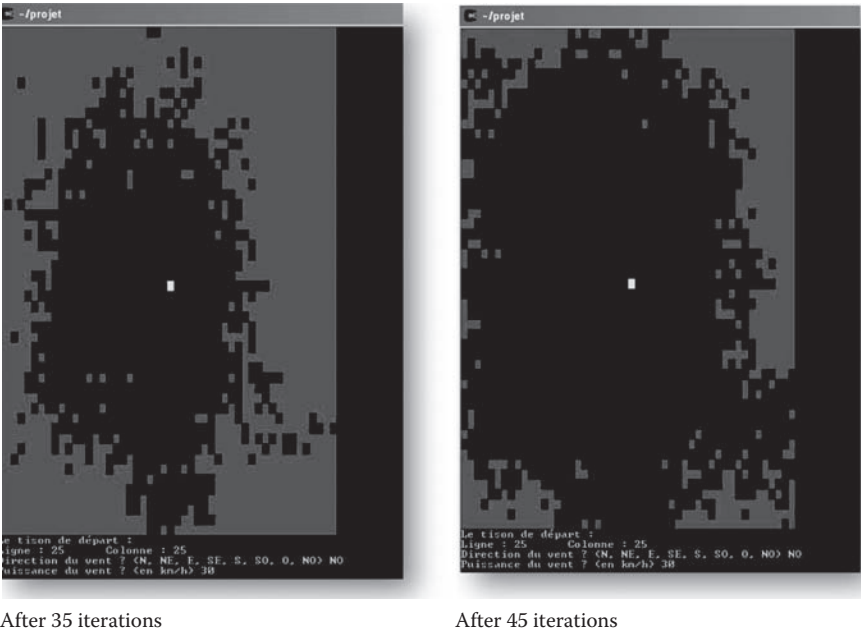


FIGURE 13.6 Fire spread under a wind of 30 km/h in the northwest direction.

13.4.4 EXPERIMENTAL MODEL 3: A MORE SOPHISTICATED APPROACH

The two previous simple models allow the design and implementation of a more complete simulator. In the latter, the finite states used can be refined using a biomass decrease in cells. According to the level of degradation of cells, corresponding states are selected (unburned, heated, burning, and burned). A difficulty relates to the number of parameters to calibrate. This number needs to be limited to explore the full state space of the final model. This model implements the near-to-near propagation of model 1 and firebrand propagation of model 2.

A Model–View–Controller (MVC) design pattern is used to facilitate the development of a graphical user interface (as shown later); [49]. In addition the MVC design pattern allows:

- Separating variable aspects from static aspects of the implementation
- Favoring composition over inheritance in this case where no classification is needed
- Programming with separated interfaces, reducing couplings between objects

Entities of the fire spread simulator according to the Model–View–Controller are:

1. *Model*, which contains the entire simulation engine *View*, which provides the window interface
2. *Controller*, which manages events from the window interface and transfers changes from the view to the model

We have also used an observer pattern to notify the view when we have a state change in the fire model. Indeed, the observer patterns helps in maintaining a list of observers automatically notified of any state change. This is particularly useful in implementing distributed event handling systems. The strategy pattern has also been used to implement a dynamic swap of algorithm particularly when we tested different variants of fire spread. More details on this use for fire spread can be found in [46].

The pseudo-random generator used is the Mersenne Twister [50] from the SSJ Java library [51]. Figure 13.7 depicts the main diagram of the entire implementation. At the design level, the elements have been divided into burnable and unburnable. The class `Simulator` is composed of `Tree(s)` and `Shrub(s)` to be more reusable. In addition, replications are used for representing spectral analyses [26]. By replicating simulations, the entire span of exponential and Gaussian laws can be explored. Every replication is saved. At the end of the simulation, spectral analyses are computed. Spectral analyses correspond to a spatial sum of probabilities of a fire spread experiment. In the `Simulator` class a method for image analysis is implemented. This method converts a plane or remote sensing image in a software `Element(s)` (with basic burnable and unburnable cell states).

The `Simulator` class has two main attributes:

- `Map` representing the propagation domain. It is an array of objects from class `Element`.
- `Element` is an abstract class implementing `Tree` and `Shrub` classes. Each pixel from the map corresponds to an element.

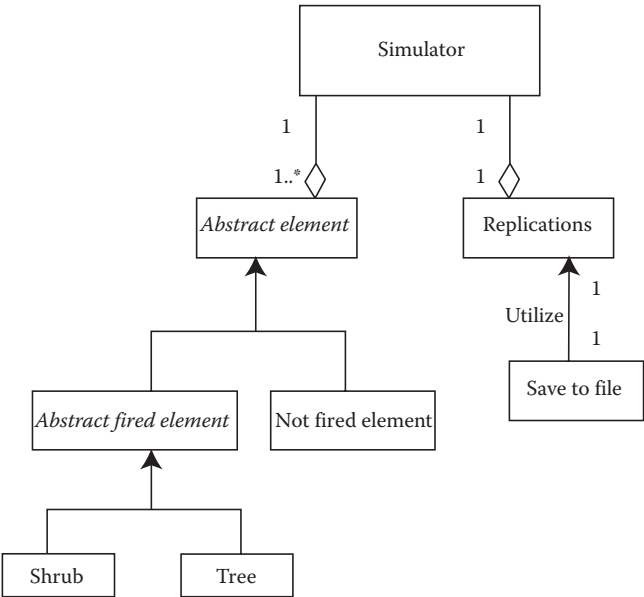


FIGURE 13.7 Simulator class diagram.

There are three main functions:

- `run()`, which corresponds to the main simulation loop (including resume actions).
- `imageProcessing()`, which transforms picture pixels in an identified element (tree or shrub) stored in a table of `Element(s)` of `Map`.
- `burnMap()`, which processes each `Element` in the `Map` that is in its ‘burning’ state’. Each call to `burnMap()` corresponds to an iteration of the simulator.

The structure of this simulator differs from the structure of [11] in many aspects:

- Replications are implemented (i.e., rerun in many simulations). Replication mechanisms still need to be investigated in order to be formally embedded in abstract DEVS simulators.
- No modularity is used for components.

Parameters of the interface are: the strength and direction of the wind, as well as the real-time simulation speed. Actual aerial pictures can be uploaded. Figures 13.8 and 13.9 depict a fire spreading somewhere on the island of Corsica. It can be noticed that the unburnable elements constituted by the road are not burned.

In Figure 13.10 we present two screen shots with the sum of 10 replications of fire propagation. Of course, spatial spectral analysis of this kind is not performed when the interactive simulation mode is active (the only difference between the two replications being the random stream).

Lastly, Figure 13.11 represents a possible fire propagation in 3-D in a virtual environment (near the ISIMA building in France).

13.4.5 EFFICIENCY AND ACTIVITY

We used the Test and Performance Tools Platform (TPTP) <http://www.eclipse.org/tptp/> for the Eclipse development environment to test the execution time performances of the implementation. As depicted in Figure 13.12, we noticed that the refresh method of the simulator class lasts 0.8 s at each execution.

The `refresh` method uses the `isActiveNextTime` method of the `Element` class to determine if this element will be on fire at the next iteration. If that is the case, the element is set to active. As it can be noted in Figure 13.13, this method is not optimized and every element of the propagation domain is tested at each iteration. However, remember the mean relative activity measure. Applying this measure, we find that $\bar{A}$ is below 1%. Therefore, activity tracking is highly recommended here to enhance performances.

As described in Figure 13.14, a new method `setActiveNextTime()` is used to embed a `HashSet` containing the active elements that are implemented. Then, the method `refresh()` only browses the now active elements through the list

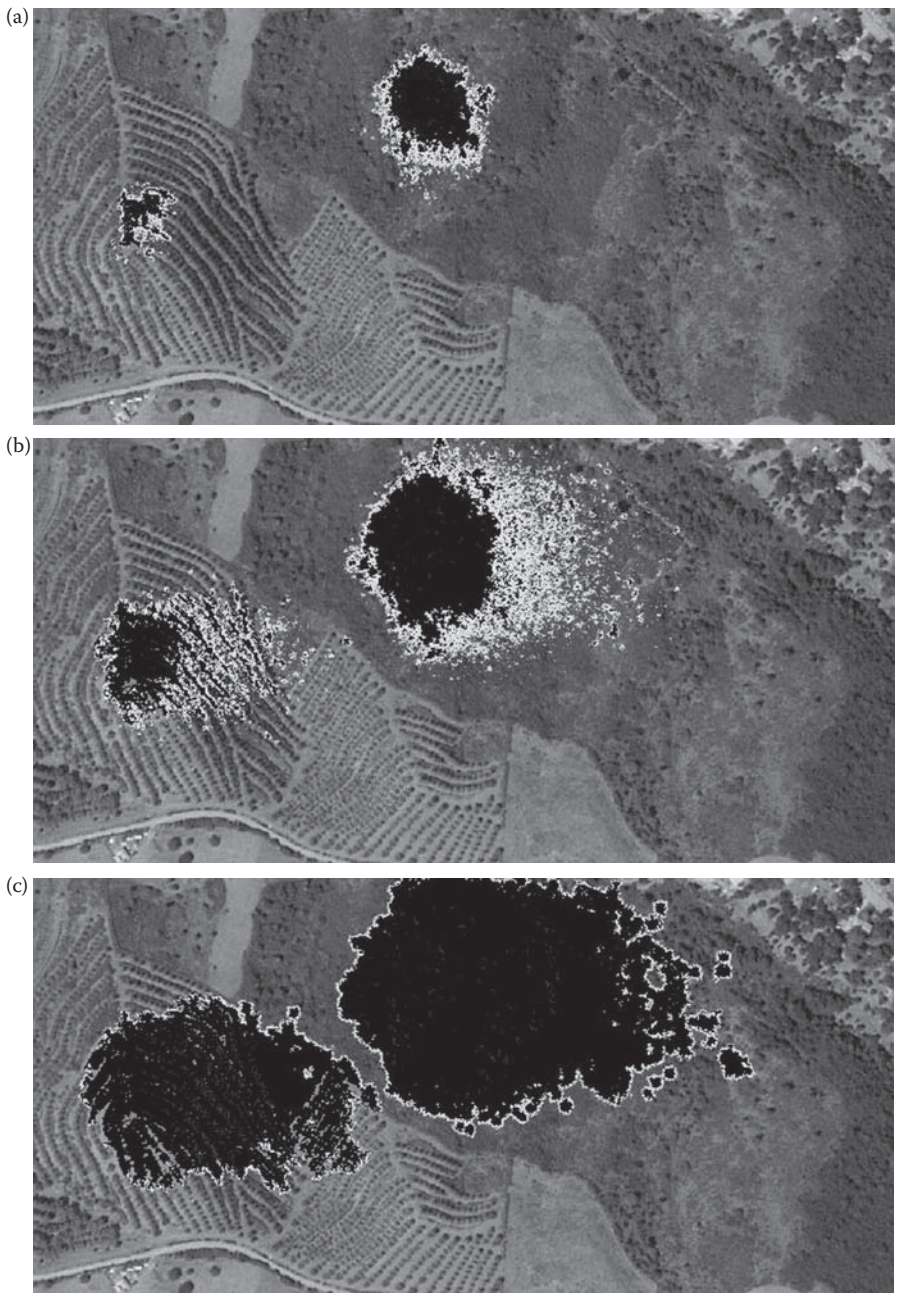


FIGURE 13.8 An example of fire spread in a visual interactive simulation. (a) After 71 iterations, with a slight wind blowing south, with two initial ignitions; (b) after 91 iterations, a strong wind blowing east was implemented after the 71st iteration; and (c) after 125 iterations, without wind since the 91st iteration.

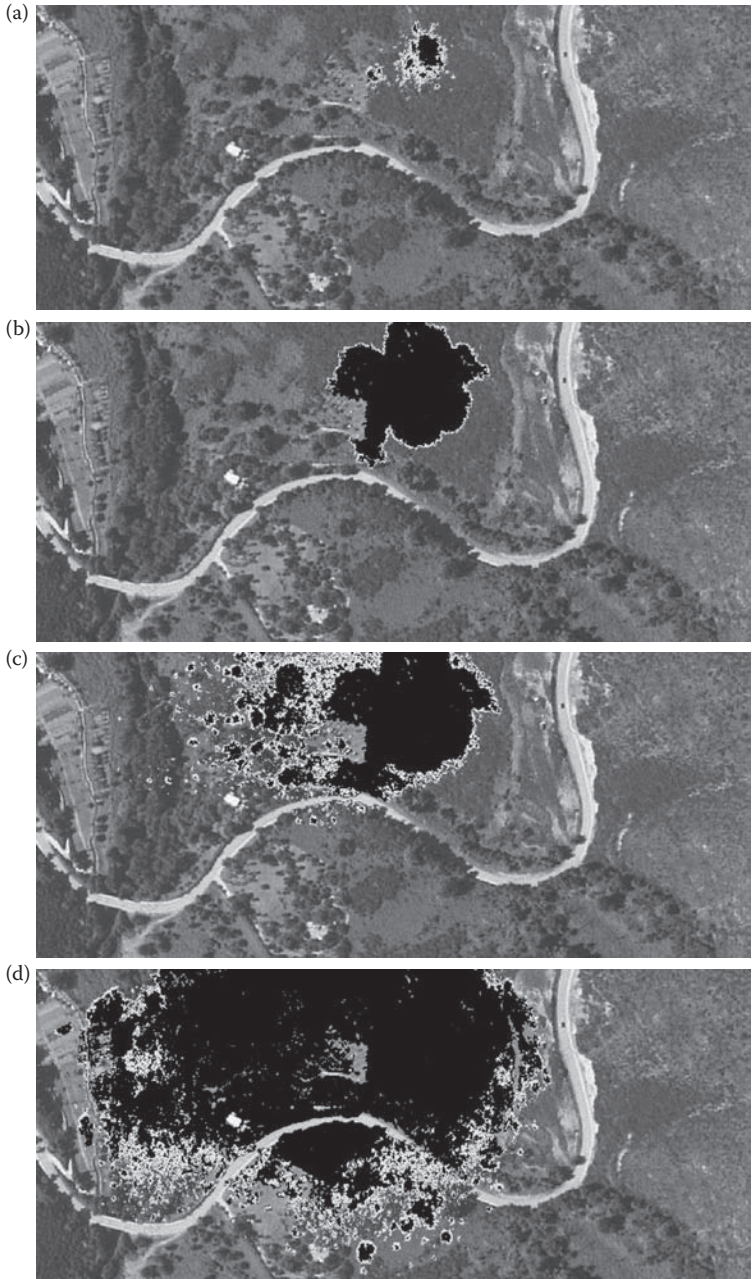


FIGURE 13.9 An example of fire spread. Notice that the fire does not burn the road except when there are trees over the road. (a) After 41 iterations, with a slight wind in southwest direction; (b) after 120 iterations, without wind since the 41st iteration; (c) after 140 iterations, with a very strong wind in the west direction since the 120st iteration; and (d) after 172 iterations, with a slight wind blowing south, since the 140st iteration.

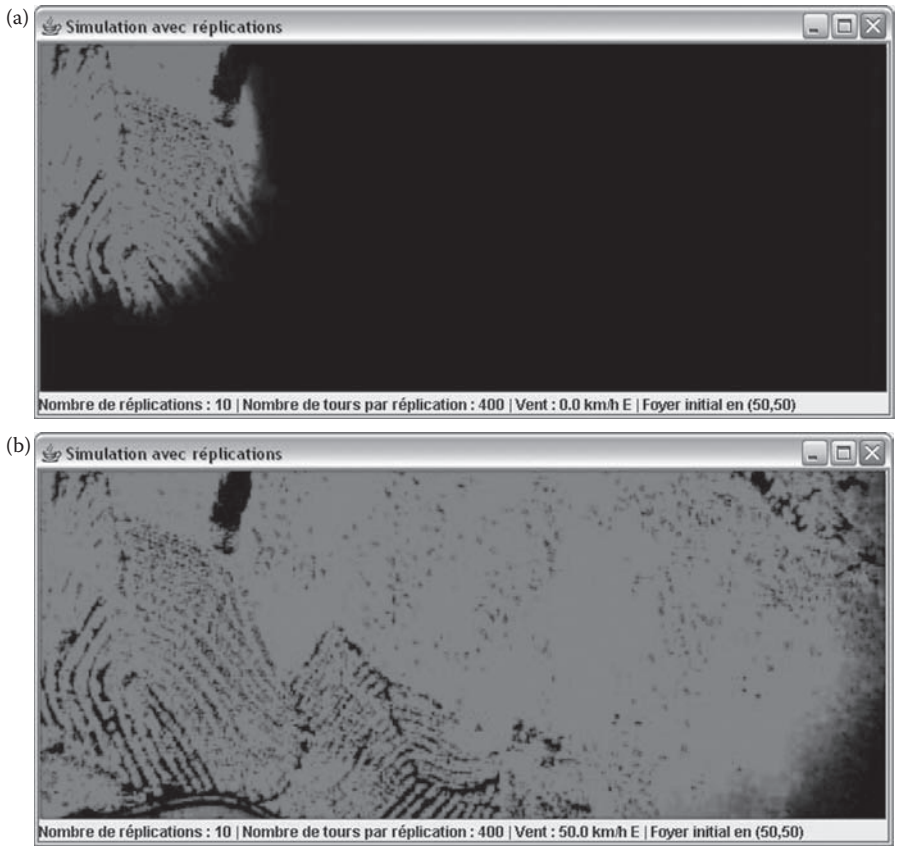


FIGURE 13.10 (a) Ignition on the top left corner after 10 replications of 400 iterations each without wind. (b) Ignition on the top left corner after 10 replications of 400 iterations each with a wind of 50 km/h in east direction

elementsActiveNextTime. After implementing the activity tracking method, execution times have been reduced by a factor of 400! Even with this kind of elementary modeling we see the value of the activity tracking approach.

13.5 DEVS DISCUSSION

In this chapter, first the activity paradigm was discussed for the design of a large (spatial) distributed system. Through this paradigm, the modularity aspects and efficiency measures were explicitly articulated. Then, a nonmodular activity tracking model of fire spread was implemented. Though more formal work is still required to extend usual DEVS abstract simulators to replication-based problems, advantages of stochastic modeling for implementing spatial constraints could be presented. In several cases, stochastic modeling is the only strategy available to represent the spatial distant and discrete interactions.



FIGURE 13.11 Wind in the east direction of 121 km/h.

>Package	Base Time (seconds)	Average Base Time (seconds)	Cumulative ...	Calls
Element	17.836966	0.000003	17.836966	5272...
Element()	0.774271	0.000003	0.774271	239553
isActive() boolean	5.666641	0.000003	5.666641	1676...
isActiveNextTime() b...	11.389609	0.000003	11.389609	3353...
setActive(boolean) v...	0.000779	0.000004	0.000779	197
setActiveNextTime(b...	0.005867	0.000003	0.005867	2061
Ellipse	0.001024	0.000008	0.001024	128
FiredElement	5.043482	0.000004	6.031027	1398...
Forest	19.780943	0.520551	40.985431	38
burn() void	5.926146	0.846592	12.207322	7
Forest.java.awt.imag...	0.031390	0.031390	5.107648	1
imageProcessing.jav...	1.569360	1.569360	5.076259	1
process() void	0.000320	0.000023	35.877760	14
refresh() void	12.253706	0.875265	23.670118	14
setDefaultAllParam...	0.000022	0.000022	0.000022	1
InfluencedArea	0.162283	0.000019	0.366520	8745
NotFiredElement	1.956256	0.000004	2.177088	553994
Shrub	0.100063	0.000006	0.299124	16295
Simulator	12.989674	0.000004	77.587404	3622...
Tree	1.050821	0.000007	2.523944	158321
erifon.util	1.615908	0.000062	9.802466	26017

FIGURE 13.12 Performances report of the simulation methods.

This work leveraged the observation that the formalization of DEVS for stochastic modeling is possible. In [52], a formal specification of pseudo-random generators is described. Pseudo-random generators being deterministic, DEVS describes stochastic models in terms of systems embedding pseudo-random generators. Recently, in [53], a formal extension of DEVS has been provided. A DEVS-based description of stochastic models through STDEVS would allow the modeling and simulation community to formally and explicitly explore this long existing

```
public void refresh()
{
    for(int i=0; i<size_.width; i++)
    {
        for(int j=0; j<size_.height; j++)
        {
            if(map_[i][j].isActiveNextTime())
            {
                map_[i][j].setActive(true);
            }
        }
    }
}
```

FIGURE 13.13 Refresh() nonoptimized method.

```
public void setActiveNextTime(boolean inactive)
{
    activeNextTime_ = inactive;
    if(inactive && (!container_.elementsActiveNextTime_.
contains(this)))
    {
        container_.elementsActiveNextTime_.add(this);
    }
}
```

FIGURE 13.14 Activity tracking method.

modeling technique. DEVS popularity to a large extent derives from the useful level of definition. It is abstract enough to allow the development of new concepts and structures. It is also precise enough to avoid ambiguities and guide modelers. Since its first book definition [52], many developments have been achieved. A constant critical evaluation of these developments is necessary to enhance the entire framework. However, these developments have to be at the best level of abstraction. Because of the increasing complexity of hardware and software architectures, metalevels (and automatic mappings from higher to lower levels) need to be designed. Considering complex systems as multilevel and highly composed, detecting and mapping simulation structures on activity will increasingly be necessary. Activity tracking mechanisms will allow improving DEVS efficiency and abstraction.

Integrating explicitly modularity and efficiency issues, as well as experimental plans and stochastic modeling strategies in DEVS, (computer) scientists will gain freedom and dispose of a more complete and powerful modeling and simulation framework.

REFERENCES

1. Fishman, G. S. *Monte Carlo: Concepts, Algorithms and Applications*, 2000. New York: Springer, 2000.
2. Kleijnen, J. P. C. *Statistical Tools for Simulation Practitioners*. New York: Marcel Dekker, Inc., 1987.
3. Amblard, F., Hill, D., Bernard, S., Truffot, J., and Deffuant, G. MDA compliant design of SimExplorer, A software to handle simulation experimental frameworks. *Proceedings of the Summer Simulation Conference (SCSC)*, 279–84, July 20–24, Montréal, QC, Canada, 2003.
4. Balci, O. *The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages*, Technical Report SRC-88-009, System Research Center. Petersburg, VA: Virginia State University, 1988.
5. Dahl, O. J., and Nygaard, K., Simula—An algol based simulation language. *Communication of the ACM* 9 (1966): 671–78.
6. Pidd, M. Object-orientation and three phase simulation. *Winter Simulation Conference*, 689–93, 1992.
7. Coquillard, P., and Hill, D. Modélisation et simulation d'écosystèmes—Des modèles déterministes aux simulations à événements discrets, Collection Ecologie, Masson, Paris, 1997.
8. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modelling and Simulation*, 2nd ed. New York: Academic Press, 2000.
9. Wainer, G., and Giambiasi, N. Application of the cell-DEVS paradigm for cell spaces modeling and simulation, *Simulation* 76, no. 1 (2001): 22–39.
10. Hu, X., and Zeigler, B. P. A high performance simulation engine for large-scale cellular DEVS models. *High Performance Computing Symposium (HPC'04), Advanced Simulation Technologies Conference (ASTC)*, 3–8, 2004.
11. Muzy, A., and Nutaro, J. J. Algorithms for efficient implementation of the DEVS & DSDEVS abstract simulators. *1st Open International Conference on Modeling and Simulation (OICMS)*, Clermont-Ferrand, France, 2005.
12. Lee, W. B., and Kim, T. G. Simulation speedup for DEVS models by composition-based compilation. *Summer Computer Simulation Conference*, 395–400, 2003.
13. Muzy, A., Innocenti, E., Hill, D., and Santucci, J. F. Optimization of cell spaces simulation for the modelling of fire spreading. *36th Annual Simulation Symposium, IEEE/SCS/ACM*, 289–96, Orlando, FL, 2003.
14. Shiginah, F. A. S. B. *Multi-Layer Cellular DEVS Formalism for Faster Model Development and Simulator Efficiency*. PhD Thesis. Electrical and Computer Engineering Department, Tucson, AZ: University of Arizona, 2006.
15. Sun, Y., and Hu, X. Partial-modular DEVS for improving performance of cellular space wildfire spread simulation. *Proceedings of the 2008 Winter Simulation Conference (WSC'08)*, 2008.
16. Wainer, G. CD ++: A toolkit to develop DEVS models, *Software—Practice & Experience* 32, no. 13 (2002).
17. Muzy, A., Nutaro, J., Zeigler, B. P., and Coquillard, P. 2008. Modeling and simulation of fire spreading through the activity tracking paradigm, *Ecol. Model.* 219, nos. 1–2 (2008): 212–25.
18. Lewis, M. A. Spread rate for a nonlinear stochastic invasion, *Journal of Mathematical Biology* 41, no. 5 (2000): 430–54.
19. Coquillard, P. Simulation of the cyclical process of heathlands: Induction of mosaics structures, evolution to irreversible states, *Ecological Modelling* 80 (1995): 97–111.

20. Coquillard, P., and Fain, J. Monte-carlo simulation of colonization by pinus sylvestris of heathland on degenerate phase on volcanic substrate. In *Functioning and Dynamics of Natural and Perturbed Ecosystems*. Edited by D. Bellan-Santini, G. Bonin, and C. C. Emig, 274–89. Paris: Lavoisier Publishing, 1995.
21. Hill, D., Coquillard, P., De Vaugelas, J., and Meinesz, A. An algorithmic model for invasive species application to *Caulerpa taxifolia* (Vahl) C. Agardh development in the North–Western Mediterranean Sea, *Ecological modelling* 109 (1998): 251–65.
22. Jung, P., and Mayer-Kress, G. Spatiotemporal stochastic resonance in excitable media, *Phys. Rev. Lett.* 74 (1995): 2130–33.
23. Falcke, M., Tsimring, L., and Levine, H. Stochastic spreading of intracellular Ca^{2+} release, *Physical Review* 62, no. 2 (2000): 2636–43.
24. Ellis, J., Mavromatos, N. E., and Nanopoulos, D. V. Quantum-gravitational diffusion and stochastic fluctuations in the velocity of light, general relativity and gravitation, *General Relativity and Gravitation* 32, no. 1 (2000): 127–44.
25. El Bitar, Z., Lazaro, D., Breton, V., Hill, D., and Buvat, I. Fully 3D Monte Carlo image reconstruction in SPECT using functional regions, *Nucl. Instr. Meth. Phys. Res.* 569 (2006): 399–403.
26. Hill, D. Modélisation des processus d’expansion: Application à *Caulerpa taxifolia*”, Conférence Internationale sur la “Dynamique des Espèces invasives,” Mars 13–15, 1997, Paris, Académie des Sciences. Tec & Doc, 219–30, 1997.
27. Hill, D., Mazel, C., and Coquillard, P. Integrating V&V in the object-oriented life cycle of ecological modelling simulation projects. *Proceedings of the European Simulation Symposium 96*, Vol II, 21–25, October 24–26, Genova, Italy, 1996.
28. Barros, F. J., and Mendes, M. T. Forest fire modelling and simulation in the DELTA environment, *Simulation Practice and Theory* 5, no. 3 (1997): 185–97.
29. Karafyllidis, I., and Thanailakis, A. A model for predicting forest fire spreading using cellular automata, *Ecological Modeling* 99 (1997): 87–97.
30. Hernández Encinas Hoya White, S., Martín del Rey, A., and Rodríguez Sánchez, G. Modelling forest fire spread using hexagonal cellular automata, *Applied Mathematical Modelling* 31 (2007): 1213–27.
31. Yassemin, S., Dragičević, S., and Schmidt, M. Design and implementation of an integrated GIS-based cellular automata model to characterize forest fire behaviour, *Ecological Modelling* 210 (2008): 71–84.
32. Weber, R. O. Modelling fire spread through fuel beds, *Progress in Energy and Combustion Science* 17 (1990): 67–82.
33. McArthur, A. G. Weather and grassland fire behaviour. *Australian Forest and Timber Bureau Leaflet* 100, 1966.
34. Rothermel, R. C. *A Mathematical Model for Predicting Fire Spread in Wildland Fuels*. Washington, DC: USDA, Forest Service Research, 1972.
35. Richards, G. D. An elliptical growth model of forest fire fronts and its numerical solution, *International Journal of Numerical Method Engineering* 30 (1990): 1163–79.
36. Albini, F. A. A model for fire spread in wildland fuels by radiation. *Combustion Science and Technology* 42 (1985): 229–58.
37. Hargrove, W. W., Gardner, R. H., Turner, M. G., Romme, W. H., and Despain, D. G. Simulating fire patterns in heterogeneous landscapes, *Ecological Modelling* 135 (2000): 243–63.
38. Kleijnen, J., and Groenendaal, W. V. *Simulation a Statistical Perspective*. New York: Wiley, 1992.
39. Jimenez, E., Hussaini, M. Y., and Goodrick, S. Quantifying parametric uncertainty in the Rothermel model, *International Journal of Wildland Fire* 17, no. 5 (2007): 638–49.
40. Gu, F., and Hu, X. Towards applications of particle filter in wildfire spread simulation. *Proceedings of the 2008 Winter Simulation Conference (WSC’08)*, 2008.

41. Vorob'ov, O. Y. Random set models of fire spread, *Fire Technology*, 32, no. 2 (1996).
42. Porterie, P., Zekri, N., Clerc, J. P., and Loraud, J. P. Modeling forest fire spread and spotting process with small world networks, *Combustion and Flame* 149, nos. 1–2 (2007): 63–78.
43. Vasconcelos, M. J., Pereira, J. M. C., and Zeigler, B. P. Simulation of fire growth using discrete event hierarchical modular models, *EARSeL Advances in Remote Sensing* 4, no. 3 (1995): 54–62.
44. Ameghino, J., Troccoli, A., and Wainer, G. Models of complex physical systems using Cell-DEVS. *Proceedings of Annual Simulation Symposium*, Seattle, WA, 2001.
45. Muzy, A., Innocenti, E., Wainer, G., Aiello, A., and Santucci, J. F. Specification of discrete event models for fire spreading, *Simulation: Transactions of the Society of Modeling and Simulation International*, SCS 81 (2005): 103–17.
46. Innocenti, E., Silvani, X., Muzy, A., and Hill, D. A software framework for fine grain parallelization of cellular models with OpenMP: Application to fire spread., *Environmental Modelling & Software* 24 (2009): 819–31.
47. Natimo, L., Hu, X., and Sun, Y. DEVS-FIRE: Towards an integrated simulation environment for surface wildfire spread and containment, *Simulation* 84, no. 4 (2008): 137–55.
48. Sun, Y., and Hu, X. Performance measurement of DEVS dynamic structure on forest fire spread simulation. *Proceedings of the 14th AI, Simulation and Planning in High Autonomy Systems (AIS 2007)*, 2007.
49. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design patterns: Elements of reusable object-oriented software. Reading, PA: Addison Wesley, 1994.
50. Matsumoto, M., and Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *Proceedings of the 29th Conference on Winter Simulation*, 127–34, 1997.
51. L'Ecuyer, P., and Buist E. Simulation in Java with SSJ. *Proceedings of the 2005 Winter Simulation Conference*, 611–20, 2005.
52. Zeigler, B. P. *Theory of Modelling and Simulation*, 1st ed. New York: Wiley Interscience, 1976.
53. Kofman, E., and Castro, R. *STDEVS. A Novel Formalism for Modeling and Simulation of Stochastic Discrete Event System*. Technical Report LSD0603, LSD, Universidad Nacional de Rosario. Accepted in AADECA 2006.
54. Muzy, A., and Zeigler, B. P. Introduction to the activity tracking paradigm in component-based simulation, *The Open Cybernetics and Systemics Journal* 2 (2008): 48–56.

14 Environmental Models in DEVS: Different Approaches for Different Applications

*Jean-Baptiste Filippi, Teruhisa Komatsu,
and David R.C. Hill*

CONTENTS

14.1	Introduction	357
14.2	A DEVS Cellular Automaton Model	360
14.2.1	Simulation Results	363
14.3	Environmental Modeling with Moving Particles and Lagrangian Tracking	366
14.3.1	The Sargassum Algae Model	366
14.3.2	Discrete-Event versus Discrete-Time Integration	370
14.3.3	Random Dispersion Calculations	371
14.3.4	Simulation of Algae Dispersion	371
14.3.5	Efficient Asynchronous Simulation of Fire Front Evolution	373
14.3.6	Front Tracking	374
14.3.6.1	Shape Behavior	377
14.3.7	Validation on a Real Fire Accident	378
14.3.7.1	Simulation Parameters	379
14.3.7.2	Simulation with Optimal Fuel Parameters	379
14.4	Coupled Fire-Atmospheric Model	380
14.4.1	Experimental Set-Up and Simulation Results	382
14.5	Conclusion	384
	Acknowledgments	384
	References	384

14.1 INTRODUCTION

The term “environmental” is a general denomination for all systems that relate to nature in some sense. In particular, systems are considered environmental in sense that they represent models to understand dramatic natural events, land planning, or to simulate the human response to a new environmental configuration.

Such environmental systems can span different temporal and spatial scales (molecular to geological) and their complexity often results from multiple interactions between parts of a system that operate on those different scales. A typical example of such a system would be a catchments basin, involving pollution, water cycles, erosion, and vegetation dynamics. Modeling such systems may require defining sub-models of the hourly molecular degradation of the pollutants, but also the impact of this pollution on the development of vegetation and the subsequent effect of vegetation on erosion.

Environmental science is devoted to the study of environmental systems, an inherently multidisciplinary science that is usually composed of four main disciplines [1–4]: Atmospheric Science, Ecology, Environmental Chemistry, and Geosciences.

Simulation is a tool used by environmental scientists to monitor and understand the complex behavior of the earth's systems. Most observations are spatial distribution of the physical, biological, and mineral compounds of the earth. With these observations used as inputs for the model to forecast future distributions and global evolution, most models developed to study these environmental systems are specifically spatially distributed.

With the computational and communication resources that have become available over the past decade, it is now possible to perform simulations of such complex systems to understand, observe, and predict their dynamics. In addition, there is a powerful trend for increasing observational data as well as data collection methods to become available, employing, for example, satellites or sensors directly embedded in a studied environment.

The consequences of these technological advances are the increase in resolution and quality of the analysis and prediction of most atmospheric and ocean models. But atmospheric science is not the sole beneficiary of these advances and the modeling and simulation of other environmental systems is now possible. Nevertheless, most of those developments require specific modeling approaches adapted to the underlying behavior of the studied system.

Each of these approaches is adapted from a view of the system, a view that can depend on the mental representation, knowledge and limitations of the modeler, and implementation issues and availabilities of the software. These approaches define how experiments will be conducted, and which methods are available to conduct each experiment. In particular there are different ways to handle time. Indeed, environmental systems also have the specificity that allows them to be coupled with systems that are acting on different time scales and at different rates. Nevertheless, most numerical simulation methods consider time as synchronous: every behavior has to be updated at the same discrete time. Because of this discrete time approach, most equations governing behavior are designed with a “time step” in mind.

An important aspect of the event approach in time management is that models are not constrained by this limitation; time is handled by discrete events that can occur at anytime during simulation. Because of the event driven approach, simulation does not have to be synchronous and processes occurring on different time scales can be processed naturally.

The modeling formalism is also a concern. Because all processes in environmental systems are tightly coupled, a modeler has to find a way to describe the composition

of multiple submodels in a formal way that can be interpreted by a human as well as a machine. Strongly organized complexity also exists between heterogeneous components that interact at various temporal and spatial scales.

Hierarchical modeling is an efficient way to describe the coupling between these heterogeneous components. The Discrete Event System Specification (DEVS) is a modular and hierarchical formalism that can be used to define such models. Nevertheless, the use of this formalism does not help when a modeler has to find the appropriate modeling approach to represent a particular system. For example, complex systems can be viewed as stochastic, fuzzy, or deterministic, with a dynamic or static structure, distributed in cells, particles, agents, networks, or fronts.

All these different model expressions have found multiple ways to be formalized in DEVS variants over the past decade and they are now available as a library with an interoperable formal description [6]. Much of this work has led to the development of a variety of DEVS-based methods that are related to a specific application domain or to specific model structures. Some of those DEVS-based methods are explicitly able to handle spatial distribution of models or model parameterizations and are well suited for the simulation of most environmental systems.

In order to perform numerical experiments on spatial models it is necessary to discretize space, and to associate a frame of reference with the model equations. Two main ways exist to build such a frame of reference; Eulerian, where the calculations are made at specific locations through which model variables are integrated over time; or Lagrangian, where the movement of models is integrated over time.

The choice between Eulerian and Lagrangian approaches depends on the focus that is made on specific entities of the studied system. The Eulerian approach is more appropriate if the expected result is a map of quantities (such as insect concentration, or pollutants in a watershed). If the focus is on the actual position of a system, or multiple entities in the system (such as ants or drifting seaweeds patches), the Lagrangian approach with spatial agents may be more suited. Eventually, if the system is evolving over an extensive area but the actual reacting area is small and requires a high resolution at a specific time (such as forest fire fronts), a hybrid Eulerian/Lagrangian approach can be used to optimize computing time so as to still handle an extensive area with high local precision. When discrete distant interactions can randomly occur between the simulated entities or agents, a stochastic simulation under spatial constraints can be proposed, but that is beyond the scope of this chapter.

Instead, this chapter presents three different environmental systems and their applications to illustrate the multiplicity of options available to the modeler, all three are using a different approach to handle space, but all are sharing the use of DEVS. The first application is a use of cellular automata (CA) to simulate the dynamics of insects. Next, an agent-based method is selected for a simulation of the spread of drifting algae in the China Sea followed by a stochastic inverse-simulation of originating area of the same species. Finally a front tracking method applied to forest fire propagation is presented and tested on a real case scenario. The fire simulation model is later coupled to an operational atmospheric model, illustrating the ease of implementing a hybrid model with DEVS.

14.2 A DEVS CELLULAR AUTOMATON MODEL

The most common Eulerian approach of modeling space in DEVS is using a cellular automaton analogy; where every cell represents an area of a studied system. This approach is not necessarily suited to model all systems if the interest is not to observe the state of the entire area, but is focused on local movement within a large area.

Cellular Automata (CA) are widely used to model different kinds of complex systems. A cellular automaton is usually represented by a matrix field of cells executing the same program simultaneously, receiving as input the state of their neighboring cells [7,8]. CA prove to be well suited to model systems that are spatially distributed, by defining the behavior of an atomic portion of the system (a cell) in relation to a distant or direct neighborhood such as presented in Figure 14.1.

Nevertheless, applications of the standard CA method to complex environmental systems proved to be difficult because of the intrinsic limitations such as synchronous execution for all cells, a single state for each cell, and a fixed neighborhood. The DEVS formalism resolved all those limitations by offering a formal framework for defining asynchronous behavior and complex structure CA. Various extensions of CA in DEVS are available, most of them focus on fire spread application [9,10]. Most notably, the work of Wainer on the Cell-DEVS formalism [11] is particularly well suited to model environmental systems as it is generic and applicable to all fields; not only is Cell-DEVS a specific formalism, it also provides a full syntax, interpreter, and development tools that make the use of CA in DEVS easier. An extensive review of Cell-DEVS applications is available in Wainer [12] with detailed descriptions of CA environmental models that are beyond the scope of this chapter. Instead a simple problem and an elementary model will be presented, without an extensive description of the specific simulation platform or formalism.

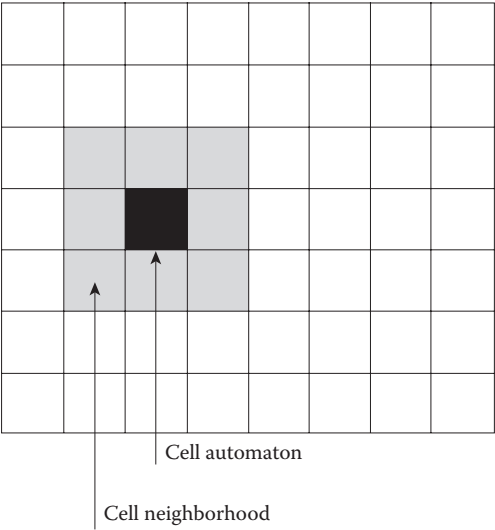


FIGURE 14.1 Cellular automata with a moore neighborhood.

A model of the dynamics of fruit flies is presented in order to illustrate the use of CA in DEVS. The medfly (*Ceratitis capitata*) is one of the most serious economic pests of fruit and vegetables. If no control methods are used, the medfly can infest all susceptible fruit such as apricots, grapefruits, and peaches and to a lesser extent fruit such as apples and clementines. To limit the medfly population, a specific and environmentally nonpolluting method of medfly control called Sterile Insect Technique (SIT) is increasingly used. This technique consists of releasing a large number of sterile males over a sufficient period of time at the optimal location. One of the main objectives of the model is to estimate the geographical distribution of the adult medfly. This model is to be used for guidance in implementing eradication procedures and preventing the spread to other locations.

The onset of medfly activity is temperature dependent. In southern France (Corsica Island) the medfly is active in late spring, summer, and autumn, when temperatures exceed an average climatic condition. The medfly is dormant over the winter as eggs, larvae (in fruit), or as pupae in the ground. As temperatures increase in spring, adults begin to emerge from the ground and flies become active.

Because cellular models are used in this application, it is not an actual medfly model that is to be defined, but a model of an orchard (a cell, with or without fruit) reacting to medfly infestation. The behavior for the orchard is presented in Figure 14.2, the orchard can have three states, empty (fruit are not present or green), ripe (fruit can become infested), and infested (fruit flies are present in the cell).

This model has been proposed in Bernardi and Filippi [13] the following structure in DEVS:

$$< X \{N, S, E, W, in1, in2\}, Y \{N, S, E, W, out\}, S\{empty, ripe, infested\} > .$$

Moreover, the model has the following parameters necessary to handle the transitions between states: {Host, Neggs, Nlarvae, Npupae, Nadults, Aeggs, Alarvae, Apupae, Aadults, Afood}

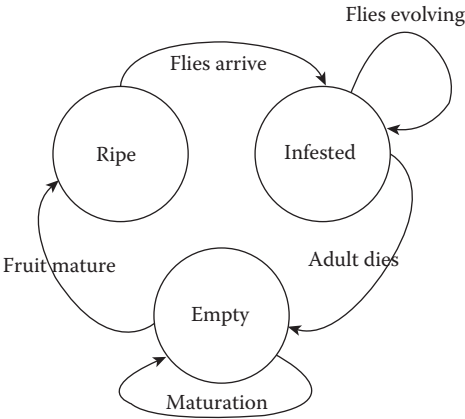


FIGURE 14.2 Fruit fly model behavior.

- N, S, E, W correspond to the North, South, East, and West ports of their neighborhood, respectively, all these ports are of use to exchange flies between cells
- in1 is an input port receiving day time temperature
- in2 is an input port receiving flies (for insemination or initialization)
- out is an output port emitting flies (for general simulation output and coupling to other models)
- Empty, ripe, infested, depends on the ripening period of the host tree and fruit flies attacks
- Host is the species host trees {apricots, grapefruit, peaches, apples, clementines, other} where other corresponds to a medfly insensitive cell
- Neggs, Nlarvae, Npupae, and Nadults are value counts of the population at each stage
- Aeggs, Alarvae, Apupae, and Aadults correspond to the maturation of the population at each stage
- Afood corresponds to the number day of diet for adult's population

The model implemented over this structure is the CLIMEX (a Computerized system for matching cLIMates in Ecology) model [14] that defines the development of cells as temperature dependent. The behavior of the medfly is specified in terms of scattering and development. Scattering occurs when the fly is an adult, as such, the fly is able to travel a given distance (a given number of cells), looking for mature fruit in unaffected areas. When a fly enters an unaffected area where adult fruit is available, it colonizes the area until there is no more fruit available, or until eggs are laid in the fruit.

The development function integrates the average day temperature to calculate the maturation of the fly. Typically a fruit fly is known to become active when a temperature of more than 10°C is reached. From the day the temperature is reached, it will change state every 120 degrees/day, meaning that it will take 10 days at an average temperature of 12°C for an egg to become a larvae. Moreover, each fruit species has different maturation dates that define when each cell needs to be updated.

Translated into DEVS, this gives the following functions:

- The λ function (output) sends the quantity of adult flies to the neighboring cells given by scattering.
- The δ_{ext} function (input) receives a given number of adults from the neighboring cells, and the day temperature, if no flies are present in the cell, the temperature event is not scheduling an update.
- The δ_{int} function (internal) updates the states according to development and integrates the temperature per day value if flies are present.
- The ta function (time advance) defines the time to the next self-activation of the cell according to the number of adults and maturation time of the fruit (thus defining propagation speed). The elapsed time is infinite if there is no adult in the cell, otherwise it is equal to the next maturation time of the fruit, marking the cell for fly availability.

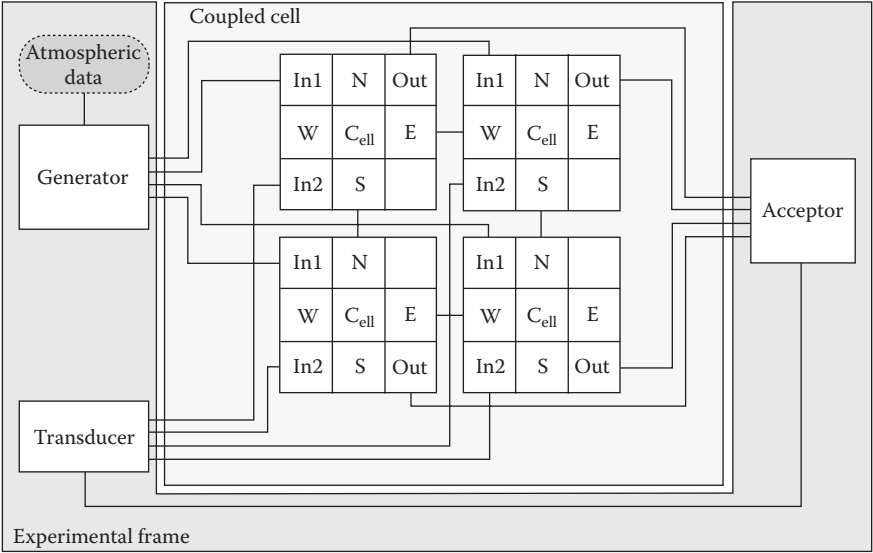


FIGURE 14.3 Couplings of the fruit fly model.

This model has been implemented in Java in the JDEVS environment [15]. Couplings between cells are presented in Figure 14.3, the structure is static and all cells are connected to each other by their (N, S, E, W) ports in a Coupled cell component.

The data preprocessing necessary to run the simulation is extracted from Geographical Information Systems, as a matrix of actual planted fruit trees for a given area (Host parameter). Nevertheless, for the same orchard we can have different atmospheric and initial flies' population conditions. Such conditions are controlled in the experimental frame [6] of the model. The link between the experimental frame and the model are shown in Figure 14.3.

An experimental frame is composed of three components; the *generator* that defines the condition of the simulation; the *acceptor* that verifies the simulation outputs; and the *transducer* that observes the simulation and may enable some feedback.

In this experimental frame the *generator* translates the atmospheric temperature data to DEVS events that are sent to all cells by *in1* ports. The *acceptor* logs the results and verifies that no final or blocking condition is met (no mature fruit ever present or all cells infested with exponential growth of adult flies). The *transducer* is the JDEVS graphical user interface (Figure 14.4) that shows the actual evolution of the infestation and enables the input of external infestation through *in2* ports.

14.2.1 SIMULATION RESULTS

Simulation is initialized in winter when pupae are dormant as larvae in clementines and grapefruit. The studied zone is an area of 3.5 km² divided in 350 cells of 100 m × 100 m each (25 × 14 cells) initialized with a typical distribution of apple (20%),

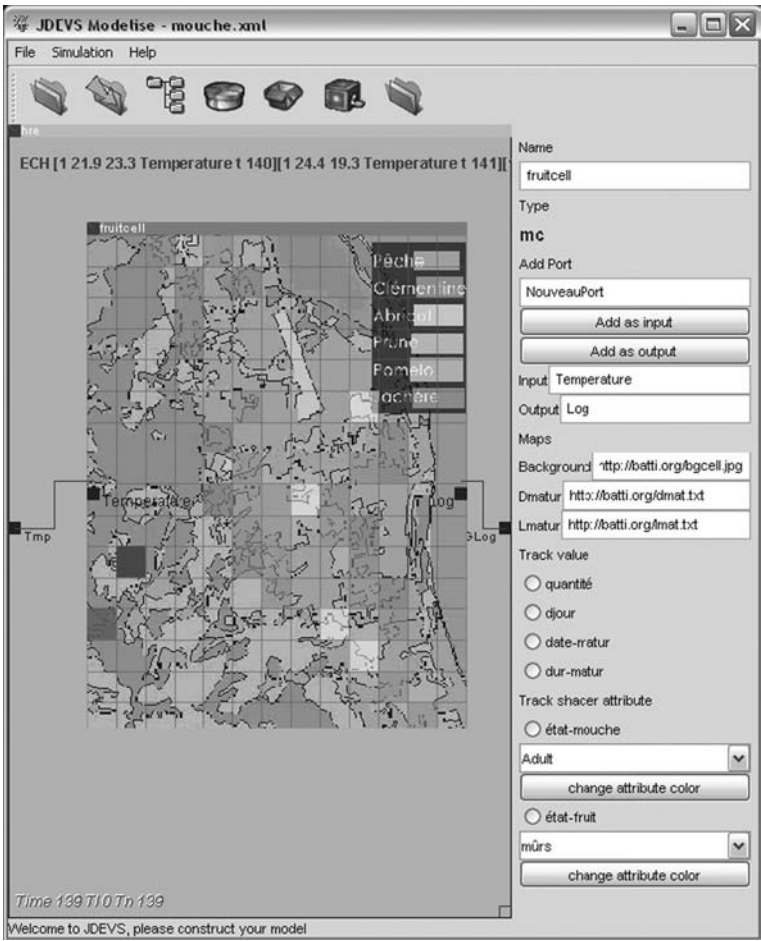


FIGURE 14.4 JDEVS experimental frame transducer.

peaches (10%), clementines (40%), and grapefruit (30%) distributed randomly in large patches following this distribution, with a geographical bias in order to distribute more grapefruit/clementines on the west side and apples/peaches on the east.

A cellular model is instantiated during the initialization, these models are coupled automatically with the neighborhood, a description of the structure of the medfly cellular model can be found in Filippi and Biscambiglia [15].

Temperature events are received each day for each cell, with a dataset of averaged daily values sampled from the period 1995–2002.

Figure 14.5 presents result maps of the simulation at different periods of a year. In areas with no control over the medfly, a large number of fruits can be infected; in fact, observations have been made of the average quantity of infested fruits for three species by the Observatoire de l'environnement de la Corse [16] (peaches, grapefruits, and apples).

Table 14.1 gives the results observed and simulated average.

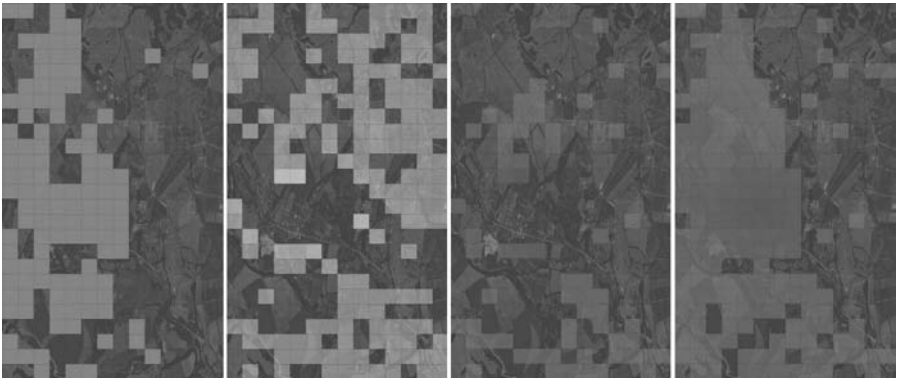


FIGURE 14.5 Flies propagation simulation. Showing initial Pupae concentration in Clementine fields in January (left), then after maturation in June as larvae in apples and peaches (center left). At the end of the season, the adult population has decreased because of lack of food in September (center right), and eventually the adult population is showing different levels of concentration (right) when grapefruits and clementines become mature (November) and apples are still available.

TABLE 14.1
Simulation Results of the Fruit Flies Model

	Peaches	Grapefruit	Apples
Simulation	85.3%	91.4%	24.8%
Observed	60%	80%	20%

We can see from Table 14.1 that the simulation is overestimating the infestation. Nevertheless, qualitatively, it is able to capture the overall trends, and in particular, provide a map of most of the infected areas. The simulation has been performed for 1 year, with a daily temperature broadcasted by the generator. Simulation time for the experiment is 340 seconds on a 2 GFlops capable computer (Intel Pentium 4). Around 4,000 internal transitions of cells were required for this simulation.

There are two main advantages of using DEVS in this application. First, DEVS provides a clear formalism for the modeler knowing where the behavior has to be implemented. The second advantage is that the model is not updating every cell on a regular time base, even if temperature events are input on a regular daily time step, a cell is only active if a fruit is mature and if it contains flies. Instead of checking every cell at every time step, the cell is woken up by a self generated event scheduled on the maturation date of the fruit. In the application presented, a daily time stepped simulation of the case would have required about 12,000 transitions because all cells would have been activated all the time. Nevertheless, some initial efforts are needed to define the model compared to a straightforward implementation, efforts that may delay the availability of simulation software in the short term.

As seen in this example, CA is appropriate when there is a large area to be observed and for which each piece can be quantified and modeled. As such, CA proposes a

so-called Eulerian view of the system. When the focus is to be put on tracking the position of a moving model on a fixed environment, the Lagrange approach is more appropriate. The next section presents a DEVS implementation of Lagrange motion of particles.

14.3 ENVIRONMENTAL MODELING WITH MOVING PARTICLES AND LAGRANGIAN TRACKING

This section presents the use of a DEVS defined model of particle tracking of drifting seaweeds. Particle tracking methods have proved to be the most robust and common approach to identifying drifting paths [17]. Trajectory and spreading models normally use resolutions of the Lagrange particle tracking technique [18] where particles simulate material transported by the water currents. The simulation of oil spills is one of the best known applications.

Nevertheless, this technique has to be used carefully as it is known that the common first order approach used to solve Lagrange equations is very sensitive to time stepping [18] and might induce numerical errors that cause erroneous drift and exaggerate the spread. The application on seaweeds presented hereafter proposes a DEVS formalization of particle tracking that is less subject to exaggerated drifting, the full description of the model is available in Filippi, Komatsu, and Tanaka [20] and the more recent adaptation to the currents computed by Earth Simulator can be found in Kofman [21].

14.3.1 THE SARGASSUM ALGAE MODEL

Drifting seaweeds play key ecological roles in offshore waters as well, as they play host to attaching or accompanying flora and fauna. Hence, drifting seaweeds constitute a moving ecosystem. With regard to fisheries, the seaweeds are spawning mediums for flying fish, Japanese halfbeak, and Pacific saury. The seaweeds also serve as nursery mediums for larvae and juveniles of some commercially important pelagic fish species, such as yellowtail or jack mackerel [22], which spawn in the East China Sea.

A patch of algae is treated as a particle that has a specific location, area, and biomass. As such, it can be considered as a spatial agent, but we will refer to the patch as a particle. Considering a patch as an agent requires extensive treatment that is itself the subject of a chapter [23]. A patch of seaweed has a lifetime, a position, and a local speed and direction of propagation. It can be in one of four different states such as presented in Figure 14.6: *Attached* if the patch is on shore; *Floating* if the patch has been detached because of changing flow or maturation; *Sinking*, if the seaweed is dying and starts to degrade; *Dead*, if the seaweed has lost all ability to float.

At initialization, the seaweed is given a random potential lifetime (usually 3 months with a deviation of 15 days). The floatability is initially set to 1 (particle at the surface) and is decreasing progressively from 1 to 0 (20 meters depth) in the last 15 days of the algae. An algae that is deeper than 20 meters does not receive enough sunlight and dies.

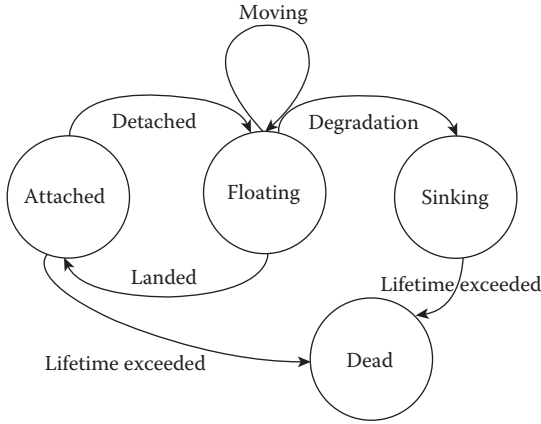


FIGURE 14.6 Algae model behavior.

In the DEVS form this yields the following structure:

- $X: \{(p,v)|(p\{speed\}, v\{x-speed, y-speed, z-speed\})\}$
- $Y: \{(p,v)|(p\{position, status\}, v\{x, y, z\}\{attached, floating, sinking, dead\})\}$
- $S: \{attached, floating, sinking, dead\}, parameters\{x, y, z, biomass, area-size, lifetime\}$
- $\delta_{int}: S \rightarrow S$ updates position in floating state, and move to the next sequential state if appropriate
- ta : Time to next position update, or next state change
- $\delta_{ext}: Q \times X \rightarrow S$ receives new speed information in case of a position or state update
- $\lambda: Q \times X \rightarrow S$ outputs z new position and state

Inputs correspond to a given location (p) and a given propagation (v). Outputs correspond to the actual location, speed, and status that are to be used for the diagnosis.

The coupling of the algae model is presented in Figure 14.7. There are a fixed number of particles for each simulation; all particles are contained in a *coupled ocean* model. In these simulations, an experiment is performed for a given set of ocean flow data. The experimental frame *generator* is linked to each particle as a broker of localized inputs. Flow data is interpolated at the very location of the particle using a bi-cubic method from matrices available from the OFES (Ocean general circulation model For the Earth Simulator) current simulation model [24]. These inputs are processed by a particle in a *floating* state to perform the integration of their advance in the flow.

Outputs are linked to the experimental frame *Acceptor* that logs results and ensures that no particle is traveling outside the boundary of the system (as the flow data is not global). The position for each particle is then sent to the *Transducer* for runtime visualization. A screenshot of the transducer graphical environment is presented in Figure 14.8.

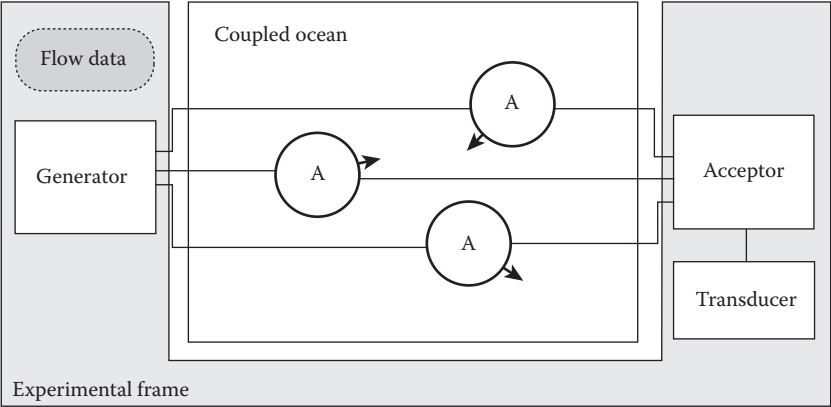


FIGURE 14.7 Couplings of the algae model.

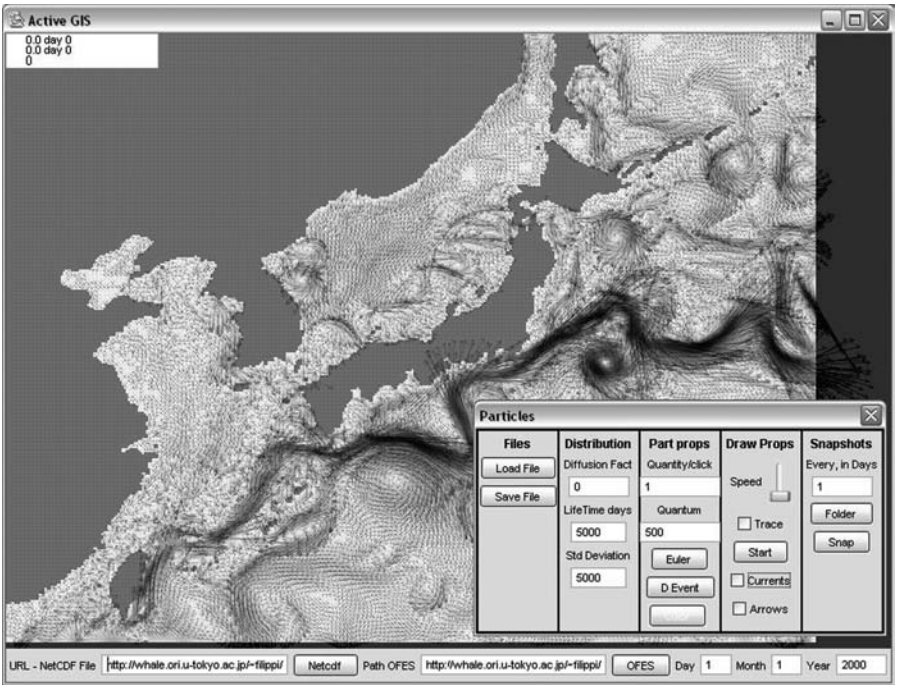


FIGURE 14.8 Experimental frame transducer developed for the particle tracking model.

A singularity of this model is that it calculates trajectories based on the assumption that behavior of drifting seaweeds, like oil spills, can be idealized by a large number of particles that move in three dimensions in a water body by advection and spreading dynamics. To determine the advective velocity field, the simulation uses data from the OFES model [24] that simulates ocean current, surface elevation, temperature, and salinity with very high resolution. OFES is used to force trajectories backward to find

where particles may come from. Velocities resulting from the hydrodynamic model are later used by the spread model to compute trajectories. A Lagrange approach is used for the trajectory resolution, based on the following equation:

$$\frac{dx_i}{dt} = v(x_i, t) + v_d(x_i, t).$$

Where $\mathbf{x}_i$ is the i particle coordinate, $\mathbf{v}_a$ the advective velocity at the particle coordinate, and $\mathbf{v}_d$ is the random velocity fluctuation. The hydrodynamic model determines the advective velocities at discrete times, t^n for $t = 0 \dots n$.

The method used to integrate this equation is an explicit first order Euler method with an approximate trapezoidal interpolation (with prediction/correction) given by:

$$(\text{predictor}) x_i^{n+1*} = x_i^n + tv(x_i^n, t^n),$$

$$(\text{corrector}) x_i^{n+1} = x_i^n + \frac{\Delta t}{2} [v(x_i^n, t^n) + v(x_i^{n+1*}, t^{n+1}) + v_d(x_i^n, t^n) + v_d(x_i^{n+1*}, t^{n+1})].$$

Where Δt is the time step, and n is the time index such that $t^n = n\Delta t$. This method is first order accurate, and particle tracking trajectories may diverge greatly from real ones as time advances unless Δt is very small.

Nevertheless, in highly spatially nonuniform flows, such as those found in oceans, this method still requires a relatively small time step to keep a good accuracy in zones of high speed currents. To overcome this limitation, we propose the use of discrete-event simulation where there is no fixed time step. Instead, the calculation is made for a quantum distance Δq . Here n is an event index corresponding to an iteration with $n = 0, 1, 2, 3, \dots$ being trajectory iteration indices. At each activation, the time advance estimate “ ta ” is computed to be the time taken by the particle to travel the distance Δq . At each particle update, local time at the particle is updated by the simulator to be $t_{next} = t + ta$. The integration method using quantum distance to perform the particle advance is related to the quantized state method (QSS1) [21]. In this application, the integration is performed in two dimensions with a given forcing flow as input.

With the trapezoidal rules, this yields the following formulation:

$$(\text{step predictor}) ta^* = \frac{\Delta q}{v_\alpha(x_i^n, t)},$$

$$(\text{predictor}) x_i^{n+1*} = x_i^n + ta^* [v(x_i^n, t^n)],$$

$$(\text{step estimate}) ta = \frac{2q}{v(x_i^n, t^n) + v(x_i^{n+1*}, t + ta^*)},$$

$$(\text{corrector}) x_i^{n+1} = x_i^n + \frac{ta}{2} [v(x_i^n, t^n) + v(x_i^{n+1*}, t + ta) + v_d(x_i^n, t^n) + v_d(x_i^{n+1*}, t + ta)].$$

14.3.2 DISCRETE-EVENT VERSUS DISCRETE-TIME INTEGRATION

Figure 14.9 shows an example of trajectories simulated using the two different schemes, but with an equal amount of transitions. The discrete time method selected for the test is first order accurate Euler method with a global time step fixed for the duration of the simulation. Compared to the Euler method, the proposed discrete-event trapezoidal method is based on “quantum distance” and thus uses a nonglobal, nonfixed time step.

As can be expected because of the lower order of the Forward Euler approximation compared to the trapezoidal integration scheme, it appears that the discrete time method (Figure 14.9, left) is less accurate in high speed flows, resulting in an exaggerated drift. The discrete-event method accuracy is independent of the current speed, and does not show an exaggerated drift (Figure 14.9, right). In fact, most transitions of the discrete time simulation occurred in very slow current areas. This region is known to be a trap in the Pacific Ocean [25], where floating elements can stay up to 2 years. As such, the inaccuracy is not so much because of the lower order of the Forward Euler approximation compared to the trapezoidal integration scheme, but more that this problem is not well solved by fixed and spatially uniform step integration. With discrete-event time step in this simulation, local ta values ranged between 170 and 3000 seconds with a quantum distance of 1000 meters. The particle position in discrete events has been updated 15,500 times corresponding to a time step of about 1000 seconds for the 6 months of the discrete time of simulation with an equal amount of transitions. Nevertheless, even with this relatively small time step in discrete time, a particle has been pushed out of the current earlier than expected (Figure 14.9, left), resulting in the test simulation lasting only 6 months. This inaccuracy likely results from the less accurate Forward Euler approximation. Once diffusion is added to the model, several thousands of trajectories have to be calculated in parallel in order to represent the possible paths of the particles. While it is possible to implement nonglobal, nonfixed time steps for various other first or

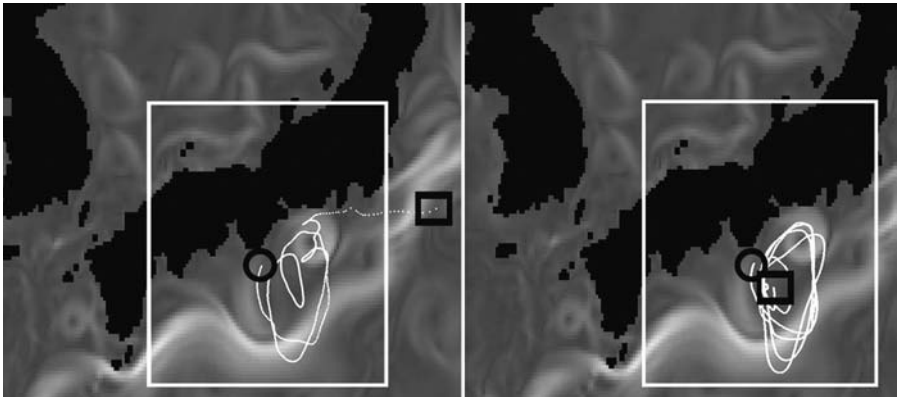


FIGURE 14.9 Six month trajectory simulation without diffusion, (a) Euler method/discrete time, (b) trapezoidal interpolation method/discrete event. Circle represents the start position, square the end position, shading shows the absolute flow speed at the end of the simulation.

higher orders of integration methods in discrete time (Euler or Runge-Kutta family), the proposed method is compliant with the DEVS formalism, while simplifying the computation of the time step with a “quantum distance” analogy. For this reason, the discrete-event method seems to be appropriate to save simulation time in this application.

14.3.3 RANDOM DISPERSION CALCULATIONS

Random velocities are obtained by random sampling in the range of velocities $[-U_r, U_r]$ (zonal flow), $[-V_r, V_r]$ (meridional flow), and $[-W_r, W_r]$ (vertical flow) to simulate spreading of seaweed patches. Those ranges are assumed to be proportional to diffusion coefficients in x , y , and z . The diffusions of velocity have been estimated from a study from Bograd and colleagues [26], which uses satellite tracking of drifters in the northern Pacific to perform empirical estimates of diffusion coefficients. The relative diffusion coefficient is expressed by the following formulae: $D_u = U_r/U(\text{rms})$ and $D_v = V_r/V(\text{rms})$, where the average of D_u was found to be 0.1 and of D_v was found to be 0.08 [26]. In the model, $D_w = W_r/W(\text{rms})$ is given by the mortality and loss of floatability of the seaweed in the floating state. Therefore, D_w is not assumed to have an additional random component, because the mortality distribution in itself is random.

This extensive survey also suggests that the use of an overall diffusion coefficient is a sufficient approximation for modeling studies. The general equation for Euler with dispersion is $V_d = (V_x + V_d)/dt$. To estimate V_d , we then apply the diffusion coefficient to the mean speed obtained from the flow calculation. The random sampling for diffusion coefficients and initial lifetime is computed using the Mesenne Twister algorithm [19].

A Simulation of seaweed dispersion in the East China Sea is presented next.

14.3.4 SIMULATION OF ALGAE DISPERSION

The Laboratory of Behavior, Ecology and Observation Systems at the Ocean Research Institute (University of Tokyo) filed an observation in the East China Sea using R/V Hakuho-Marui. They mapped geographical distributions of drifting seaweeds and collected samples of drifting seaweeds to detect the species. Figure 14.10 shows the distribution of Sargassum in the East China Sea (triangles, squares, and circles) as found by the Laboratory in May 2002. The origin of the Sargassum species that has been newly found as drifting seaweed is thought to be geographically limited to around the Chinese coast in the East China Sea.

The results show that the seaweeds are indeed distributed over the area where the algae has been mapped, moreover it also hints that Sargassum might be found along a path more to the north than the observation, which provides a good hint on where the next observation should be performed, although access to the area is very restricted for non-Chinese ships.

The simulation of drifting seaweeds from the estimated release date in April is presented in Figure 14.10 (bottom left). For this simulation, 100,000 particles are initialized near the shore, with a Δq of 1 km for an area of 3000×2000 km. The

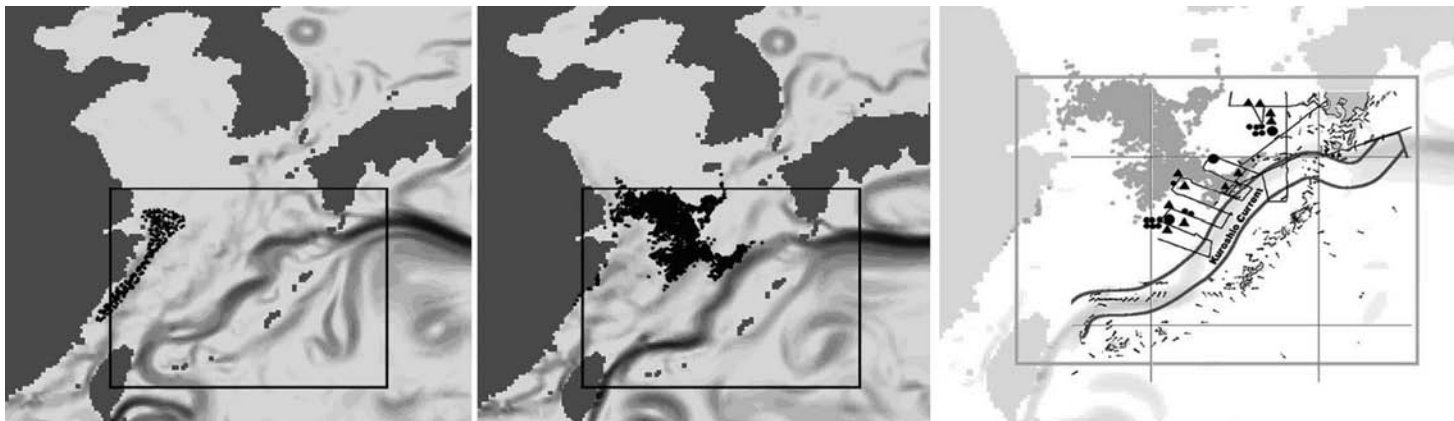


FIGURE 14.10 Initial, April (left) and mid May (center) positions of virtual drifting *Sargassum* rafts around islands off Zhejiang Province, China. Drifting seaweeds observed during the Hakuho Maru KH02-1 leg1 cruise in May 2002 is shown on the right. Drifting seaweeds are depicted as circles (low concentration), squares (average concentration), and triangles (high concentration). The grey thin line represents the ship path.

simulator is implemented as a flat simulator with one global time sorted event list with the most imminent event placed at the head. Each particle advance requires an event to be scheduled. As the fastest moving particle requires more frequent events, these are head-inserted to save the sorting of the full list at each event insertion. For this simulation, about 8% of all particles generated 90% of the sum of all events. Simulation time for this case is about 3 minutes on a 3 Gflops capable Intel dual core processor.

Compared to the fruit flies model, where orchards will always produce the same fruit, ocean flows evolve during the simulation and require a very high spatial resolution. In this application, the advantage of using Lagrangian tracking instead of CA is that it is not necessary to actually map the entire area, but rather focus on the location of the model.

Other applications can also benefit from discrete-event Lagrangian tracking. The same method adapted to fire front simulation is presented next, where the model is focused on the front instead of a burning area.

14.3.5 EFFICIENT ASYNCHRONOUS SIMULATION OF FIRE FRONT EVOLUTION

Most work done in the field of numerical methods of fire spread have used models that prognosticate Rate of Spread (RoS) to reconstruct an evolving fire front. RoS for every portion of the front is used to reconstruct the shape of the fire over time from an initial solution. Forest fire spread is also the subject of numerous DEVS models as covered in a separate chapter of this book [27].

Eulerian and Lagrangian approaches can be used to simulate forest fire spread. With an Eulerian approach, usually formalized in DEVS with cellular models, the environment takes the form of a matrix, much like the fruit flies model presented as a first example of this chapter. The main limitation of this approach is that it does not scale well as resolution becomes a major issue in the case of a large forest fire propagation. A large wildfire can cover an area of several hundreds of square kilometers but a three meter wide road can have a very strong blocking effect on the fire development. A 12 km $\times$ 12 km area at a resolution of 3 meters will require 16 million cells, and in a straightforward approach, just as many DEVS atomic models. A cellular dynamic approach, such as presented in Muzy [10,27], only tracks models that are actually active, thereby greatly reducing the number of components in memory, but it still requires the combustible data to be stored in a large matrix.

A possible way to overcome this limitation is to use a Lagrangian approach to the problem, representing the front and the combustible distribution with polygons (vector format). This approach works by reconstructing the propagation of the front by moving segments of this front.

Among the reconstruction methods, the most commonly used is the method of ellipses, where ellipses of the size of a fire advance for a given time step and then are drawn around an initial front to reconstruct the front one step forward as in the FARSITE simulator [28]. The method of ellipses is not the only one that can be used to reconstruct the very fire front, as one can use level set [29], volume of fluid [30], or markers [31]. This family of methods is usually referred to as front tracking methods.

DEVS adaptation of front tracking is formalized in Vector-DEVS [32] and in adaptation of DS-DEVS [41,42] for evolving shapes.

14.3.6 FRONT TRACKING

Front tracking method [33] is the general denomination for any numerical scheme that integrates the behavior of the very interface between two phases of a system (burned/unburned, water/oil, etc.). In a more general front tracking method, the propagation operator does not have to be an ellipse but it can be any function that can provide a propagation speed for a specific direction of the front, a set of state, and local topology (such as local curvature).

The method proposed in this paper derives from the markers method that proves to be relatively quick and accurate [31]. This front tracking markers method has been transformed into the Vector-DEVS for the purpose of the fire spread model.

In the general method of markers, the interface is discretized by a set of points or markers. At each step, markers are moved according to the speed function in the direction of the normal front at this point (displacement vector) much like particles were advected by the flow in the previous example. The application of the front tracking method to fire front is presented in Figure 14.11.

Front markers are distributed along the back fire line. The propagation direction for each marker is defined to be along the bisector of the angle between the marker, the preceding marker, and the next marker. The actual propagation speed along this direction is deduced from a fire rate of spread model that is used to normalize (provide the propagation speed) the propagation vector.

A marker can be in one of three states, such as shown in Figure 14.12: A Burned state in which it may never propagate again; a Burning state in which it may propagate; and a Dormant state, in which it could propagate in case of a wind shift.

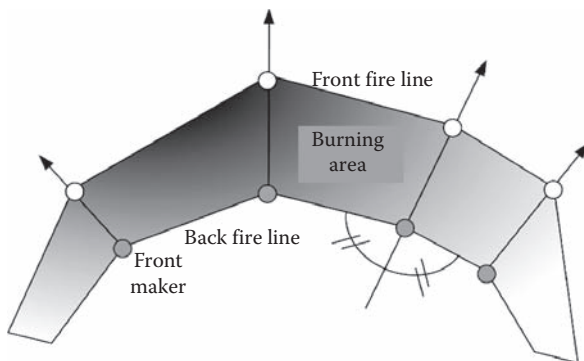


FIGURE 14.11 Front tracking markers method. Grey circles represent markers along the back fire line, lines show the propagation vector on the bisector of the local angle at the marker. White circles are the projected location of the markers after the local burning duration and form the front fire line. Area between the back and the front fire line is the actual burning area.

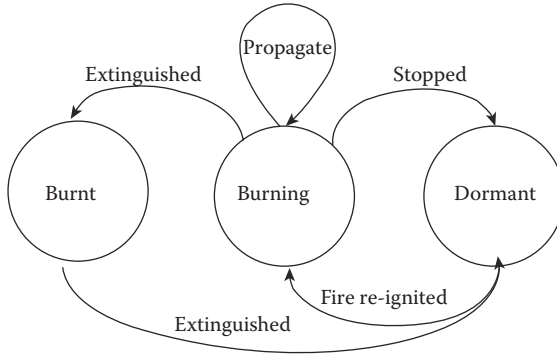


FIGURE 14.12 Fire marker behavior.

Such fire markers (noted M) in the DEVS form have the following structure:

- $X: \{(p, v) | (p\{speed\}, v\{x-speed, y-speed\})\}$
- $Y: \{(p, v) | (p\{position, status\}, v\{x, y\}\{burning, dormant, burned\})\}$
- $S: \{Burning, Dormant, Burned\}, parameters\{x, y, heat, next\ in\ shape, previous\ in\ shape\}$
- $\delta_{int}: S \rightarrow S$ updates position in burning state, and move to the next sequential state if appropriate
- $ta: S \rightarrow R^+$ returns the time to next position update, or next state change
- $\delta_{ext}: Q \times X \rightarrow S$ receives new speed information in case of position or structure update
- $\lambda: Q \times X \rightarrow S$ outputs new position and state

The structure is very similar to the particle model presented in the algae model. The main difference with the algae problem is the dynamic nature of the structure of the front. In addition to the location (p) and propagation vector (v), a front marker has a heat parameter that is computed at each activation to ensure that the marker is active (nondormant) and it has a link to the next and previous markers in the front.

Figure 14.13 presents the couplings. As defined in Vector-DEVS, front markers are contained in a *Shape Manager*, and linked by a network executive that may dynamically add or remove markers. Each set of linked markers corresponds to a fire front, and many fire fronts are linked by an executive in a global *World* coupled model.

Figure 14.14 presents the coupling between the fire model and the experimental frame. As in the seaweed model, the *Generator* is the element that formats atmospheric and fuel data for the front markers. Outputs are linked to the experimental frame *Acceptor* that logs results and ensures that fire is still present and within the simulation area. Polygons of every front are then sent to the *Transducer* for runtime visualization.

Reconstruction of the front fire width is achieved by projecting the future location of the marker along the propagation vector. Nevertheless, the main problem of this method is that collision or intersection must be checked each time a marker is moved to take care of intersections between fronts.

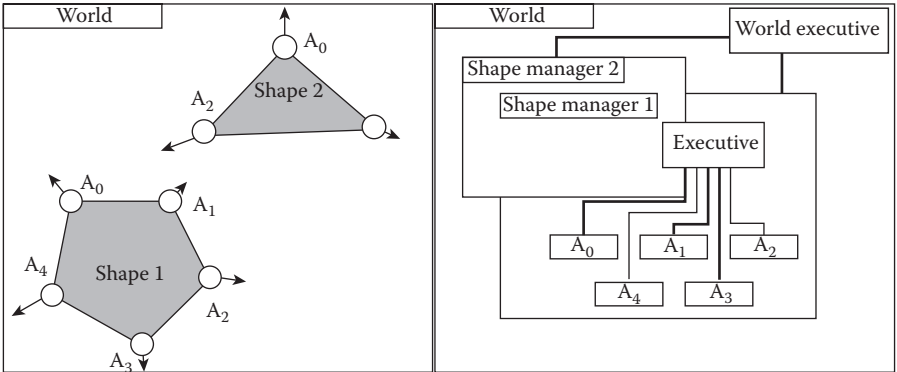


FIGURE 14.13 View of the system (left) and structure (right) of the fire coupling model.

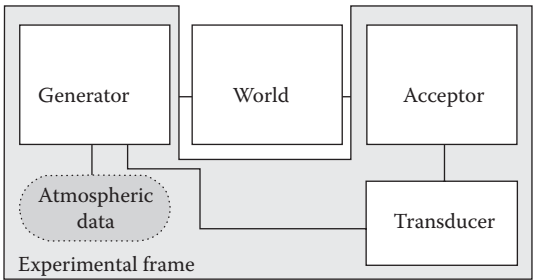


FIGURE 14.14 Coupling between the fire model and the experimental frame.

In time driven simulations, for methods using a spatially uniform and fixed time step, all state changes must be performed synchronously at each time step. Because the time step is a spatially uniform time interval, it is conditioned by the smallest detail that needs to be taken into account in the simulation: The Courant-Friedrichs-Levy (CFL) condition $\Delta t < \Delta x / V$ where V is the fastest speed in the whole domain, and Δx the size of the smallest detail, or the size of a unitary cell in a mesh. In the conventional method of markers, the time step is spatially uniform and constrained by the fastest marker at each step (such as in FARSITE [28]). In forest fires, speed for the same front can range from a few cm/s (back fire) to a few m/s (head fire), in such configurations, markers are recalculated regardless of the actual distance covered. Typically, at each time step the slower markers will be moved by a very small distance (low speed) while the faster markers will be moved by a large distance (high speed).

By using DEVS it is possible to move every marker independently, like a particle in the previous example, calculating the marker advance only by a significant distance, thus removing the constraint of a global CFL condition. As such, the proposed discrete-event integration method may be defined as a spatially nonuniform, nonfixed time step.

Moreover the topological checks for intersection are only performed when markers have traveled this significant distance. The marker advance is integrated by the

same QSS [21] related method as the algae in the previous example, with the speed dependant of the fire velocity function R .

Local slope angle and wind speed for R at the very location of a marker is approximated by bicubic interpolation if the elevation and wind data are in grid format. At the local scale, quantum distance is driving the numerical integration of a marker path. The behavior of the overall front is handled by the shape behavior that is adding or removing points considering the local topological configuration of the marker.

14.3.6.1 Shape Behavior

At each marker advance, a propagation vector is recalculated and a local check on the local front topology is performed to add or delete markers and merge fronts if necessary. Two main processes are changing the shape of the fire front by adding or removing markers:

- Front decomposition (Figure 14.15) markers are added between two other markers if a marker and its neighbor marker are separated by more than $4\Delta q$, ensuring that markers cannot be separated by a distance less than $2\Delta q$.
- Front recomposition (Figure 14.16) markers are deleted if separated by a distance less than Δq to merge fronts and avoid cusps.

At the local scale, quantum distance is driving the numerical integration of a marker path while behavior of the overall front is handled by the reaction to topological change events of decomposition or recomposition. Finally, what drives the marker is

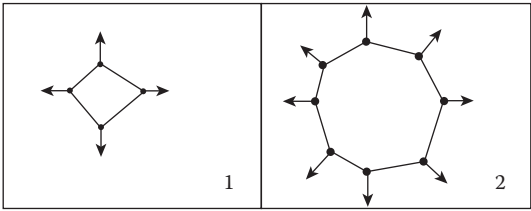


FIGURE 14.15 Decomposition (refinement) after a critical size is reached: (1) before decomposition and (2) the refined shape.

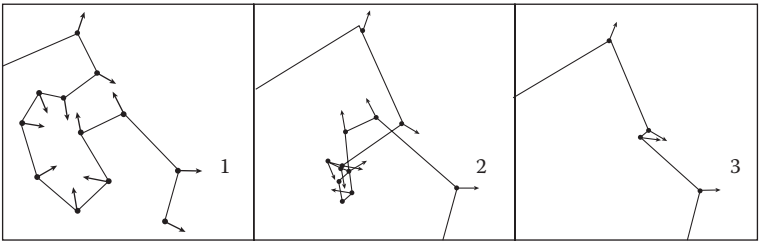


FIGURE 14.16 Recomposition after collision is found: (1) before intersection, (2) intersections detected, and (3) after decomposition.

a fire rate of the spread model, the R function. The selected model for this application has been developed and is available in Balbi [34]. At any given point of space, this model provides the marker's speed given the markers direction.

14.3.7 VALIDATION ON A REAL FIRE ACCIDENT

To verify the robustness and speed of the method in propagating 2-D fire fronts, we ran the model to simulate a fire based on a real scenario. We selected the 2005 Lançon fire that took place in the south of France and burned about 800 Ha of shrubs and forest. On that day, a North Westerly wind of 50 km/h was blowing, providing extreme conditions of propagation. The fire started at 9:40 a.m. and lasted until 4:30 p.m.

This accident is particularly well documented, as a vegetation map is available, as well as roads, ignition points, and three contours of the fire front evolution at noon, 2.30 p.m., and 4:30 p.m.

Nevertheless, it is known that fire-fighters attacked the fire numerous times in order to protect the urban area of “Val de Sibourg” central to the fire and to constrain the fire on its right flank. As there is no precise data available for these attacks, we ran the simulation without taking into account the effect of the fire-fighters.

The wind map has been calculated as a stationary solution using mainstream computational fluid dynamics software, the resolution of the wind map is the same as the digital elevation model: 50 m $\times$ 50 m. Figure 14.17 presents the actual situation of the Lançon area on the day of the fire.

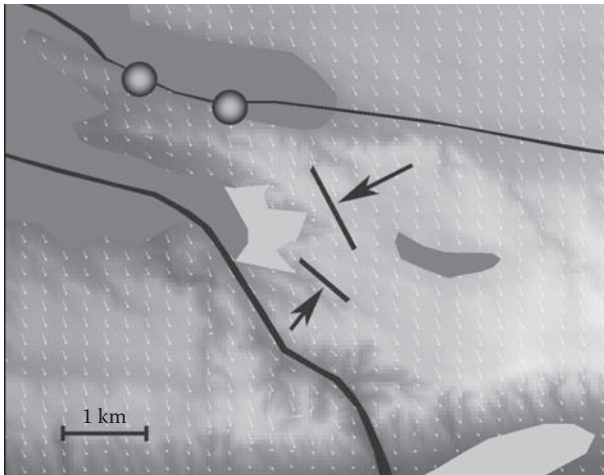


FIGURE 14.17 Map of the Lançon area. Polygons in dark and light grey correspond to crops and urban areas and are considered nonburnable. Shades of grey in the background represent the elevation ranging from 500 m for plain white to 50 m for plain black. Black lines correspond to large roads and big dots to the ignition points that started the fire at 9:40 a.m. The two arrows are pointing to the main estimated areas of fire fighting. Thin white lines show the wind vector as simulated.

14.3.7.1 Simulation Parameters

The simulation has been run with a quantum distance of 10 meters. Because the vegetation is quite homogeneous, the simulation is driven mostly by decomposition (most events are generated because the marker moves over the quantum distance).

Four parameters are required in order to characterize the fuel (vegetation) for the fire propagation model. These parameters are the potential rate of spread, humidity, heat release rate, and flame opacity. This set of optimal parameters has been identified to fit a previous experiment [35] also conducted during the same season and with the same vegetation class.

14.3.7.2 Simulation with Optimal Fuel Parameters

Simulation results and observations are presented in Figures 14.18 through 14.20. The simulation was started at 9:40 a.m. at exactly the same points as the estimated ignition points, and stopped at 4:30 p.m. (last known contour). Although no adjustments were made to the parameters during the simulation run, the simulated contours and times are in good accordance with the observed front. Nevertheless, the simulation results show a strong development on the right front that is not observed in the actual fire, a difference probably because of the attack of the fire fighters that prevented the fire from developing on the right flank.

Simulation time is under 30 seconds on a 3 GFlops capable computer. Considering the high resolution of the simulation (10 meters) the Vector-DEVS method of front tracking simulation proves to be very efficient for the simulation of a real fire in less than a minute.

However, the simulation does not take into account atmospheric effects, such as fire induced winds that can dramatically change wind speed and direction in specific areas such as canyons. The next section presents the same DEVS forest fire model coupled to a conventional mesoscale atmospheric model.

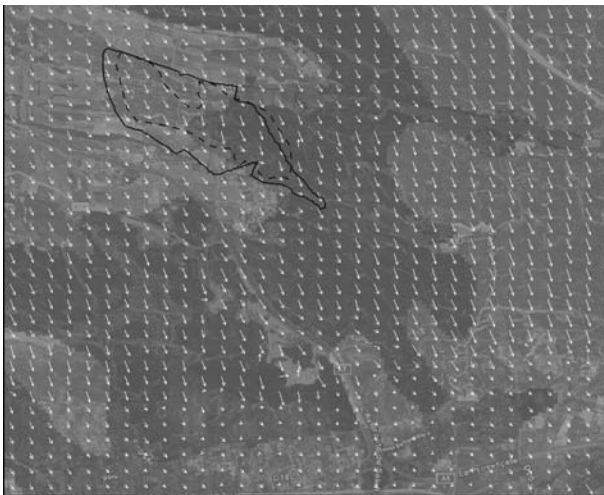


FIGURE 14.18 Simulation results and observations (dashed line) at noon.

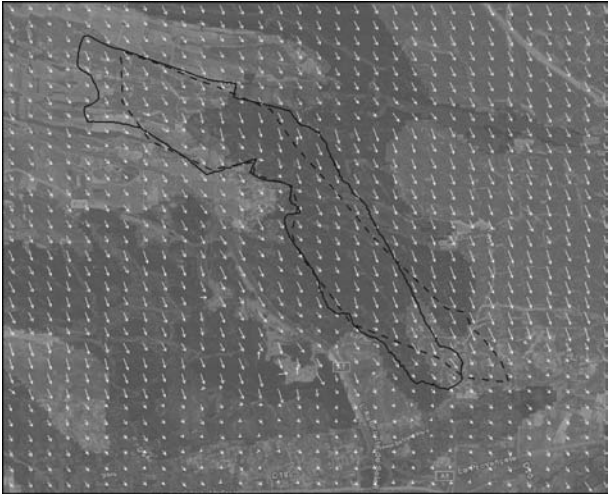


FIGURE 14.19 Simulation results and observations (dashed line) at 2:30 p.m.

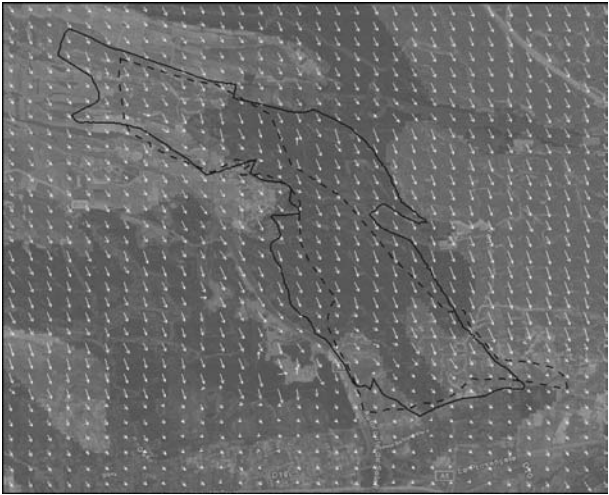


FIGURE 14.20 Simulation results and observations (dashed line) at 4:30 p.m.

14.4 COUPLED FIRE-ATMOSPHERIC MODEL

Wildland fire initiation and spread is known to be heavily influenced by wind as observed during prescribed fires [36] and in simulations [37]. Direction of the spread, propagation speed, and fire plume are the result of strong interactions between the wildfire and the atmosphere occurring at different scales.

This study presents the coupling between the DEVS forest fire spread model and the MesoNH atmospheric model that has been developed to serve research purposes with operational uses in mind. Both models can already be used on the scale of a

large fire while being able to take into account the complex topography, fuel, and fuel break distribution typical of the Mediterranean region. The anelastic nonhydrostatic mesoscale model Mésó-NH [38] is used to integrate the fire induced atmospheric winds that convect the fire front over the surface.

The wild fire propagation model directly uses the wind matrix at ground level in order to interpolate the local wind for each of the front markers. Wind matrices are updated at each atmospheric time step and the wind is assumed to be constant during the entire duration of the step. In order to provide feedback from the fire to the atmosphere, three different surface matrices must be generated from the wildfire model to force the atmospheric model: Heat flux in W m^{-2} , flux of water vapor in kg m^{-2} , and radiative temperature in K. These matrices are generated by the method presented in Figure 14.21.

For each grid cell, the surface of the intersection between the burning area of the front shape and the actual cell area is computed. The ratio of the actual burning area over the cell area is then computed to provide the burning matrix, with values ranging continuously from 0 (no fire) to 1 (all burning). Heat fluxes are approximated by multiplying this matrix with the nominal, vegetation and load dependant, heat flux for every m^2 of vegetation. The water vapor matrix is obtained by multiplying this matrix with the nominal water content per m^2 and the temperature matrix by the average temperature of a wild fire flame. Those matrices are passed to the atmospheric model at every atmospheric time step, just before getting the immediate wind matrix used to convect the fronts.

In order to implement this coupling method in DEVS, a simple coupling encapsulating model has been added to the general couplings as shown in Figure 14.22. The DEVS coupling model is an atomic model that has a time advance function equal to the atmospheric time step. The internal transition method of the submodel that represents the coupling updates the wind matrix of the ground layer and generates the flux files to be utilized by the atmospheric model. These inputs are passed through the experimental frame Generator at the next atmospheric step.

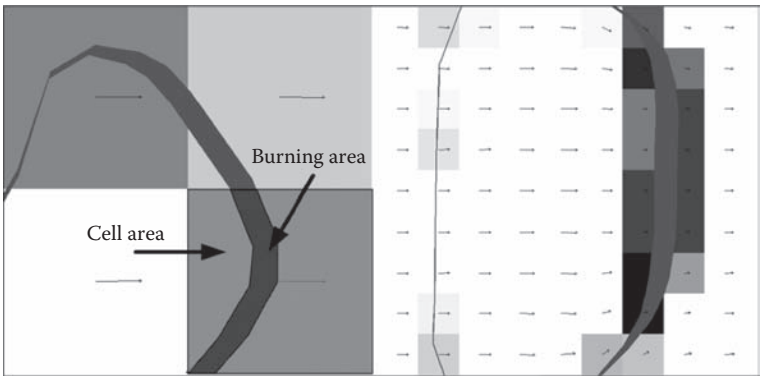


FIGURE 14.21 Integration of burning area is made on every atmospheric grid cell (shades of gray correspond to burning/cell ratio with black as maximum). The grey shape is the actual fire front.

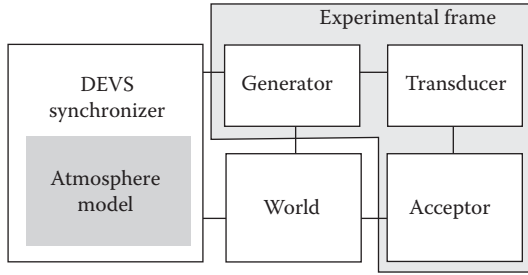


FIGURE 14.22 Coupling of the fire/atmosphere coupled model.

14.4.1 EXPERIMENTAL SET-UP AND SIMULATION RESULTS

Numerical simulations have been carried out for a typical configuration of a fire spread. The domain was $1.2 \text{ km} \times 1.2 \text{ km}$ in size with a height of 1.2 km , covered by a grid of $40(x) \times 40(y) \times 40(z)$ cells.

The ground layer represents a constant uphill slope ranging from 0 m to 120 m in height. The atmosphere was forced with a constant westerly low wind of 3 m/s constant with height. Diagnostics for the plume were performed by injecting a scalar tracer with a concentration and distribution equal to the burning ratio of each grid point and for each atmospheric time step.

Simulations are run with and without coupling. In the case of no coupling, no fluxes or radiative temperature fields are forcing the atmospheric model. In the case of coupling, the nominal heat flux is 100 kw/m^2 , water vapor flux of 0.1 kg/m^2 , a radiative temperature of 800 K , quantum distance of 2 m , and a burning time of 30 s .

Figure 14.23 (left) presents the simulation of an extreme line ignition after only 180 seconds of simulations. Compared to the previous experiments, the local wind velocity is increased to 6.5 m/s very close to the fire front, increasing the head fire rate of spread from 0.2 (without coupling) to 0.5 m/s (with coupling). The wind at ground level is concentrated in a zone of confluence just ahead of the fire head with its maximum immediately ahead of the fire.

Considering the much larger injection area, the convective effect is stronger here. The vertical section shown in Figure 14.23 (right) presents a well defined plume with a maximum wind velocity of 24 m/s in the plume, and an area of maximum tracer concentration as high as 300 m , suggesting that the smoke is advected directly in the higher atmosphere by strong convection. Significant concentration of tracer here reaches an altitude of 850 m . Although it is not the subject of this study to verify the results quantitatively, the values are well in accordance with estimated values observed in large experimental fires such as the International Crown fire experiment [39,40].

The simulation was effected on a cluster of two 4 GFlops capable computers. The calculation time for the run is about 20 minutes for 180 seconds of simulations. The calculation of the fire front advance is achieved by a flat simulator and accounts for only 5 seconds of computer time. The coupling method requires 50 seconds of mainly input/output operations for the entire duration of the run. While still performing fast on a relatively modest computer, the scale of the simulation (about a square kilometer

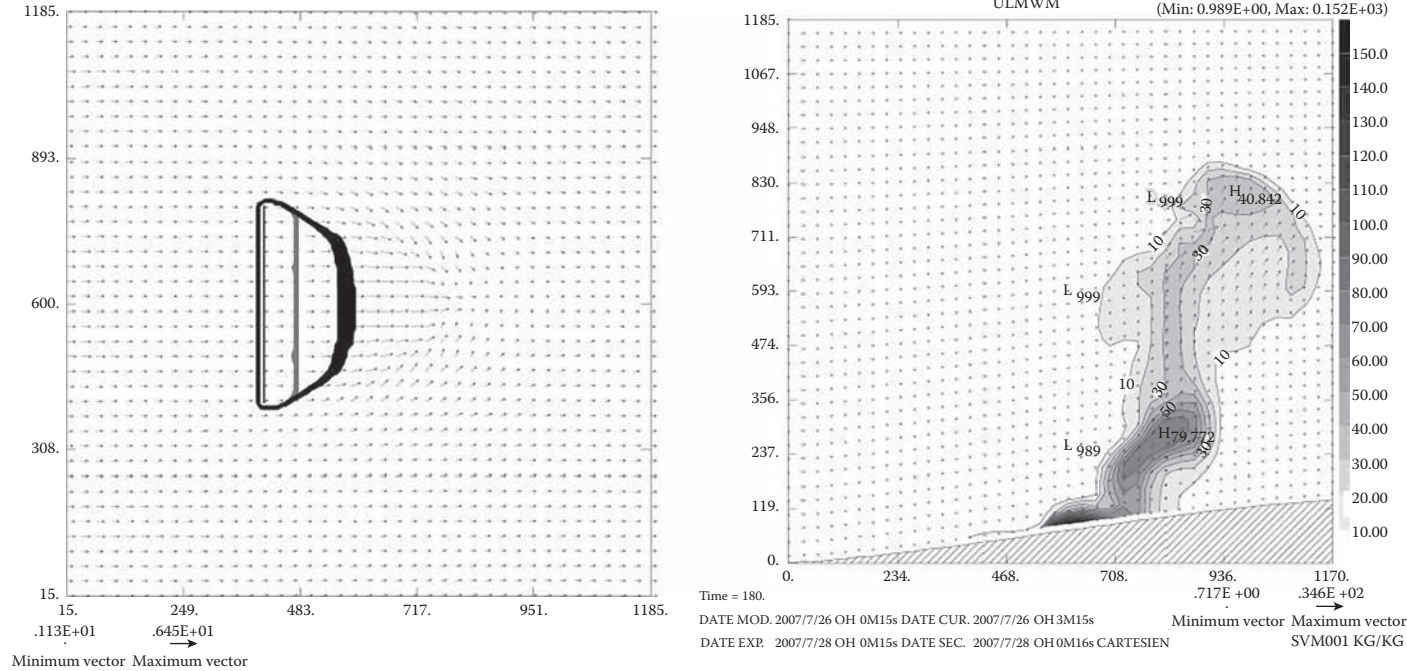


FIGURE 14.23 Fire lines (left) after 180 seconds of simulation for the coupled (black) and noncoupled (grey) propagation. Arrows denote the horizontal 10 meters height wind vectors of the coupled case. The thin dark line represents the initial ignition point for both cases. The right image corresponds to the vertical cross section of the coupled case at $y = 600$ m, shading represents concentration of the injected passive tracer and arrows represent wind vectors.

for about an hour) is smaller than a large wildfire (hundreds of square kilometers for about a day) that still require the use of a supercomputer.

14.5 CONCLUSION

This chapter has presented three different applications of DEVS for the modeling of environmental systems. The choice has been made not to focus on actual implementation but on the overall presentation of the models, showing the benefits added by the DEVS formalism to the different approaches used in these simulations. Depending on what is to be observed, the actual model can be very different, concentrating on a cell, a particle, or a front, but nevertheless all are considered as DEVS automaton, with inputs, outputs, update, and a time advance function that handles the asynchronous nature of the systems and the time granularity, naturally.

In the first application, a cellular Mediterranean fruit fly model was developed to estimate the concentration of insects in mature fruits. Applying DEVS on this problem made the modeling task easier as it helped to define the asynchronous nature of the problem with a well defined structure.

In the second application, DEVS is used to define models of algae advected by a prescribed flow. Defining the model as DEVS helped in this case to cope directly with the large diversity of current speed in the sea, as the position in the DEVS models is not integrated over time, but for specific events, defined here as the time to travel a given distance: the quantum distance.

The last application extended the particle model by linking particles together to form a front. The method helped to focus the computation directly on the very front line of a fire, providing simulation times under a minute for a large fire that ran for a whole day with a resolution of 10 meters. To illustrate that it is possible to couple DEVS models with discrete time simulations, the fire spread simulator was then coupled with a mesoscale atmospheric model, enabling it to simulate fire plume and fire induced winds.

In all these simulations, the use of DEVS or DEVS variants proved to offer a well defined framework that removed the constraints on the way models are conceived and offers a framework within which models are built.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the assistance of Colonel Claude Picard and Frédérique Giroud of CEREN who gave us access to the fire data under the CNRS GDR “Incendie” program. Work on the forest fire propagation model has been supported by the CNRS grant PEPS 07_36.

REFERENCES

1. Jorgensen, S. E., and Bendoricchio, G. *Fundamentals of Ecological Modelling*. Oxford: Elsevier, 1994.
2. Hill, D., and Coquillard, P. Ecological modelling and simulation, Chapter 29 in *Handbook of Dynamic System Modeling*, 29.1–29.18. Edited by P. Fishwick. Boca Raton, FL: CRC Press, 2007.

3. Ford, A. *Modeling the Environment: An Introduction to System Dynamics Models of Environmental Systems*. Washington, DC: Island Press, 1999.
4. White, I. D., Mottershead, D. N., and Harrison, S. J. *Environmental Systems: An Introductory Text*, 2nd ed. Boca Raton, FL: Routledge, 1992.
5. <http://www.iso.org>
6. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*. 2nd ed. New York: Academic Press, 2000.
7. Toffoli, T., and Margolus, N. *Cellular Automata Machines*. Cambridge, MA: MIT Press, 1987.
8. Wolfram, S. *A New Kind of Science*. Champaign, IL: Wolfram Media, 2002.
9. Ntaimo, L., and Zeigler, B. P. Forest fire spread and suppression in DEVS, *Simulation* 80 (2004): 479–500.
10. Muzy, A., Wainer, G., Innocenti, E., Aiello, A., and Santucci J. F. Comparing simulation methods for fire spreading across a fuel bed. *Proceedings of AIS'2002 Conference*, Vol. 1, 219–24, 2002.
11. Wainer, G. CD ++ : A toolkit to develop DEVS models. *Software—Practice & Experience* 32, no. 13 (2002): 1261–1306.
12. Wainer, G. Applying cell-DEVS methodology for modeling the environment, *Simulation* 82, no. 10 (2006): 635–60.
13. Bernardi, F., and Filippi, J. B. A generic framework for environmental modeling and simulation. *Proceedings of the IEEE SMC 2003 Conference*, 1810–15. New York: IEEE, 2003.
14. Vera, T., Rodriguez, R., Segura, D. F., Cladera, J. L., and Sutherst, R. W. Potential geographical distribution of the mediterranean fruit fly, *Ceratitidis capitata* (Diptera: Tephritidae), with emphasis on Argentina and Australia, *Environmental Entomology* 31, no. 6 (2002): 1009–32.
15. Filippi, J. B., and Bisgambiglia, P. The JDEVS environmental modeling and simulation environment, *Environmental Modelling & Software* 19, no. 3 (2004): 261–74.
16. <http://www.oec.fr/>
17. Reynolds, A. On the nonextensive statistical mechanics of tracer-particle motions in turbulence, *Physical Review Letters* 91 (2003).
18. García-Martínez, R., and Flores-Tovar, H. Computer modeling of oil spill trajectories with a high accuracy method, *Spill Science and Technology Bulletin* 5, no. 5–6 (1999): 323–30.
19. Matsumoto, M., and Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, no. 1 (1998): 3–30.
20. Filippi, J. B., Komatsu, T., and Tanaka, K. Computer modeling and discrete event simulation of seaweeds trajectories. *Proceedings of OICIMS Conference* 119–27, 2005.
21. Kofman, E. Quantization-based simulation of differential algebraic equation systems, *Simulation* 79, no. 7 (2003): 363–76.
22. Komatsu, T., Tatsukawa, K., Filippi, J. B., Sagawa, T., Matsunaga, D., Mikami, A., Ishida, K., Ajisaka, T., Tanaka, K., and Aoki, M. Distribution of drifting seaweeds in eastern East China Sea, *Journal of Marine Systems* 67, no. 3–4 (2007): 245–252.
23. See Chapter Nr 6 “Effective and Efficient Modeling and Simulation with DEVS Variants.”
24. Masumoto, Y., Sasaki, H., Kagimoto, T., Komori, N., Ishida, A., Sasai, Y., Miyama, T., et al. A fifty-year eddy-resolving simulation of the world ocean—Preliminary outcomes of OFES (OGCM for the Earth Simulator), *Journal of the Earth Simulator* 1 (2004): 35–56.
25. Hurlburt, H., Wallcraft, A., Schmitz, W., Hogan, P., and Metzger, E. Dynamics of the kuroshio/oyashio current system using eddy-resolving models of the north pacific ocean, *Journal of Geophysical Research* 101, no. 10 (1996): 941–76.

26. Bograd, S., Thomson, R., Rabinovich, A., and LeBlond, P. Near-surface circulation of the northeast pacific ocean derived from woc-svp satellite-tracked drifters, *Deep-Sea Research Part II* 46, nos. 11–12 (1999): 2371–2403.
27. See Chapter Nr 13 “Stochastic Modeling Strategies for the Simulation of Large (Spatial) Distributed Systems: Application to Fire Spread.”
28. Finney, M., and Andrews, P. The FARSITE fire area simulator: Fire management applications and lessons of summer 1994. *Proceedings of Interior West Fire Council Meeting and Program*, 209–16, 1994.
29. Mallet, V., Keyes, D. E., and Fendell, F. E., 2009: Modeling Wildland Fire Propagation with Level Set Methods, *Computers & Mathematics with Applications* 57, no. 7 (2009): 1089–1101.
30. Hirt, C. W., and Nichols, B. D. Volume of fluid (VOF) method for the dynamics of free boundaries, *Journal of Computational Physics* 39, no. 1 (1981): 201–25.
31. Lallemand, P., Luob, L., and Pengb, Y. A lattice Boltzmann front-tracking method for interface dynamics with surface tension in two dimensions, *Journal of Computational Physics* 226, no. 2 (2007): 1367–84.
32. Filippi, J. B., and Bisgambiglia, P. Enabling large scale and high definition simulation of natural systems with vector models and JDEVS. *Proceedings of the IEEE/SMC, IEEE/CS 2002 Winter Simulation Conference* (2), 1964–1970, 2002.
33. Risebro, N., and Tveito, A. A front tracking method for conservation laws in one dimension, *Journal of Computational Physics* 101 (1992): 130–39.
34. Balbi, J. H., Rossi, J. L., Marcelli, T., and Santoni, P. A. A 3D physical real-time model of surface fires across fuel beds, *Combustion Science and Technology* 179 (2007): 2511–37.
35. Santoni, P. A., Simeoni, A., Rossi, J. L., Bosseur, F., Morandini, F., Silvani, X., Balbi, J. H., Cancellieri, D., and Rossi, L. Instrumentation of wildland fire: Characterisation of a fire spreading through a Mediterranean shrub, *Fire Safety Journal* 41, no. 3 (2006): 171–84.
36. Clements, C. B., Potter, B. E., and Zhong, S. In situ measurements of water vapor, heat and CO₂ fluxes within a prescribed grass fire, *International Journal of Wildland Fire* 15, no. 3 (2006): 299–306.
37. Clark, T. L., Coen, J., and Latham, D. Description of a coupled atmosphere-fire model, *International Journal of Wildland Fire* 13 (2004): 49–63.
38. Lafore, J. P., Stein, J., Asencio, N., Bougeault, P., Ducrocq, V., Duron, J., Fischer, C., et al. The Meso-NH atmospheric simulation system. Part I: Adiabatic formulation and control simulations, *Annales Geophysicae* 16 (1998): 90–109.
39. Putnam, T., and Butler, B. W. Evaluating fire shelter performance in experimental crown fires: The International Crown Fire Modelling Experiment (ICFME), *Canadian Journal of Forest Research* 34, no. 8 (2004): 1600–1615.
40. Stocks, B. J., Alexander, M. E., Wotton, B. M., Stefinger, C. N., Flannigan, M. D., Taylor, S. W., Lavoie N., et al. Crown fire behaviour in a northern jack pine-black spruce forest, *Canadian Journal of Forest Research* 34 (2004): 1548–60.
41. Barros, F. J. Semantics of dynamic structure event-based systems. *Proceedings of the Second International Conference on Distributed Event-Based System*, 245–52, 2008.
42. Barros, F. J. Dynamic structure multiparadigm modeling and simulation, *ACM Transactions on Modeling and Computer Simulation* 13, no. 3 (2003): 259–75.

Section IV

Standardization

15 DEVS Standardization: Foundations and Trends

Gabriel A. Wainer, Khaldoon Al-Zoubi, Olivier Dalle, David R.C. Hill, Saurabh Mittal, José L. Risco Martín, Hessam Sarjoughian, Luc Touraille, Mamadou K. Traoré, and Bernard P. Zeigler

Since the early 1970s, the modeling and simulation (M&S) community has been attempting to formulate approaches to modeling as system specification formalisms. In many cases, different models (i.e., mathematical representations) of the source systems existed before their computerized incarnations. Examples of this include, for instance, differential equations (which have continuous states and continuous time), models of systems operating on a discrete time base, and so on. As discussed in other chapters of this book, the Discrete Event System Specification (DEVS) formalism was defined to bring coherence to the field of discrete-event M&S by using an organized system-theoretical framework. In spite of this coherence and the unity that this theory introduces into the field of M&S, today, almost 40 years later, the Discrete Event System Specifications are still held hostage by their simulation language implementations and algorithmic code expressions.

We believe that the need for a widely accepted framework is now more necessary than ever because of the fragmentation in the field that has resulted from the growing specialization of knowledge. The need for “knowledge workers” who can synthesize disciplinary fragments into cohesive wholes is increasingly recognized. M&S—as a generic, nondiscipline specific, set of activities—can provide a framework of concepts and tools for such knowledge work.

The following chapters present different approaches to solve these problems. The presented approaches each provide a solid foundation for standardization and by contrasting their specific strengths and less developed qualities, the significant progress in the area is shown. These approaches build on new methods and the underlying technologies have made model building and simulation execution easier and faster by riding piggyback on the technology advances in software and hardware. More recently, fundamental issues such as model credibility (e.g., validation, verification, and model family consistency) and interoperation (e.g., repositories, reuse of components, and resolution matching) have moved to the front and center under the impetus of the High Level Architecture (HLA) standard. The HLA focused on interoperability of existing geographically dispersed M&S assets, and it was initially built focusing on defense applications (which still are the main application focus of the standard).

These days, however, other users are interested in distributed simulation applications as well, including those in business enterprises and e-commerce (which are becoming important as complexity increases and lead-times diminish). The HLA does not address this kind of application easily. In addition, it does not focus on how to solve the problem of creating models to be executed in the simulation environment.

Instead, the DEVS theoretical foundation makes it (in principle) independent of the various programming languages and hardware platforms. Clear examples of this have been seen in previous chapters, where we could see the wide variety of groups working on extensions to the DEVS formalism (each of them focusing on real-time modeling, cellular model definition, dynamic structure applications, etc.) and several modeling tools based on these extensions. Nevertheless, there still is a need to standardize these notations and simulation tools. Within a Study Group of the Simulation Interoperability Standards Organization (SISO), a standard has been under development to support interoperability of DEVS models implemented in different platforms as well as with legacy simulations. The goal of the study group is to find a core of the DEVS formalism that is suitable for standardization of activities at the level of modeling, thereby bridging the gap between existing simulation frameworks and modeling activities using a standard notation. In this part, we will discuss fundamentals of various designs on how to achieve these goals through the standardization of DEVS models and their simulators.

One of the main issues to be addressed is a direct consequence of separating the model from the simulator: this results in multiple ways in which the same model can be simulated—all adhering to the abstract simulator specification. For instance, there are *virtual-time* simulators (where the simulator can skip from one event time to the next without traversing the intervening time interval) and *real-time* simulators (where time is interpreted as wall clock readings, so the simulator must wait for the interval to its next scheduled event to expire before handling the event). In addition to the different combinations of model type/simulation software, a standard allows for the use of different forms of distribution of model components (e.g., single processor vs. multiprocessor, and within the latter, conservative vs. optimistic time advance for virtual-time as well as centralized vs. noncentralized time control in real-time execution). The standard can also be independent of different implementation platforms, such as Windows versus UNIX; different programming languages, such as Java versus C++; and different networking and middleware frameworks such as .Net versus Apache. As we can see, such a standard can have multiple simulation scenarios. For example, a model may be simulated in virtual-time and in real-time both in a distributed and a nondistributed fashion.

The remaining three chapters of this book discuss in detail the various recent standardization approaches in DEVS simulation tools. In most cases, Web Services technology is highly leveraged, which has proven useful in achieving model and simulation interoperability. Developments in the World Wide Web have prompted many efforts in the distributed simulation field for modeling, executing simulations, and creating model libraries that can be assembled and executed over HTTP. By means of XML and web services technology these efforts have entered into a new phase with exponentially increasing utility as new standards are becoming available.

ACKNOWLEDGMENTS

This section of the book has been built based on the effort of numerous other researchers (a noncomprehensive list can be found in the DEVS Standardization Study Group webpage, <http://cell-devs.sce.carleton.ca/devsgroup/>). In particular, Professor Tag-Gon Kim (KAIST, Korea) has sponsored this group, with the goal of demonstrating the feasibility of a DEVS Simulator standard. Numerous other members of the Study Group have participated in the standardization efforts, including Fernando Barros, Claudia Frydman, Norbert Giambiasi, Jan Himmelspace, Xiaolin Hu, Moon H. Hwang, Jim Nutaro, Lin Uhrmacher, and Hans Vangheluwe. Pieter J. Mosterman contributed with the numerous editions and organization of these chapters.

16 An Introduction to DEVS Standardization

*Gabriel A. Wainer, Khaldoon Al-Zoubi,
David R.C. Hill, Saurabh Mittal, José L. Risco
Martín, Hessam Sarjoughian, Luc Touraille,
Mamadou K. Traoré, and Bernard P. Zeigler*

CONTENTS

16.1	Introduction	393
16.2	Background	399
16.2.1	The DEVS Formalism	399
16.2.2	DEVS Simulation Protocol	400
16.2.3	Extensible Markup Language (XML).....	402
16.2.3.1	Document Type Definition (DTD).....	403
16.2.3.2	XML Schema	403
16.2.4	Distributed Simulation Interoperability Middleware	404
16.2.4.1	Common Object Request Broker Architecture (CORBA).....	405
16.2.4.2	SOAP-Based Web-Services	405
16.2.4.3	REST-Based Web-Services.....	408
16.3	Designing Interoperability of DEVS Components	411
16.3.1	Model-Based Interoperability	411
16.3.2	Simulator-Based Interoperability	412
16.4	Comparing Existing Designs and Implementations	414
16.4.1	Standardizing DEVS Model Representation	414
16.4.2	Standardizing Interoperability Middleware	418
16.5	Summary	422
	References.....	423

16.1 INTRODUCTION

Since the early 1970s, the M&S (modeling and simulation) community has been trying to formulate approaches to modeling as system specification formalisms. As seen in the previous chapters of this book, the proliferation of DEVS-based M&S engines has brought the need to improve and standardize DEVS tools, facilitating the work of DEVS designers independently of the programming language implementations

or algorithmic code expressions used. To understand the problem, let us consider that DEVS categorically separates the Model, the Simulator, and the Experimental frame. Building on this separation of concerns, the DEVS Protocol specifies the abstract simulation engine that correctly simulates DEVS atomic and coupled models. Interpreted in a distributed simulation context, the DEVS abstract simulator gives rise to a general protocol that has specific mechanisms for declaring who takes part in the simulation (i.e., the federates in the case of an HLA federation). It also specifies how federates interact in an iterative cycle that controls how time advances, when federates exchange messages, and perform internal state updating. A significant feature in comparison to simulation based on the HLA standard, is that if the federates in simulation are DEVS compliant then the simulation can be proved to be correct in the sense that the DEVS closure under coupling theorem guarantees a well-defined resulting structure and behavior.

DEVS modeling and simulation research groups are interested in DEVS interoperability in order to enhance model composability and reusability of DEVS models and non-DEVS models in different languages and platforms. The problem to interoperate heterogeneous DEVS models with DEVS simulators is that DEVS simulators implement the DEVS modeling formalism in a growing number of diverse programming environments. Although the DEVS formalism specifies the same abstract simulator algorithm for any simulator, different simulators implement the same abstract simulator using different programming languages and environments.

In other words, the model and the simulator exist in the same programming language. Consequently, legacy models as well as models that are available in one implementation are difficult to translate from one language to another (even when both the implementations are object oriented). Other constraints such as libraries inherent to the programming language (i.e., C++ or Java) are a source of complexity that prevents the desired interoperability. This is particularly important in large scale M&S, including M&S of Systems of Systems. In order to support such interoperability, various standards have been developed.

The new computing methods of recent years (in particular, Grid computing and Cloud computing) introduced new ways of sharing computing power and storage in heterogeneous environments. Using these technologies, resources are virtualized as services consumed on demand (with minimal limitation for resource location). By exposing the unused resources available in most organizations, simulation performance can be improved through replication processes. This model of computation has been extremely successful, but it is still limited for more advanced simulations (for instance, when online interactivity is needed, for interfacing a simulation with sensor networks, etc.). Likewise, simulation interoperability on the Grid can require complex ad hoc tailoring, which is a costly and elaborate process.

Parallel simulation middleware (e.g., GATech Time Warp [1], Warped [2], SPEEDES and WarpIV [3], etc.) had usually focused on tightly coupled systems. Instead, distributed simulation middleware must allow partitioning and running simulations remotely. For instance, DIS (Distributed Interactive Simulation [4]) allowed the development of distributed simulation-based training solutions (by sharing data and computing power remotely). Other solutions included the HLA

(High Level Architecture [5]), which was designed for interoperability of distributed simulation assets, and TENA (Test and Training Enabling Architecture [6]), which was built on top of CORBA (and subject to its strengths and limitations, see below) to enable real-time interoperation of assets in geographically distributed test ranges. This distributed simulation middleware focuses on data sharing, distributed processes, communication, and time management (in HLA), and has facilitated the development of large-scale distributed simulations. Nevertheless, model reuse using this kind of middleware is still difficult, ad hoc, and costly. The motivation for the discussion in this chapter stems from this need of model interoperability between the disparate simulator implementations; we intend to discuss the means to provide simulators that are transparent to model execution. We claim that a DEVS-based standard would improve sharing and interoperability of M&S assets, both locally and distributed, including specifications to facilitate a System of Systems that interact using a net-centric platform. At this level, model interoperability is one of the major concerns.

As discussed earlier in this book, there are now numerous libraries and tools for expressing DEVS models across the globe, such as DEVSJAVA [7], DEVS/C++ [7], CD++ [8], ADEVS [9], DEVS-Suite [10], James [11], and so on. (For a comprehensive list, check <http://cell-devs.sce.carleton.ca/devsgroup/?q=node/>.) Although this proliferation of libraries is witness to the numerous advantages in DEVS M&S, its multiplicity not only complicates sharing of models, it also requires modelers to learn the specific programming language in which the simulator is implemented, and also the rules for integrating a model with that simulator. An important side effect of the latter is that most of the times the modelers then become locked to this language.

The different designs to be discussed in this chapter show efforts to bridge the gap between the different implementations by employing standard means (i.e., XML) in order to share model information and make a step toward model interoperability and reuse.

The different designs discussed in this and the following two chapters deal with two different interoperability objectives:

1. *Standardizing DEVS model representation:* The idea is to allow a platform-independent DEVS model representation to be executed by a DEVS-based simulator. In this case, a DEVS model is executed on a single processor or on a parallel/distributed simulation environment. This allows model reusability without the need of performing long-distance distributed simulation. Typically, models are stored in repositories and retrieved as needed, as shown in Figure 16.1. In this figure, the Model boxes correspond to DEVS specifications independent of the application domain. These specifications must be independent of computer programming languages, for instance in set theory or some kind of calculus. From such formal specifications, software specifications—such as XML—can be derived. The boxes called Simulator represent the protocols that are needed to execute the specified models. By means of XML, the modeler may use a standard representation of a DEVS scenario that does not require much programming knowledge, making automatic transformations from the XML representation to

a particular simulator representation. The implementations shown in this chapter are based on XML (i.e., Document Type Definition [DTD] and later on XML Schema, usually called XML Schema Definition [XSD]).

- 2. *Standardizing Interoperability Middleware*: The idea is to allow for interfacing different simulation environments, providing synchronization for the same simulation run across a distributed network regardless of their model representation, as shown in Figure 16.2.

The development of this middleware is the area of most interest to overcome current distributed simulation challenges and to meet future expectations, as indicated by a number of surveys [13,19]. Developing a standardized DEVS simulation protocol would allow enabling different DEVS implementations to simulate the same DEVS model hierarchy partitioned between various DEVS domains, while allowing all

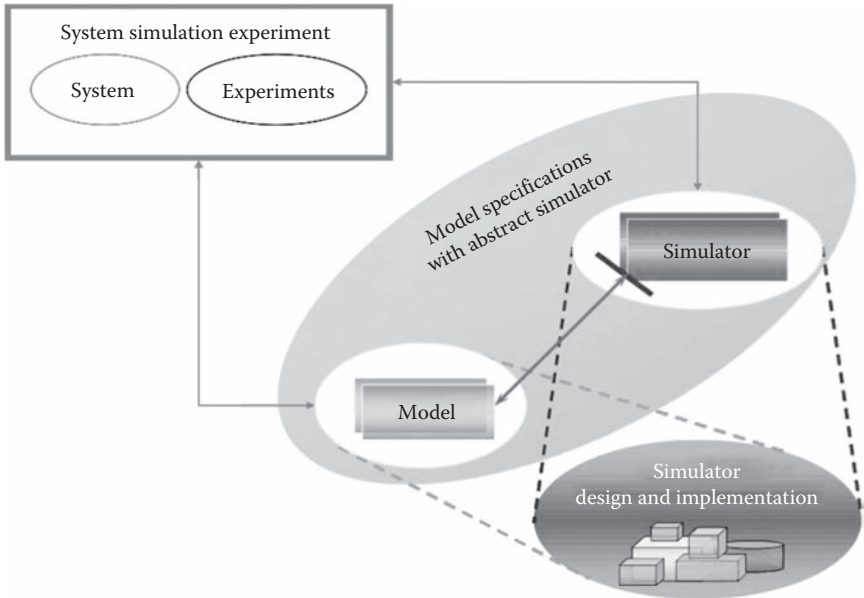


FIGURE 16.1 Concept of standardized independent-platform model representation.

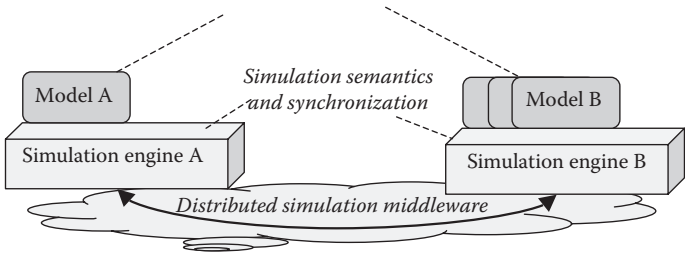


FIGURE 16.2 Concept of standardized distributed simulation middleware.

DEVS domains to execute their legacy models, performing distributed simulation experiments between different heterogeneous simulation models and engines. All middleware designs presented in this chapter offer simulation resources as consumable services. In this case, simulation entities act as peers (i.e., clients and servers at the same time) to each other to synchronize a simulation session.

Each of these two objectives form two different set of standards and should be treated separately. In other words, a simulation tool does not need to conform to both sets of standards, stated above, to be able to reuse other resources from different working groups.

DEVS-to-DEVS Interoperability is a basic form of interoperability that is enabled by a DEVS standard. Adoption of the DEVS standard would facilitate new development to achieve interoperability at the syntactic, semantic, and pragmatic levels. This is accomplished at the two levels stated above: model representation and middleware synchronization.

DEVS-to-Non-DEVS Interoperability is more complex. We need to establish mechanisms to implement the core simulator interface, or the Abstract Model interface to interoperate at the syntactic level with DEVS and other non-DEVS peers. In its strongest form, such simulation methodology guarantees well-defined time preservation and simulation correctness as a sound basis to aim for interoperability at the higher levels. This part relies more heavily on the middleware standard set. In this case, the middleware needs to hide as much as possible internal detailed implementation, including the DEVS formalism representation, in order to allow practical DEVS-to-non-DEVS simulation synchronization.

Each of the above objectives faces different challenges. The main challenge of standardizing a DEVS model representation is in integrating the standardized representation into local simulation environments. In other words, the target simulation platform must interpret the standardized representation in a manner similar to its local model representation. For example, some DEVS tools define their models as programming language structures (e.g., Java/C++ classes). In this case, the problem extends beyond the applied programming language and also involves the complications of plugging the resulted programming structure into the overall hierarchy design of the DEVS tool. Afterward, those models need to be compiled with DEVS tools legacy source code. In other cases, some DEVS tools define their coupled representation in tool-dependent textual format. The only issue here is in converting the standardized representation into the required textual format. Standardization independent of domain knowledge can be formulated in terms of handling different syntactical specifications and programming languages while handling alternative simulator designs, and simulator features. Syntactical interoperability can be defined in terms of the generality of the DEVS model specification and its underlying abstract simulator algorithm, both of which are independent of programming languages. Different design choices can support the same model syntax and abstract simulator semantics. In particular, the engine that performs a simulation includes features (e.g., visual component-based notation, display of simulation data trajectories, and configuration of simulation experiments) that must not have any side effect with respect to correctness of the simulator protocol [15]. On the one hand, the

execution speed (performance) of the simulator may be adversely affected and on the other hand, users can benefit from important features such as simulation animation, viewing time-based trajectories and run-time design of experiments [10].

On the other hand, standardizing a middleware protocol to execute distributed simulation between heterogeneous simulation entities faces different challenges. In this case, a programming language or other local implementation issues are not a problem, since the standardized protocol should hide and overcome these issues, as in the case of any Grid/Cloud computing system. However, the main problem is to determine how the different environments synchronize a simulation session. This includes a number of issues such as the simulation semantics (e.g., messages versus programming parameters), synchronization protocol (i.e., how information is sent and received), and so on. These issues, also shown in Figure 16.2, are important because they affect the flexibility of improving simulation algorithms, the required amount of software changes of DEVS tools, and the ability of a DEVS tool to evolve independently with other unrelated features to the standards. In fact, current DEVS tools support distributed simulation using different underlying technology, with their own defined synchronization protocol to manage a simulation between different distributed partitions.

Interoperability among different simulation engines is generally considered domain-neutral. Execution of models implemented in two or more different programming languages can be guaranteed to be DEVS-compliant for a unique abstract simulation algorithm. The specifics of a system and the conditions under which it is to be experimented with can be modeled and simulated in terms of system and experimental frame models. The simulation engine is domain-neutral and the domain-specific modeling constructs are built from generic DEVS modeling constructs. It is desirable for domain-specific models to be independent of the simulation engines that can execute them.

DEVS implementations have evolved independently over the years, which make interoperating them a nontrivial task. The implementation of the DEVS formalism can range from one simulator implemented in a single programming language and executed on a single processor to multiple simulators implemented in multiple programming languages and executing on distributed processors (Figure 16.3). Therefore, in

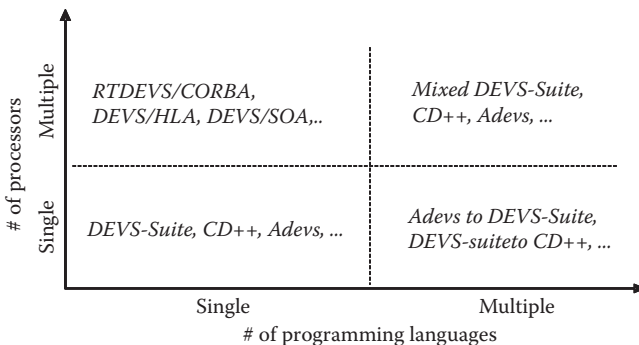


FIGURE 16.3 Centralized and distributed DEVS-based simulators.

order to manage complexity of simulation modeling (and more generally the entire M&S life cycle), standardization is considered a necessity for interoperability among its different simulators and highly beneficial for nondistributed simulators.

16.2 BACKGROUND

16.2.1 THE DEVS FORMALISM

For convenience of reference, let us briefly review the Discrete Event System Specification (DEVS) formalism [16] as a M&S specification that aims at studying discrete event systems. The model consists of components connected together through external port(s) where events are exchanged among models via those ports. Of course, as in any discrete-event simulation, the models being simulated change state only at discrete points in time, upon the occurrence of an event. The DEVS expresses a system as a number of connected behavioral (atomic) and structural (coupled) components. The basic building block of DEVS models is the atomic DEVS model whereas coupled models connect a number of atomic or/and other coupled models.

The P-DEVS formalism [15] expresses a system as a number of connected behavioral (atomic) and structural (coupled) components. A P-DEVS atomic model is formally defined as:

$$M = \langle X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \text{ta} \rangle,$$

where

X is the set of input values

S is the set of states

Y is the set of output values

$\delta_{\text{int}}: S \rightarrow S$ is the internal transition function

$\delta_{\text{ext}}: Q \times X^b \rightarrow S$ is the external transition function,

where

X^b is a set of bags over elements in X

$\delta_{\text{ext}}(s, e, \phi) = (s, e)$

$\delta_{\text{conf}}: S \times X^b \rightarrow S$ is the confluent transition function

$\lambda: S \rightarrow Y^b$ is the output function

$\text{ta}: S \rightarrow R +_0 \rightarrow \infty$,

where

$Q = \{(s, e) \mid s \in S, 0 \leq e \leq \text{ta}(s)\}$ is the total state set

e is the time elapsed since last transition.

At any given time, an atomic model is in some state $s \in S$. It remains in state s for the time period specified by the state time advance function $\text{ta}(s)$, with the assumption of not receiving external events. When the lifetime of the atomic model state expires, the model outputs value $\lambda(s) \in Y$, and changes its state as indicated by the internal transition function $\delta_{\text{int}}(s)$. A P-DEVS model uses a bag of inputs (X^b) to exploit parallelism in the system, hence executing multiple concurrent events simultaneously.

Furthermore, the model also changes its state as defined by the external transition function $\delta_{\text{ext}}(s, e, X^b)$. If the atomic model receives one or more external events $x \in X$ before the expiration of $ta(s)$, it merges the functionality of multiple external transitions into a single one. A confluent transition function (δ_{con}) is used to conclude the next state of the model, thereby resolving the collisions when receiving external events and internal transitions simultaneously.

It is worth noting that the main difference between DEVS and P-DEVS formalisms is the addition of the *confluent* function (δ_{con}), which is responsible for determining the next state of the model when an external input arrives at the same time of an internal transition. The definition of the *confluent* function is determined by the modeler so that the correct behavior can be modeled depending on the system under study. The physical system model is created by integrating different DEVS models together through their input and output ports, resulting in a *coupled DEVS model*. A coupled DEVS model consists of atomic and/or other coupled models connected together. A P-DEVS coupled model is formally defined as:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, \text{EIC}, \text{EOC}, \text{IC} \rangle.$$

Both X and Y define the sets of input and output events, respectively. The D is a set of indices for the components of a coupled model and, for each $d \in D$, M_d is a basic P-DEVS model (atomic or coupled). The external input coupling (EIC) specifies the connections between external and component inputs, while the external output coupling (EOC) describes the connections between component and external outputs. The connections between the components themselves are defined by the internal coupling (IC). Thanks to the property known as closure under coupling, a coupled model can be reduced to a behaviorally equivalent atomic model, and thus be treated as a basic component in construction of more complicated hierarchical models.

16.2.2 DEVS SIMULATION PROTOCOL

DEVS treats a model and its simulator as two distinct elements. The DEVS *simulation protocol* describes how a DEVS model should be simulated; whether in stand-alone fashion or in a coupled model. Such a protocol is implemented by a processor, which can be a simulator or a coordinator. As illustrated in Figure 16.4, the DEVS protocol is executed as follows:

1. It starts with the coordinator first distributing, to each of the simulators in the collection, each other's addresses and then telling each of them to perform their initialization function.
2. A cycle is then entered in which the coordinator requests that each simulator provide its time of next event and determines the minimum of the returned values to obtain the global time of next event.
3. Each of the simulators applies its *computeInputOutput()* method to produce an output that consists of a collection of content (port/value) pairs—for DEVS simulators this is a composite message computed according to the DEVS formalism based on the current state of its model.

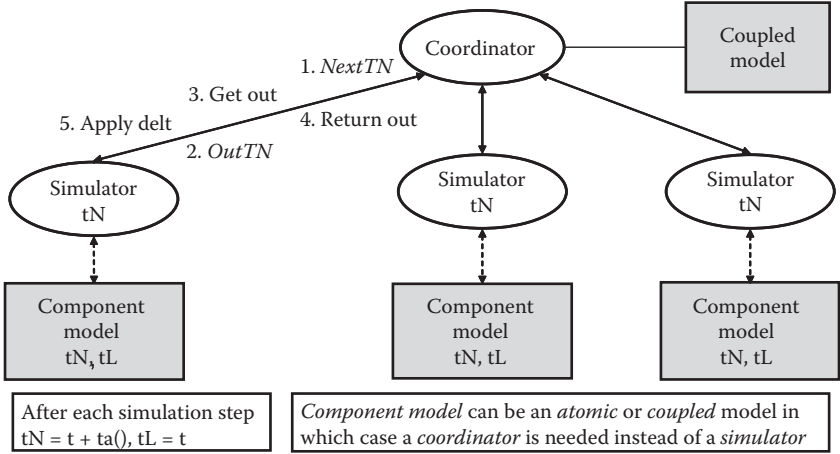


FIGURE 16.4 Abstract parallel DEVS simulator protocol.

4. Then each simulator partitions its output into messages intended for recipient simulators and sends these messages to these recipient simulators—for DEVS simulators these recipients are determined from the output ports in the message and the coupling information that will have previously been received from the coordinator.
5. Finally, each simulator executes its *ApplyDeltFunc* method that computes the combined effect of the received messages and internal scheduling on its state, a side effect of which is to produce the time of next event, tN —for DEVS simulators this state change is computed according to the DEVS formalism and the tN is updated using the time advance of its model.
6. The coordinator obtains the next global time of next event and the cycle repeats.

It should be noted that although the abstract simulator is unique, its actual implementations can be quite different. This is not only because of software design, but also because of the abstract simulator itself. The abstract simulator does not impose any strict ordering for the messages sent/received when multiple components are scheduled to receive inputs at the same time. For example, when the coordinator *nextTN* requests are sent to two or more simulators, the order in which the *outTN* responses are received can be arbitrary. This is expected since the parallel DEVS formalism is defined to assume no dependency between two messages received from one or more components. Therefore, there cannot be any dependencies between two simulators that are used together in distributed fashion.

Various environments, for instance DEVS/SOA (described in Section 18.2) use this simulation synchronization protocol. This protocol has also been modified in order to improve performance by targeting the remote simulation messages. For example, DDSP (Section 18.3) removed unnecessary remote message transmission, and RISE (Section 18.5) further reduces this bottleneck via aggregating simulation messages together in the same XML message.

16.2.3 EXTENSIBLE MARKUP LANGUAGE (XML)

The XML [17,18] is a specification for storing information or describing that information structure in textual documents and is increasingly adopted as a standard for the interchange of information in diverse fields of science, engineering, government, and business [12]. XML plays a major role in the DEVS model representation designs in this chapter as well, as many of them define an XML-based common model representation to be executed in any DEVS environment. It also plays a central role in the middleware standard implementation in the way simulation semantics is exchanged in the form of XML messages or in the underlying technology of the distributed simulation middleware. XML is widely supported by a variety of programming environments, allowing developers to manipulate an XML-based document.

XML was standardized by the World Wide Web Consortium (W3C), and it differs from other markup languages such as HTML in allowing users to define their own tags, provided they adhere to the XML standards. For example, Figure 16.5 shows a customized XML document, listing DEVS tools according to their name and applied programming language. In this example, we defined the “Tools” element to hold all “Tool” elements. Each “Tool” element holds two other elements: “name” and “language” elements.

The power of XML is its simplicity and flexibility to be extended and customized as needed. Thus, it can be used for data storage and transportation by any application. In this case, data are structured so that it can be extended or reduced (“filtered”) as needed. For example, assume that the XML document shown in Figure 16.5 is transferred between two applications. In this case, for instance a new “Tool” element can be easily added to the document. On the other hand, this extensibility comes at a price. For example, the XML document shown in Figure 16.6 describes the same information shown in Figure 16.5 but differently. In this case, element “language” is renamed to element “lang,” and element “name” is converted to attribute “type.” This is easily realized by human readers with simple XML documents. Typically, though, XML documents are handled by software that considers element “name” in Figure 16.5 to be different from attribute “type” in Figure 16.6. Therefore, defining information in XML is not enough to interoperate two applications, but it is only the

```
<?xml version="1.0"?>
<Tools>
  <Tool>
    <name>CD++</name>
    <language>C++</language>
  </Tool>
  <Tool>
    <name>DEVJSJAVA</name>
    <language>Java</language>
  </Tool>
  ...
  ...
</Tools>
```

FIGURE 16.5 XML document example 1.

```

<?xml version="1.0"?>
<Tools>
  <Tool type="CD++">
    <lang>C++</lang>
  </Tool>
  <Tool type="DEVJSJAVA">
    <lang>Java</lang>
  </Tool>
  ...
  ...
</Tools>

```

FIGURE 16.6 XML document example 2.

```

<! ELEMENT Tool
      (name, language)>
<! ELEMENT name (#PCDATA)>
<! ELEMENT language (#PCDATA)>

```

FIGURE 16.7 DTD example.

first step. For that reason, standardizing XML documents (i.e., structure, data, etc.) is necessary to obtain practical interoperability.

The Document Type Definition (DTD) and XML Schema, introduced in the following sections, can be used to validate XML documents (for example, the expected XML building blocks). However, the elements and attributes still need to be standardized so that the communicating application can synchronize correctly.

16.2.3.1 Document Type Definition (DTD)

The DTD define the legal building blocks (i.e., list of legal elements and attributes) of an XML document. For example, Figure 16.7 shows a DTD document that reads as follows: the element “Tool” contains two elements: “name” and “language”; both are of type “(#PCDATA),” which means that these elements are only allowed to contain text (PCDATA stands for Parsed Character Data). The DTD provides more types such as EMPTY to define an empty element, or an element number of occurrence, and so on.

The DTD defines rules for all the XML documents elements and attributes. However, those DTD rules must be defined in the subject XML documents, so that tools can validate XML documents against those rules.

DTDs can be useful, but there are other methods, for instance, XML Schema languages (discussed next). Some of the problems of using DTDs [18] are the need for additional parsers (since it does not follow XML syntax), the lack of support for namespaces or data typing (i.e., requiring data to be integer, string, etc.), and the limited capacity for defining the number of nested children elements (for a parent element).

16.2.3.2 XML Schema

In 2001, the W3C developed a new schema to overcome the DTD shortcomings. This schema is also named XML (which is confusing since DTDs are also XML).

Thus, it is often called XSD or, after version 1.1, XML Schema Definition Language (XSDL) [18]. XML Schema defines the legal building blocks of an XML document, as in the case of DTD. XML Schemas support data types. They are written in XML syntax; hence, are extensible and scalable. For example, Figure 16.8 shows two defined XML elements: “name” of type “string,” and “height” of type “integer.”

16.2.4 DISTRIBUTED SIMULATION INTEROPERABILITY MIDDLEWARE

The main objective of distributed simulation middleware is interfacing different simulation environments, allowing synchronization for the same simulation run across a distributed network, with heterogeneous simulation software components, as shown in Figure 16.2. For example, each simulation environment may differ from other entities in its simulation engine, algorithms, model representation, and formalism. Thus, it comes as no surprise that a number of surveys placed the middleware of distributed simulation as the area of most interest to overcome current distributed simulation challenges and to meet future expectation [13,19]. For example, a recent study carried out by Strassburger, Schulze, and Fujimoto [19] found that future efforts must include the integration of heterogeneous resources, and the joining of computer resources for complex simulations and training sessions. The study also identified some research challenges for distributed simulation middleware such as the *Plug-and-Play* capability. To support such capability, the middleware should couple heterogeneous models effortlessly even at runtime without prior knowledge, hence with a business mentality of “Try-before-you-buy” way of thinking via widely accepted standards in the industry. In fact, the underlying technology is the framework of the middleware; therefore, it affects the flexibility of any standards and its ability to meet current and future expectations.

The Defense sector is currently one of the largest users of distributed simulation technology, mainly to provide a virtual distributed training environment between remote parties, relying on the High Level Architecture (HLA) middleware to provide a general architecture for simulation interoperability and reuse [20,21,22]. The run-time infrastructure (RTI) software layer connects and synchronizes different HLA simulation entities (called federates), as shown in Figure 16.9.

```
<element name="name"
  type="string"/>
<element name="height"
  type="integer"/>
```

FIGURE 16.8 XML schema example.

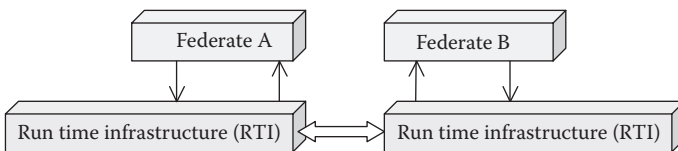


FIGURE 16.9 HLA interaction overview.

Despite the results obtained by this standard, the current adoption of distributed simulation in the industry is still limited. Further, since its adoption in 1996, HLA has only made limited inroads in industry because of a number of issues such as its complexity and lack of interoperability in interfacing different RTI vendors, because the RTI-to-RTI interface is not standardized. Instead, a DEVS-based standard has better prospects for successfully achieving these goals. There are several technologies that have been used to create DEVS middleware. In the following sections, we provide a brief review of each of these technologies, focusing on the way the consumers communicate with each other at the software level. This is important because standard requirements eventually need to be realized by software, hence playing a major role in defining the required changes for legacy systems software. A full analysis of distributed simulation current challenges and future trends is available in Wainer and Al-Zoubi [23].

16.2.4.1 Common Object Request Broker Architecture (CORBA)

CORBA [24] is an open standard for distributed object computing defined by the Object Management Group (OMG). CORBA object services are defined using the Interface Definition Language (IDL), which can then be compiled into programming language stubs such as C, C++, or Java (IDL syntax is similar to other programming languages).

Clients in CORBA invoke methods in remote objects in remote procedure call (RPC) style fashion (using IDL stubs). The method call may return another CORBA handle (i.e., address) where the client can invoke methods of the returned object. CORBA IDL stubs and skeletons glue operations between the client and server sides. The Object Request Broker (ORB) layer provides a communication mechanism for transferring client requests to target object implementations on the server side.

Building distributed simulations using CORBA is straightforward, since CORBA enables application objects to be distributed across a network. Therefore, the issue becomes identifying distributed object interfaces and defining them in IDL, hence a C++/Java local operation call becomes a remote procedure call (hidden by CORBA). Therefore, to support distributed simulation using CORBA all that is required is translating existing C++/Java simulation interfaces into a CORBA IDL definition.

The use of CORBA in newly started projects has recently been declining, though. Henning [25] provides a number of reasons for this decline such as the standards complexity, the politics of accepting new standard designs, and the lack of certain needed features.

16.2.4.2 SOAP-Based Web-Services

WSDL (Web-Services Definition Language) and SOAP (Simple Object Access Protocol) are the main elements that enable SOAP-based Web-services (WS) interoperability. SOAP-based Web-services provide interoperability in a similar way as CORBA: WSDL is equivalent to CORBA's IDL role, where SOAP corresponds to ORB data marshaling/serialization function. Further, Web-service ports are addressed by Unified Resource Identifiers (URI) where CORBA objects are

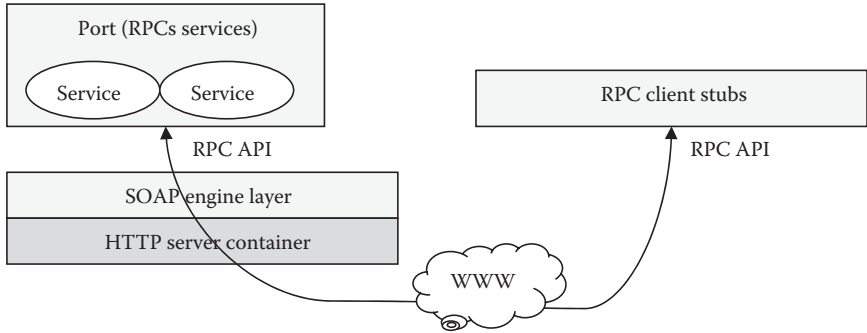


FIGURE 16.10 SOAP-based web service client/server architecture. (From Wainer, G. and Al-Zoubi, K., *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*, Wiley, New York, 2010.)

addressed by references. Both ports and objects contain a collection of procedures (called services by WS) similar to Java/C++ classes. Those procedures glue software components across the network, providing an RPC-style type of software interoperability, as shown in Figure 16.10.

The server exposes a group of services via ports; each service is actually an RPC whose semantics are described via that procedure parameters. Client programmers must construct service stubs with their software at compile time. Clients, at runtime, consume a service by invoking their stub, which, in turn, is converted into an XML SOAP message (describing the RPC call). This message is usually wrapped within an HTTP message, and sent to the server port using the appropriate port URI. Once the message is received at the HTTP server, it is passed into the SOAP layer (usually called the *SOAP engine*). SOAP engines are usually Java programs running inside HTTP servers, called *Servlets*. The SOAP layer parses the SOAP message, and converts it into an RPC call applied to the appropriate procedure of the proper port. The server returns results for the clients in the same way. Thus, the SOAP message role is to provide a common representation among all parties to the invoked procedure at runtime. For example, in Figure 16.11 we can see that, once the procedure “*boolean startSimulation(int in0)*” is invoked by a client, the SOAP engine converts it into the message shown in Figure 16.11 and transmits it to the server, which in turn is converted into the appropriate procedure call. The server subsequently replies in a similar way.

Service providers need to publish their services as XML WSDL documents to enable clients to discover and use them. One way of doing so is via a broker called Universal Description, Discovery, and Integration (UDDI). UDDI is a directory for storing information about web services and is based on the W3C and Internet Engineering Task Force (IETF) standards.

Clients programming stubs (Figure 16.10) are generated via compiling the WSDL document. Figure 16.12 shows a WSDL example for the procedure “*boolean startSimulation(int in0)*.” Lines 1–7 show the messages used by the Web Service to send the request and to handle the response. Lines 9–17 show the *port-type* definition

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5   <SOAP-ENV:Body>
6     <ns1: startSimulation xmlns:ns1="http://WS-Port-URI/">
7       <in0 xsi:type="xsd:int"> 998 </in0>
8     </ns1: startSimulation>
9   </SOAP-ENV:Body>
10 </SOAP-ENV:Envelope>

```

FIGURE 16.11 SOAP message request example. (From Wainer, G. and Al-Zoubi, K., *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*, Wiley, New York, 2010.)

```

1 <wsdl:message name="startSimulationRequest">
2   <wsdl:part name="in0" type="xsd:int"/>
3 </wsdl:message>
4
5 <wsdl:message name="startSimulationResponse">
6   <wsdl:part name="startSimulationReturn" type="xsd:boolean"/>
7 </wsdl:message>
8
9 <wsdl:portType name="DCDppPortType">
10  <wsdl:operation name="startSimulation" parameterOrder="in0">
11    <wsdl:input message="impl:startSimulationRequest"
12      name="startSimulationRequest"/>
13    <wsdl:output message="impl:startSimulationResponse"
14      name="startSimulationResponse"/>
15  </wsdl:operation>
16
17 </wsdl:portType>
18
19 <wsdl:binding name="DCDppPortTypeSoapBinding"
20   type="impl:DCDppPortType">
21   <wsdlsoap:binding style="rpc"
22     transport="http://schemas.xmlsoap.org/soap/http"/>
23   <wsdl:operation name="startSimulation">
24     <wsdlsoap:operation soapAction=""/>
25     <wsdl:input name="startSimulationRequest">
26       <wsdlsoap:body encodingStyle="http://..."
27         namespace="http://..." use="encoded"/>
28     </wsdl:input>
29
30     <wsdl:output name="startSimulationResponse">
31       <wsdlsoap:body encodingStyle="http://..."
32         namespace="http://..." use="encoded"/>
33     </wsdl:output>
34   </wsdl:operation>
35 </wsdl:binding>

```

FIGURE 16.12 Excerpt of WSDL document example.

to define the used operations by the Web Service. Lines 19–35 show the binding part, which defines the message format and ports protocol details. The `<wsdlsoap:binding>` element style (in line 35) attribute uses the RPC-style. The SOAP input/output encoding style for the operation *startSimulation* is defined in lines 25–33.

Once the client programming stubs are generated, programmers still must write their code in the body of those stubs, and so tools are usually needed to generate templates of the source code. In practice, this process can be tedious; in particular, the tools used to generate templates usually avoid overwriting existing stubs, thus, programmers need to rewrite and validate them manually. Also, service composition scalability and lack of dynamicity may be a problem at the client side, because the stub needs to be written and compiled for every service at the server side. Dynamic invocation in Java solves this problem for Java web platforms making it the preferred environment for new developments from this perspective.

RPCs are heterogeneous (because they were invented by different programmers) and distributed simulation connectivity semantics are described on the parameters of those RPCs. RPCs directly influence the interoperability integration effort, since they are actually the Application Programming Interface (API) of a simulation component. Further, RPCs often reflect the software internal implementation since they glue distributed software together.

16.2.4.3 REST-Based Web-Services

The Representational State Transfer (REST) [26] style provides interoperability by imitating the World Wide Web (WWW) style and principles. REST answers the question of what makes the Web architectural style successful; hence, it is a reverse engineering of the Web architecture. REST exposes all services as resources with uniform connectors (channels) where messages are transferred between those resources through those uniform channels. REST is usually implemented using HTTP, URIs, and usually XML because these are the main pillars of the Web today. In this case, resources (services) are named and addressed by URIs, resource connectors are HTTP channels (usually called methods), and connectivity semantics are usually described in XML messages. This type of design is a recipe for a plug-and-play interoperability, as a consumer may search, locate, and consume a service at runtime. To achieve plug-and-play interoperability in a wide range of scales, certain ingredients are needed (as in the case of the Web), as follows:

- Universal accepted standards (such as HTTP, URIs, and XML)
- Implementations are hidden in black boxes (called resources in REST)
- Each resource (or service) has uniform connectors/channels (REST uses HTTP channels/methods)
- Each resource (or service) is addressed with the universal unique identifier (i.e., URI in case of REST)
- Message-oriented type of connectivity semantics (usually as XML)

REST principles perfectly match the HTTP, since it is the existing Web protocol. HTTP exposes all services as URIs that can be accessed via few well-defined channels (called methods in HTTP standards): GET (to read a resource), PUT (to create/update a resource), POST (to append to a resource), and DELETE (to remove a resource). Thus, REST clients always communicate in the same standardized way, as shown in Figure 16.13. The client must know three things to invoke a service: (1) the service URI, (2) the HTTP channel, and (3) the message semantics and format. For example, a Web browser invokes a service from a Web site by sending a request via the GET channel to that Web site URI. In response, the Web site transfers its representation in the form of a message (e.g., HTML) to the client, thereby transferring the representational state to the client, as indicated by the REST name.

In contrast, SOAP-based WS exposes services as RPCs in ports where each port is addressed in a single URI. RPCs have heterogeneous interfaces and they have a split implementation. It is worth noting that SOAP-based WS uses the HTTP POST channel to transmit the description of all RPCs as SOAP messages.

RESTful Web-services [27] are gaining increased attention with the advent of Web 2.0 [28] and the concept of mashup (e.g., IBM Mashup Center enterprise solutions [29]). At this point RESTful WS is supported by the leading WS tools in conjunction with the SOAP-based WS. Mashup applications deliver new functions and services on the Web by combining different information or capabilities from more than one existing source.

For example, suppose a Web site is used for trip planning. In this case, based on the trip destination, this Web site may display weather forecast, hotel rates, and average meal cost of the trip destination. Of course, it is impractical for a server to implement such capabilities for every possible destination in the world. Thus, this Web site obtains the necessary information from different sources based on the planned trip, giving the impression of implementing these capabilities. Further, this Web site may search for this information at runtime before being able to read it. Moreover, the Web site can also cache the searched results, hence enhancing performance of different users heading to similar destination. Achieving

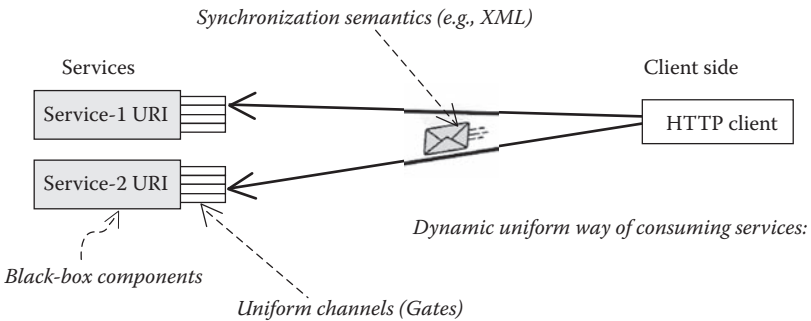


FIGURE 16.13 RESTful web-service client/server architecture.

plug-and-play interoperability (in the style of the World Wide Web) for the Web site in this example would not be feasible, and mashing up the various services dynamically at runtime would be very difficult. For example, the Web site in the example would need to know in advance that it has to send a request to the URI of an information source via the corresponding HTTP GET channel, so it can retrieve an HTML document. If one must build a programming stub for every possible information source, the application design would be very complex and interoperability unfeasible.

Instead, the APIs of RESTful applications are expressed as URI templates [30] that can be created at runtime. Variables in URI templates (written within braces {}) are assigned at runtime by clients before a request is sent to the server, enabling clients to name their services URIs at the server side. For example, the username in template `<.../users/{username}>` can be substituted with any string to get the actual URI instance (such as `<.../users/user1>` or `<.../users/user2>`). Further, URIs may include query variables to define the request scope by appending them to a URI after a question mark “?” For instance, a request via GET channel to URI `<http://www.google.com/search?q=DEVS>` would instruct the Google search engine to return information only about keyword “DEVS.”

Service providers usually describe their URI template either textually and/or as a Web Application Description Language (WADL) [31] document. WADL is an XML description that describes a RESTful API, hence corresponds to WSDL in SOAP-based WS. WADL describes each service as shown in Figure 16.14. In this example, line #1 shows the service URI. Lines 2–11 describe the PUT channel. Lines 3–5 state that XML is the supported format by this message. Lines 6–10 define the response of requests made on the PUT channel. In this case, it only lists the possible generated faults. Lines 12–17 define the other supported channels.

```

1  <resource path="sim/workspaces/{userworkspace}">
2  <method id="PutWorkspace" name="PUT">
3    <request>
4      <representation mediaType="text/xml"/>
5    </request>
6    <response>
7      <fault mediaType="text/html" status="400">
8        <doc>BAD_REQUEST: Error while parsing XML document</doc></fault>
9      ...
10   </response>
11 </method>
12 <method id="GetWorkspaceState" name="GET">
13   ...
14 </method>
15 <method id="DeleteWorkspace" name="DELETE">
16   ...
17 </method>
18 </resource>

```

FIGURE 16.14 Excerpt of WADL document example.

16.3 DESIGNING INTEROPERABILITY OF DEVS COMPONENTS

The goal of a DEVS standard would be to provide a simple and mostly automated way of executing simulations that involve remote and/or heterogeneous DEVS models. This can be achieved by taking two different approaches to be discussed in the following sections: *model-based* interoperability and *simulator-based* interoperability.

16.3.1 MODEL-BASED INTEROPERABILITY

The model-based approach aims at providing interoperability by making it possible to integrate heterogeneous DEVS models in a single simulation. To achieve this, models themselves are deployed, and for a large number of models it is easier to comply with this Model Interface rather than complying with the Core Simulator Interface. Using a standard representation of models, like the ones proposed in Chapter 17, different techniques inspired by Model Driven Engineering (MDE) [32] can be employed to improve the process.

In online (dynamic) model-based interoperability solution, models themselves are deployed as services instead of simulators. Using this model driven approach, the operations invoked through the network are no longer simulation mechanisms, but model functions, such as δ_{in} , δ_{ext} , the *time advance function*, and so on. This extends the scope of practical DEVS interoperability. For a large class of existing model implementations it is easier to comply with this Model Interface (compared to complying with the Core Simulator interface).

Using a standard representation of models, different techniques can be employed to smooth the process by making the generation of the adapter mostly automated. The following describes this approach in the case of web services (see Figure 16.15), but could be extended to other integration middleware.

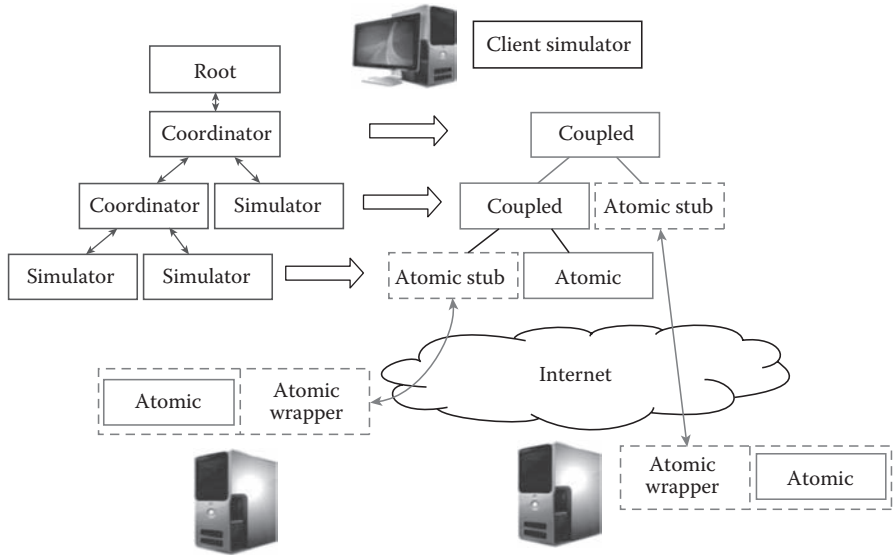


FIGURE 16.15 Local simulation of distant and local models.

Given a model written for a specific framework (CD++, DEVSTJava, James, etc.), the first thing to do is to generate its representation in a platform-independent language (which would allow the description of the model for all the tools). This description, and more particularly the port types defined in it, can be fed into model transformation tools to generate:

- A web service description, as a WSDL file
- A wrapper that will adapt the model interface to the service interface, by forwarding the operation calls

Therefore, each model is assigned its own description file and its model wrapper; which are fully specialized for this particular model. In this way, model consumers are provided with complete knowledge of the nature of the messages expected by the model, increasing both type safety and understanding of the model semantics.

Once this is done, the client can retrieve the description of the model and generate a stub that will conform to its framework and behave exactly as the original model, by invoking the wrapper service. The automatic generations mentioned above can be seen as automatic transformations in the model-driven engineering terminology. Regarding remote coupled models, two solutions are possible. The first is to use some kind of adapter, as in the simulator-based approach, which will make the coupled model appear as an atomic one. The second possibility is to take into account that coupled models do not have any operational semantics. All they do is describe their ports and their couplings, in a static way. As a consequence, we can use their XML representation to generate a copy of the model in the local framework, without losing any useful information.

In a nutshell, model providers:

1. Write a model in their favorite DEVS framework.
2. Automatically generate the model's XML description, from which they generate a WSDL file and a web service encapsulating the model.
3. Deploy the web service, providing access to the underlying model.

On their side, the model consumers have to:

1. Find the location of the models they need, using some kind of model directory or more classical discovery procedures such as web searches or acquaintances.
2. Give the addresses of the models' web services to a generation tool that will download the WSDL files and generate client stubs adhering to the modeler framework.
3. Run the simulation. The framework's simulator deals with local and remote models without even knowing it.

16.3.2 SIMULATOR-BASED INTEROPERABILITY

The main idea of this approach (used in DCD++ and DEVS/SOA, to be presented later), is to have a collection of simulation services distributed over the Internet.

These services provide several operations for simulating atomic or coupled DEVS models in a unified manner, by using the DEVS simulation protocol and the closure under coupling property of coupled models. The overall simulation is coordinated by a main service, which acts like an entry point for the user. This architecture is summarized in Figure 16.16.

In order to communicate, the DEVS services expose a standard interface that accepts the usual simulation protocol messages: *initialize*, *get time of next event*, *run transition* and so on.

From a user perspective, the simulation process consists of:

1. Writing a coupled DEVS model using their favorite framework.
2. Providing a list of DEVS servers on which they wish to distribute the simulation.
3. Deploying the DEVS models to the servers (either by downloading them from their location or activating existing remote models).
4. Starting the simulation

It should be noted that the above is one form of many possible protocols that can provide various forms of conservative and optimistic simulation, each of which must be proved to be correct as a implementation of the DEVS closure under coupling property [10].

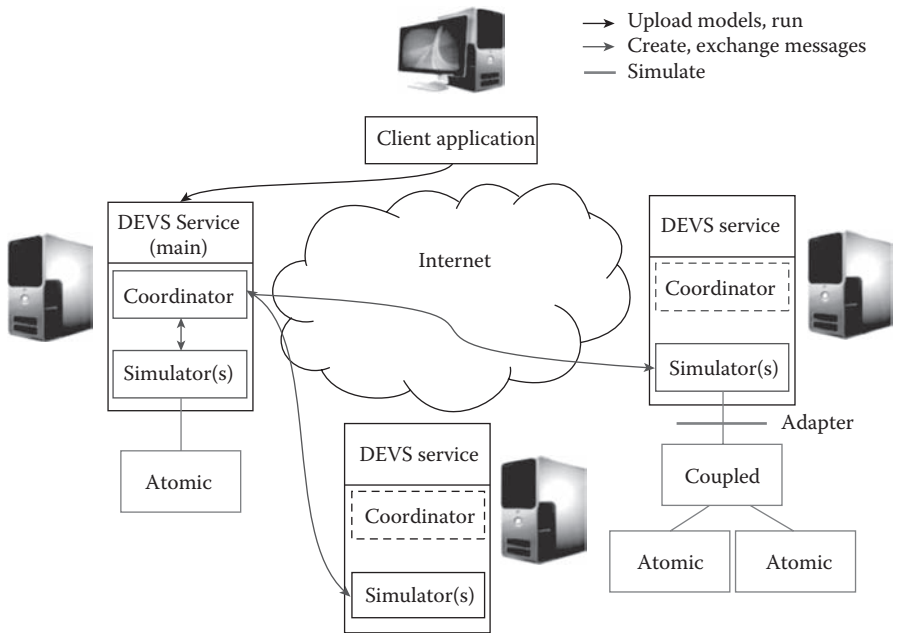


FIGURE 16.16 DEVS components interoperability through simulators communication.

Implicit in the above description are the following constraints involving methods in the *CoreSimulatorInterface*:

- The *sendMessages()* method “must” employ the *putContentOnSimulator()* method as follows: for any simulator to which it wishes to send a content, it must call the recipient’s *putContentOnSimulator()* method with the recipient and the content as arguments.
- Further, in applying its *computeInputOutput()* method, a simulator “must” be able to interpret the contents (satisfying the *ContentInterface*) it has received from the other simulators.

Notice that we cannot enforce these “must” requirements and cannot prove that the simulation executes a desired behavior, unless we are given further information about its behavior. One way to ensure these conditions is when the simulators are truly DEVS simulators in that they satisfy the interfaces and constraints given below. Failing this additional rigor, the interoperation involving DEVS and non-DEVS is purely at the technical level similar to that of a federation of simulators in HLA. This contrasts with the situation in which the federation is in fact derived from a DEVS coupled model for which correct simulation of the coupled model is guaranteed according to the DEVS formalism.

To sum up both interoperability solutions exposed here, we can say that the simulator-based approach (i.e., via distributed simulation middleware, as discussed in Chapter 18) is aimed at coupling distributed or local models over remote servers, whereas the model-based approach, discussed in Chapter 17, aims at integrating distributed models in a local simulation.

16.4 COMPARING EXISTING DESIGNS AND IMPLEMENTATIONS

The two objectives discussed (*simulator-based* interoperability and *model-based* interoperability) face different challenges and have different purpose. Thus, they form two different sets of standards, and they can be treated separately. In other words, a simulation tool does not need to conform to both sets of standards to be able to reuse other resources from different working groups. In recent years, there have been a number of efforts to deal with these issues, and we will compare and discuss these approaches here from the two objective viewpoints.

16.4.1 STANDARDIZING DEVS MODEL REPRESENTATION

Standardizing DEVS model representation allows a model to run on any DEVS simulation environment. This is powerful in the sense that a model can be retrieved from a repository and run locally on a different tool other than the originally intended running environment of that model. This is highly recommended to avoid performing distributed simulation between remote environments for obvious performance reasons.

Different working groups have used XML as a mechanism for interchanging model information. In order to do so, we need to define an XML vocabulary, which is formally defined by an XML Schema against which every file written in

the vocabulary can be automatically validated. This arrangement gives an XML vocabulary several important advantages over *Platform Specific Models* (PSMs):

- Validation against a schema promotes stability of the standard.
- The schema can restrict data types.
- The schema can define key data to insure consistency in the models (including the validation of important properties in the data values, such as uniqueness or duplicity of use, when needed).
- The schema can be extended to include constraint types or simulator directives. Files that are valid under the original schema continue to be valid under the extended one (though, of course, the reverse is not guaranteed).

This broader relevance has benefits for DEVS systems:

- When scenarios are stored in XML format, DEVS technology results are more readily integrated into broader information technology infrastructures.
- XML is the data interchange language of Web services.
- XML lends itself very well to compression.
- XML-based *eXtensible Stylesheet Language Transformations* (XSLT) offer a convenient way to specify translations of XML documents. If a DEVS scenario (with perhaps corresponding results) is stored in XML, then XSLT is easily applied to the scenario to produce a Web browser document that displays the results data in reports that are suitable for people to read.
- Encryption standards such as XML Encryption are emerging for XML data [12]. This encryption is important to commercial DEVS applications where the scenarios contain confidential data.

Libraries for these purposes—validating files, defining keys, compressing files, and the like—are provided by numerous XML tools designed for manipulating and parsing XML data. It suffices to define our XML vocabulary in the form of a schema to utilize these tools. This contrasts to ad hoc formats that require writing, debugging, and maintaining routines equivalent to these tools.

Figure 16.17 shows a DEVS/XML standard definition framework. The DEVS/XML Schema includes the grammar for both the structure and behavior of DEVS models (i.e., coupled and atomic models). When it is defined, the system designer is able to create DEVS models in XML that may be validated against the DEVS/XML Schema. This forms a *valid DEVS/XML Model*. Next, there are two possibilities. The first one is to transform the previous scenario into a PSM for a concrete simulation engine, such as DEVSJAVA, DEVS/C++, CD++, xDEVs, and so on. The second one is the development of a DEVS/XML simulator in a high-level programming language. Finally, a reverse engineering layer must be provided to come back from the PSMs to the *Platform Independent Models* (PIMs). The reverse engineering is the most difficult to accomplish. In this case DEVS/XML must include a (complex) parser to obtain the corresponding DEVS/XML Scenario from a set of DEVS PSM files (atomic and coupled models), which compose the DEVS Scenario. Thus, in some cases, other approaches are needed to facilitate the reverse engineering.

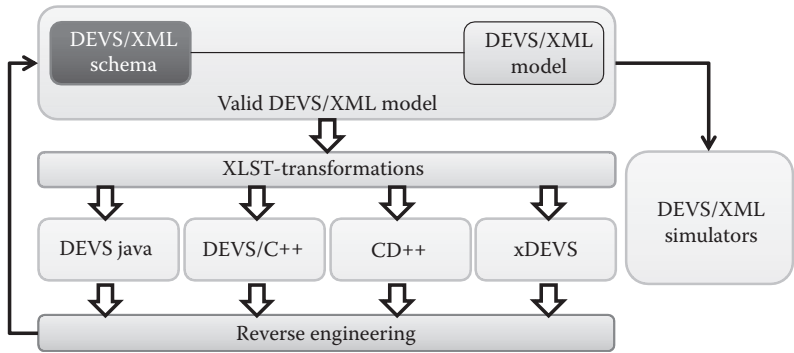


FIGURE 16.17 DEVS/XML standard definition.

TABLE 16.1
Different DEVS/XML Designs of Standard

	Reference	Structure	Behavior	PIM→PSM	XML Simulator	Reverse Engineering
DCD++	[39]	✓	✓	✓	×	×
DEVS/Schema	[33]	✓	✓ ^a	✓	×	✓ ^c
DEVS/SCXML	[35]	✓	✓ ^b	✓	✓	×
DEVSMML	[37]	✓	✓	✓	×	✓ ^c
DML	[38]	✓	✓	✓	×	×
eUDEVS	[36]	✓	✓	✓	×	×
FD-DEVS	[34]	✓	✓ ^a	✓	✓	×
RESTful DCD++	[40]	✓	✓	✓	×	×

^a Behavior limited by the corresponding Schema definition.
^b Behavior limited by the corresponding SCXML Schema definition.
^c Implemented for two simulation platforms (DEVJSJAVA and xDEVJS).

Table 16.1 shows different approaches (some not presented here) defined to implement a DEVS/XML standard, including DEVSMML and DML that will be described in the next chapters. All the approaches define a DEVS structure, whereas the behavior is completely defined in eUDEVS, DEVSMML, and DML. The transformations from PIMs to PSMs are also defined in all the standards. In addition, two XML simulators have been developed: one for FD-DEVS and another one for DEVS/SCXML. Finally, the reverse engineering has been partially applied to DEVS/Schema and DEVSMML.

The different DEVS implementations have evolved independently over the years, specifying their own model representation and rules. In practice, most of the DEVS tools would compile to a standard representation in their local specification; hence, avoiding redesigning the software implementation to handle the standard representation directly. Figure 16.18 shows a typical sharing scheme. In this example, the DEVS model representation is stored in a repository from which it is retrieved to be executed on the target platform. The main challenge, as shown in Figure 16.18,

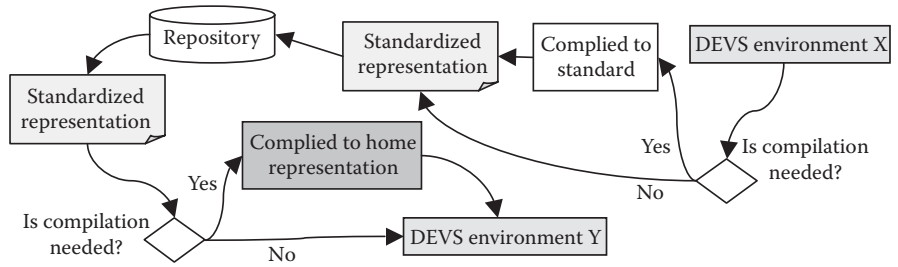


FIGURE 16.18 Standardized model representation flow.

is at the target platform and pertains to compiling the standardized representation into a local representation. This is actually different from converting a homegrown representation into a standardized representation, since the resulting representation may require producing new source code. As such, the newly produced source code must be integrated and compiled with a legacy source code before being able to run the simulation. This problem extends beyond the use of a programming language. For example, converting an XML representation of an atomic model into a C++ class does not guarantee its successful integration with a C++ DEVS-based tool. In fact, many other issues may be involved such as, for example, registering the atomic model, including other appropriate legacy C++ header files, compiler version, and so on. In other words, one should be familiar with the internal implementation structure to be able to compile newly injected source code. Consequently, the main effort that is to be expected when moving to support the DEVS standardized representation models is developing a converter from/to the homegrown representation to/from the standardized representation. This challenges the designs presented here to provide a value proposition that makes it worth the implementation effort to develop such a converter. This is an essential issue to overcome in order to reach practical standards that are widely adopted. One possible path to pursue is in working together in developing the common feature of this converter where each party can then extend it to resolve their internal representation specific issues.

Table 16.2 shows the list of designs for standardizing DEVS model representations. Except for DEVSMML, which uses DTD, all of the designs use the XML schema (also called XSD) for the DEVS representation. XML schemas are more powerful than DTDs and became a W3C Recommendation on May 2, 2001. DEVS/SCXML and CoSMoS include graphical notations on top of the XML model representation. In this way, standardization in XML is also needed for the graphical notations before it is converted into DEVS XML representation. This introduces another standardization layer, which will add more complexity to a more broadly adopted standardization process. DML tackled the targeted programming languages, allowing XML representation to be more aware of used programming language. This is an interesting approach, making XML conversion to a programming language easier. However, this requires the modelers to know all programming languages that a model may eventually be converted to. It is worth noting that all of the designs assume that DEVS models are written in a programming language. However, this is not always true. For example, CD++ defines coupled models as textual scripts but

TABLE 16.2
Summary of Standardizing DEVS Model Representation Designs

Design	Description
CoSMoS (Component-based System Modeling)	It deals with developing models graphically and code generation.
DEVSMML	Document Type Definition (DTD) XML representation for both DEVS atomic and coupled models
DEVSM/SCXML (DEVS State Chart XML)	DEVS State Machine (XFD-DEVS SM) model is XML representation that needs to be converted to run by simulation. This XFD-DEVS SM is converted from SCXML, which is an XML representation of a UML model.
DEVSM/XML	XML representation for both DEVS atomic and coupled models
DML (DEVS Markup Language)	It provides XML for DEVS models while making it more programming language aware, such as adding snippet code in XML.
SOAP/RESTful DCD++	XML Model representation, and XML distributed model partitioning/interconnections.
XFD-DEVS (XML Finite Deterministic DEVS)	It provides XML Schema Definition (XSD) representation for both DEVS atomic and coupled models. It is tabular-based methodology to automate the DEVS state machine specification process.

defines atomic models as C++ classes. This type of differences may lead to having a specific converter for a simulation environment (Figure 16.18).

To conclude, the biggest challenge that these designs face is bringing them closer together to form a more broadly based new standardized representation. This is because existing software needs to be changed to conform to the new standardized representation.

16.4.2 STANDARDIZING INTEROPERABILITY MIDDLEWARE

Interfacing different simulation environments is intended to synchronize the same simulation run across distributed network (i.e., distributed simulation). Distributed simulation can provide many benefits such as heterogeneous model reuse without the need to standardize model representation. Other benefits include removing impediments to moving people/equipment to other locations and information hiding—including the protection of intellectual property rights. This type of interfacing is the main objective of the distributed simulation middleware, which connects all participants.

The middleware is the area of most interest to overcome current distributed simulation challenges and to meet future expectation, as indicated by a number of surveys of experts of different simulation background [13,19]. These surveys also identified some research challenges for a distributed simulation middleware:

1. *Plug-and-Play* capability: The middleware should couple heterogeneous models effortlessly even at runtime. Thus, standards should be simple to understand, quick to support, and with low-risk for legacy systems (i.e.,

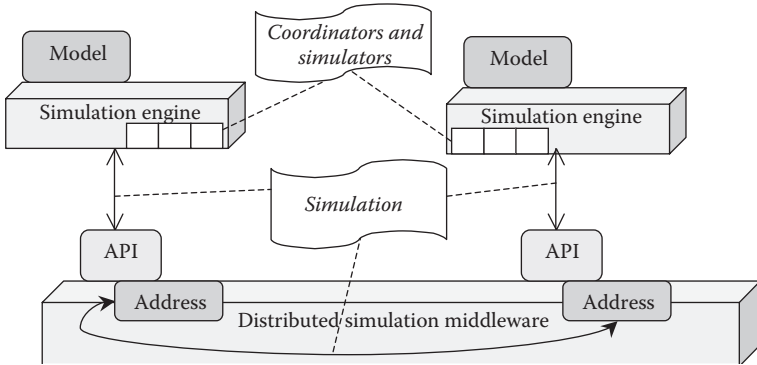


FIGURE 16.19 Standardized middleware reference model

changing implementation is not acceptable). Further, plug-and-play indicates a high-level of dynamicity; for instance, simulations should be able to join/disjoin a simulation session with a history. In other words, having to change the source code and then compile it in order to be able to connect with a simulation component is not a plug-and-play approach.

2. Automated *semantic interoperability between domains* is necessary to achieve the plug-and-play challenge. To automate this process, connectivity *semantic* must be defined in the form of messages (i.e., XML) rather than programming parameters as in most of RPC-style systems.

Figure 16.19 shows the relationship between the middleware and the simulation environments. A simulation engine executes a number of Coordinators (i.e., coupled model simulator) and Simulators (i.e., atomic model simulator). Simulation engines synchronize simulation activities via the middleware. In this case, a simulation engine packs the simulation information (i.e., called *simulation semantics*) and passes it to the middleware via the API. Therefore, the simulation semantics and API play a major role in making the standards flexible enough for dynamicity and future improvements, since simulation engines influence each other only through this information. Afterward, the middleware delivers the simulation semantics to the destination *address*.

Each of the major elements shown in Figure 16.19, simulation semantics, API, and addresses, is important and directly affect the interoperability standards. Further, the underlying technology framework influences the design of those elements. The presented designs here use three technologies: CORBA, SOAP-based Web-services, and RESTful Web-services. We illustrated all of the distributed simulation issues and future trends in [23].

The following requirements can be compiled for a middleware standard that combines the qualities of the separate approaches in order to achieve a broad based yet practical standard:

- Respect existing systems without requiring them to change software implementation dramatically. In other words, it needs to avoid standardizing software implementation. This is important to allow different teams to evolve

independently. The standards should allow a team to extend their tools with other non-standard related features without breaking the standard support. Thus, standards should not make any assumptions on how a certain DEVS or non-DEVS implementation package is realized.

- It must handle any type of DEVS model representation. In this case, a model is placed in a simulation engine domain that is capable of executing it. This is important because legacy models can still be used.
- It must be flexible enough for future improvements and changes. In other words, the standards should be portable to another technology that may appear in the future.
- The standards should support a full simulation experiment environment. The method should define the modeler-required steps to couple distributed models, execute the simulation, and retrieve results. Further, the standards should allow different non-related sessions (i.e., experiments) to be executed simultaneously.
- It must not take much effort to support with low risk to existing systems.
- It must be extendable to interoperate DEVS-based tools with non-DEVS tools.

Table 16.3 summarizes the four current approaches with respect to the reference model shown in Figure 16.19. *DEVS/SOA* defines interfaces for simulators and coordinators where simulation information is passed in the form of programming procedures. The RPC call is then converted to a SOAP message by a SOAP engine and wrapped in an HTTP message. *DEVS/SOA* variant 2 goes on to provide a DEVS Namespace, derived from the XML Namespace, to support interoperability among DEVS implementations that employ different internal representations of the values within DEVS messages. *DEVS/SOA* distributes models (via DEVSMML script) based on a DEVS tool, source code, and a machine IP address. This may be a problem when conducting multiple simulation sessions simultaneously. The *Distributed DEVS Simulation Protocol* is also based on SOAP WS, as in the *DEVS/SOA* case. However, it defines a single WS port to wrap an entire DEVS domain, which allows fewer RPCs and more implementation hiding. All of the simulation information is exchanged in XML messages that are sent as SOAP attachments. This also makes it less dependent on the SOAP WS framework. All models are wrapped in a single coupled model simulated by the main domain. DEVS models are distributed among domains according to an XML document. The *Shared Abstract Model* is based on CORBA. It defines interfaces for simulators and coordinators (called proxies) where simulation information is passed in the form of programming procedures. The *RESTful Interoperability Simulation Environment* (RISE) is based on RESTful Web-services. It uses HTTP uniform channels (methods) to communicate all of the simulation information as XML messages. Each domain is wrapped behind three URIs that can be named at runtime. It uses a single Time Management component to synchronize all domains when the conservative-based method is applied. On the other hand, domains can directly exchange those XML messages when an optimistic-based method is used. RISE assumes an entire model is placed in each domain. Modelers connect the input/output ports of all models by means of an XML configuration document. RISE does

TABLE 16.3
Summary of Standardizing DEVS Interoperability Designs

Design	Technology	API	Simulation Semantics	Addressing	Simulation Algorithm	DEVS Model Level
DEVS/SOA (Variant 1)	SOAP WS	RPC-style	Procedure Parameters	WS Port (URI) per coordinator and simulator	Conservative	Coupled
DEVS/SOA (Variant 2)	SOAP WS	RPC-style	DEVS (XML) Namespace	WS Port (URI) per coordinator and simulator	Conservative	Coupled
Distributed DEVS Simulation Protocol	SOAP WS	RPC-style	XML	Single WS Port (URI) wrapper per simulation environment	Conservative	Coupled
Shared Abstract Model Approach	CORBA	RPC-style	Procedure Parameters	CORBA object/proxy per coordinator and simulator	<i>Undefined</i>	Coupled
RESTful Interoperability Simulation Environment (RISE)	RESTful WS	Four HTTP Uniform Channels (methods)	XML	Three resources (URI) per simulation domain	Conservative/Optimistic	Independent

not interfere with internal domains details, including DEVS coordinators and simulators, hence making it independent of the DEVS formalism.

The main objective of standardized DEVS would be to enable different DEVS implementations to interface and coordinate among each other to simulate the same model structure across their domains. A standardized DEVS imposes some requirements and assumptions in order to make the proposed protocol achievable and acceptable by different teams. Because of these requirements, minimum design changes are expected to each DEVS implementation, mainly by hiding the detailed implementation behind a wrapper and focusing only on the exchanged information that is needed to perform simulation and coordination among distributed models. Such Coordinators must find the models that they want to send their messages without worrying about other details such as constructing messages in XML documents or where which specific DEVS implementation is simulating the other models.

The DEVS simulation protocol was also discussed to show the exchanged messages format and contents. Further the overall simulation coordination showing each DEVS domain role in the phases of each simulation cycle was described. In addition, the head/proxy structure was introduced to coordinate a coupled model simulation in the distributed environment in order to reduce the number of exchanged messages across the network. However, a standard is not limited to one algorithm, hence more schemes may be added in the future and used easily by including this information in the exchanged XML documents.

The presented protocol assumed the usage of web-services technology as the communication framework. However, the protocol takes into account that the DEVS simulation messages should be easily ported to different communication architecture in the future, if needed to do so. This is accomplished by constructing all simulation messages in XML documents so that any changes in the protocol messages will be done to those XML documents rather than to the web-services specific communication interfaces. The DEVS Namespace formulated in DEVS/SOA is a foundation for such an extension.

16.5 SUMMARY

In this chapter, we have introduced the main ideas and concepts about DEVS M&S standardization. We discussed different designs to bridge the gap between implementations of DEVS M&S software, most of which are based on standard XML notations to share information, and to achieve model reuse.

Two different interoperability objectives must be addressed: how to represent DEVS models that can interoperate in a platform-independent fashion and how to standardize the interoperability of the middleware used for simulation purposes. These objectives have different purposes, and they should be treated separately. The following two chapters will focus on each of these two objectives in detail.

Different technologies have been proposed to deal with interoperability middleware: CORBA, XML, Web Services (SOAP-based and RESTful), the High Level Architecture and other technologies. In every case, this middleware must execute a DEVS simulation protocol, and should be able to parse and interpret the model representation.

We have discussed different simulation environments and compared their facilities, including DEVS/Schema, FD-DEVS, DEVS/SCXML, eUDEVS, DEVSML, and DML. These tools have evolved independently over the years, and a need for interfacing these tools has emerged. We also presented a discussion of different representation designs, including DEVS/XML, DEVSML, XFD-DEVS, DEVS/SCXML, DML, and CoSMoS.

Finally, we discussed the different aspects (including underlying technology, API style, synchronization algorithms, etc.) for various DEVS environments, including DEVS/SOA (and its different variants), the Distributed DEVS Simulation Protocol, the Shared Abstract Model approach, and RISE (the RESTful Interoperability Simulation Environment).

REFERENCES

1. Fujimoto, R. M. *Parallel and Distribution Simulation Systems*. New York: Wiley, 1999.
2. Radhakrishnan, R., D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. "An object-oriented time warp simulation kernel." *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments, LNCS 1505*, 13–23, 1998.
3. Bailey, C. A., R. M. McGraw, J. S. Steinman, and J. Wong. "SPEEDES: A brief overview." *Proceedings of SPIE*, 190–201, Orlando, FL: Bellingham, WA: SPIE, 2001.
4. IEEE Std. 1278. "IEEE standard for modeling and simulation. Distributed Interactive Simulation (DIS)." *IEEE Std. 1278*. New York: IEEE, 1995.
5. IEEE Std. 1516.1-2000. "IEEE standard for modeling and simulation. High Level Architecture (HLA)—Federate Interface Specification." *IEEE Std. 1516.1-2000*, i–467. New York: IEEE, 2001.
6. Noseworthy, J. R. "Developing distributed applications rapidly and reliably using the TENA middleware." *Proceedings of Military Communications Conference, MILCOM 2005*. Atlantic City, NJ, 2005.
7. Zeigler, B. P., and H. Sarjoughian. "ACIMS—Software site," <http://www.acims.arizona.edu/SOFTWARE/software.shtml> [Accessed: November 2009].
8. Wainer, G. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2009.
9. Nutaro, J. "Adevs, A Discrete Event System simulator," <https://sourceforge.net/projects/adevs/> [Accessed: November 2009].
10. Kim, S., H. S. Sarjoughian, and V. Elamvazhuthi. "DEVS-Suite: A component-based simulation tool for rapid experimentation and evaluation," *Spring Simulation Multi-Conference*, San Diego, CA, 2009.
11. Himmelspach, J., R. Ewald, and A. M. Uhrmacher. "A flexible and scalable experimentation layer for JAMES II." *Proceedings of the Winter Simulation Conference*. Edited by S. Mason, R. Hill, L. Moench, and O. Rose, 827–35, Winter Simulation Conference, December 2008.
12. Zeigler, B., and P. Hammonds. *Modeling & Simulation-Based Data Engineering: Pragmatics into Ontologies for Net-Centric Information Exchange*. New York: Academic Press, 2007.
13. Freigassner, R., H. S. Sarjoughian, H. Praehofer, and B. P. Zeigler. "A systems approach to a validation and verification methodology within the FEDEP six-step process." *Simulation Interoperability Workshop, 01E-SIW-085*, June, London. New York: IEEE Press, 2001.
14. Boer, C., A. Bruin, and A. Verbraeck. "A survey on distributed simulation in industry," *Journal of Simulation* 3, no. 1 (2009): 3–16.

15. Chow, A. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *Simulation Transactions* 13, no. 2 (1996): 55–67.
16. Zeigler, B., T. Kim, and H. Praehofer. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. New York: Academic Press, 2000.
17. *Extensible Markup Language (XML) 1.0* (Fifth Edition). <http://www.w3.org/TR/REC-xml/> [Accessed: March 2010].
18. Goldberg, K. H. *XML: Visual QuickStart Guide*, 2nd ed. Berkeley, CA: Peachpit Press, 2009.
19. Strassburger, S., T. Schulze, and R. Fujimoto. "Future trends in distributed simulation and distributed virtual environments: Results of a peer study." *Proceedings of Winter Simulation Conference (WSC 2008)*, Miami, FL, 2008.
20. "IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA)—frameworks and rules." *Technical Report 1516*. New York: IEEE, 2000a.
21. "IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA)—federate interface specification." *Technical Report 1516.1*. New York: IEEE, 2000b.
22. "IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA)—object model template (OMT) specification." *Technical Report 1516.2*. New York: IEEE, 2000c.
23. Wainer, G., and K. Al-Zoubi. "An introduction to distributed simulation." Chapter 11 in *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*, Edited by C. Banks and J. Soklowski. New York: Wiley, 2010.
24. Henning, M., and S. Vinoski. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999.
25. Henning, M. "The rise and fall of CORBA," *Communications of the ACM* 51, no. 8 (2008). Also available at <http://queue.acm.org/detail.cfm?id=1142044> [Accessed March 2010].
26. Fielding, R. T. *Architectural Styles and the Design of Network-Based Software Architectures*, Doctoral dissertation, Irvine, CA: University of California, 2000. Available at: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> [Accessed October 2008].
27. Richardson, L., and S. Ruby. *RESTful Web Services*. Sebastopol, CA: O'Reilly Media, 2007.
28. O'Reilly, T. *What is Web 2.0?* <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html> [Accessed May 2009].
29. IBM Mashup Center. <http://www-01.ibm.com/software/info/mashup-center/> [Accessed June 2009].
30. Gregorio, J. *URITemplates*. <http://bitworking.org/projects/URI-Templates/> [Accessed October 2008].
31. *Web Application Description Language (WADL)*. <https://wadl.dev.java.net/> [Accessed October 2008].
32. Schmidt, D. C. "Model-driven engineering—Guest editor introduction," *IEEE Computer* 39, no. 2 (2006): 25–31.
33. Risco-Martín, J. L., S. Mittal, M. Á. López-Peña, and J. M. de la Cruz García. "A W3C XML schema for DEVS scenarios," *DEVS Integrative M&S Symposium DEVS'07, Spring Simulation Multi-Conference*, March 2007.
34. Mittal, S., B. P. Zeigler, and M. H. Hwang. *XFD-DEVS: XML Schema for Finite Deterministic DEVS*, <http://www.duniptechnologies.com/research/xfddevs/>
35. Risco-Martín, J. L., S. Mittal, B. P. Zeigler, and J. Mendel. "From UML statecharts to DEVS state machines using XML." *Multi-Paradigm Modeling, IEEE/ACM International Conference on Model-Driven Engineering Languages and Systems*, Nashville, TN, September 2007.

36. Risco Martin, J. L., S. Mittal, J. Mendel, and B. P. Zeigler. "eUDEVS: Executable UML using DEVS theory of modeling and simulation," Invited paper to *SIMULATION: Transactions of SCS*, 2009.
37. Mittal, S., J. L. Risco-Martín, and B. P. Zeigler. "DEVSMML: Automating DEVS execution over SOA towards transparent simulators." *Spring Simulation Multiconference, SpringSim 2007*, 287–95, Norfolk, VA, 2007.
38. Touraille, L., M. K. Traore, and D. R. C. Hill. "A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models." *Proceedings of the 2009 ACM/SCS Spring Simulation Multiconference, SpringSim 2009*, March 22–27, San Diego, CA, 2009.
39. Al-Zoubi, K., and G. Wainer. "Interfacing and Coordination for a DEVS Simulation Protocol Standard." *The 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, British Columbia, Canada. New York: IEEE, 2008.
40. Al-Zoubi, K., and G. Wainer. "Using REST Web Services Architecture for Distributed Simulation." *Proceedings of Principles of Advanced and Distributed Simulation PADS 2009*, Lake Placid, New York, 2009.

17 Standardizing DEVS Model Representation

*Gabriel A. Wainer, Khaldoon Al-Zoubi,
David R.C. Hill, Saurabh Mittal, José L. Risco
Martín, Hessam Sarjoughian, Luc Touraille,
Mamadou K. Traoré, and Bernard P. Zeigler*

CONTENTS

17.1	Introduction	427
17.2	DEVSMML	428
17.2.1	DEVSMML Structure	429
17.2.2	DEVSMML Behavior	430
17.2.3	The DEVSMML Application	433
17.3	DEVSMML/SOA	434
17.3.1	DEVSMML/SOA Software Architecture	436
17.3.2	DEVSMML/SOA Modeling Layer	437
17.4	DML: DEVSMML Markup Language	439
17.4.1	DEVSMML Structure	440
17.4.2	DEVSMML Behavior	441
17.4.2.1	Parameterization and Initialization	442
17.4.2.2	Model Behavior	443
17.5	DCD++	445
17.5.1	RESTful Interoperability Simulation Environment (RISE)	445
17.5.2	Modeling Using the Distributed DEVSMML Simulation Protocol	449
17.5.3	CD++ DEVSMML Model Structure and Behavior	451
17.6	CoSMoS	452
17.7	Summary	456
	References	457

17.1 INTRODUCTION

As discussed in the previous chapter, the idea of *Standardizing Discrete Event System Specification (DEVSMML) model representation* is to allow a platform-independent DEVSMML model representation to be executed by a DEVSMML-based simulator. In this case, a DEVSMML model is executed in a single processor or parallel/distributed simulation environment. This allows model reusability without the need for performing long-distance distributed simulation. Typically, models are stored in repositories and retrieved as

needed. This type of interoperability is very important to achieve, because often modelers already have or can easily install a DEVS simulator on their machine. However, it is much more difficult for them to retrieve and reuse already existing models that were intended to run on a specific DEVS environment. Therefore, we need some way to provide access and share these numerous models. Having a platform independent DEVS model representation format allows us to go a step further in automatic model transformations and generate entire local copies of distant models.

Using transformations based on code parsing, we can generate the XML description of existing models written for a specific framework. These files can be stored in *model repositories*, accessible through the Internet, so that anyone (with the proper authorizations) can access them [1]. When facing a problem to which modeling and simulation (M&S) will be applied, the modeler starts by searching for an existing model that could fulfill the requirements. If there is none, the modeler can write their own but still take advantage of existing models by reusing them in their coupled models. To do so, the users download a model description from a repository in their framework, and then they use generic transformations to generate the code for the model. After this, they simulate their models and those of other users on their own simulation platform.

Compared to distributed simulation interoperability, this approach has the benefit that it drastically reduces the communications with remote servers. In fact, after the model description has been downloaded, there is no more need for accessing the network. However, the drawback of this is that a given model can have several unsynchronized versions all over the world. A solution to this issue could be to use some sort of update feed, but this is still more complicated than simply invoking a service without caring about the underlying implementation, which can continuously improve and evolve. Simultaneously, knowing the details about the implementation can provide insight into the structure of the model (which is an important factor for model-based publishing) [2].

To sum up both interoperability solutions exposed here, we can say that the simulator-based approach (i.e., via distributed simulation middleware, as discussed in Chapter 18) is aimed at coupling distributed or local models over remote servers, whereas the model-based approach aims at integrating distributed models in a local simulation.

Although the DEVS mathematical formulation is universal and well known, representing complex models with DEVS is still a difficult task (and it is not easy to understand). Consequently, there are a number of alternatives to represent DEVS models with different notations and tool support. In the following sections, we will introduce model-based efforts by different M&S groups around the world that have tackled the problem of standardizing DEVS models and their simulators.

17.2 DEVSMML

The DEVSMML is a novel way of writing DEVS models in XML. DEVSMML is built on JAVAMML (an XML implementation of JAVA). JAVAMML is used to specify the behavioral logic of atomic models and the structure of coupled models. DEVSMML models are transformable back and forth between Java and DEVSMML, with the

objective to provide interoperability between various models and to reuse and create dynamic scenarios.

The layered architecture of DEVSML is shown in Figure 17.1. A DEVS model can be provided as a set of platform-specific components, as a set of DEVSML components, or as a combination of both (see layer 1 in the figure). Next, the final composed model can be transformed into a full DEVSML representation (layers 2 and 3). To do so, the user must provide the platform from which their model originates (e.g. DEVSJAVA, CD++, etc.). Finally, using layer 4, the DEVSML model can be transformed to any of the compatible DEVS platforms (layer 5).

17.2.1 DEVS STRUCTURE

To ensure that the final DEVSML complex model is “DEVS compliant,” a single atomic *Document Type Definition* (DTD) and a single coupled DTD are defined. Thus, every DEVSML atomic or coupled model may be validated against the DEVSML DTDs. Both DTDs are listed in Figures 17.2 and 17.3.

Both DTDs define the structure and behavior of a DEVS model. As Figures 17.2 and 17.3 show, both atomic and coupled components are formed by standard DEVS structural definitions. In the following code snippet (Figure 17.4), we show how the structure of a Lorentz Attractor coupled model may be defined. Note that typically, a DEVS coupled model consists of structural definitions, so no JavaML code is needed in the transformation from Java to DEVSML.

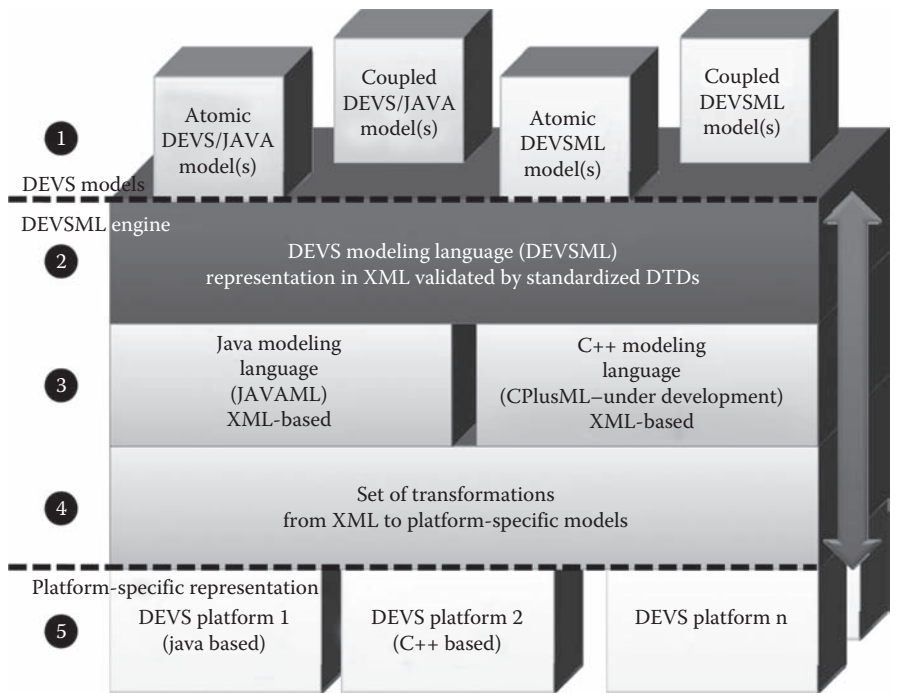


FIGURE 17.1 DEVSML layered architecture.

```

<!-- DEVS ATOMIC MODEL -->
<!ENTITY % variable-info
    "name CDATA #REQUIRED
    type CDATA #REQUIRED">
<!ELEMENT atomic (inputs,outputs,states,ta,deltint,delttext,deltcon,
lambda,services?,java-specific?)>
<!ATTLIST atomic
    name ID #REQUIRED
    simulator (devsjava|xdevs) #REQUIRED
    host CDATA #REQUIRED>
<!ELEMENT inputs (port*)>
<!ELEMENT port EMPTY>
<!ATTLIST port
    name CDATA #REQUIRED>
<!ELEMENT states (state*)>
<!ELEMENT state EMPTY>
<!ATTLIST state
    %variable-info;>
<!ELEMENT outputs (port*)>
<!ELEMENT ta (block*)>
<!ELEMENT deltint (block*)>
<!ELEMENT delttext (block*)>
<!ELEMENT deltcon (block*)>
<!ELEMENT lambda (block*)>
<!ELEMENT services (service*)>
<!ELEMENT service (method)>
<!ATTLIST service
    name ID #REQUIRED
    port CDATA #REQUIRED>
<!ELEMENT java-specific (package-decl,import*,constructor*,method*)>
<!ELEMENT import EMPTY>

```

FIGURE 17.2 DEVSML atomic DTD.

The structure definition of both atomic and coupled DEVS components is easy to describe in XML following the DTD shown in Figures 17.2 and 17.3.

17.2.2 DEVS BEHAVIOR

The behavior of a DEVS model is limited to atomic components; that is, the body of the transition and output functions as well as the time advance function. When the behavior of an atomic DEVS component is generated, all this code is transformed to JavaML. However, platform-specific sentences are transformed to platform-independent ones, extending the JavaML specification with new tags such as *platform-specific*, *state-assignment*, *port-read*, etc. For example, if an atomic component contains DEVJSJAVA code such as:

```

void deltint() {
    holdIn("standby", 3.5); // DEVJSJAVA library function
    call
}

```

```

<!-- DEVS COUPLED MODEL -->
<!ENTITY % connection-info
    "component_from CDATA #REQUIRED
    port_from CDATA #REQUIRED
    component_to CDATA #REQUIRED
    port_to CDATA #REQUIRED">
<!ELEMENT devs (scenario,models)>
<!ELEMENT scenario (coupled)>
<!ELEMENT coupled (inputs,outputs,components,internal_
connections,external_input_connections,
external_output_connections,java-source-program)>
<!ATTLIST coupled
    name ID #REQUIRED
    model CDATA #REQUIRED
    simulator (devsjava|xdevs) #REQUIRED
    host CDATA #REQUIRED>
<!ELEMENT inputs (port*)>
<!ELEMENT port EMPTY>
<!ATTLIST port
    name CDATA #REQUIRED>
<!ELEMENT outputs (port*)>
<!ELEMENT components (coupledRef|atomicRef)*>
<!ELEMENT coupledRef (components?)>
<!ATTLIST coupledRef
    name CDATA #REQUIRED
    model CDATA #REQUIRED
    simulator (devsjava|xdevs) #IMPLIED
    host CDATA #REQUIRED>
<!ELEMENT atomicRef EMPTY>
<!ATTLIST atomicRef
    name CDATA #REQUIRED
    model CDATA #REQUIRED
    simulator (devsjava|xdevs) #IMPLIED
    host CDATA #REQUIRED>
<!ELEMENT internal_connections (connection*)>
<!ELEMENT external_input_connections (connection*)>
<!ELEMENT external_output_connections (connection*)>
<!ELEMENT connection EMPTY>
<!ATTLIST connection
    %connection-info;>

<!ELEMENT models (model*)>
<!ELEMENT model (java-source-program)>
<!ATTLIST model
    name ID #REQUIRED>

```

FIGURE 17.3 DEVSML coupled DTD.

It is transformed to DEVSML using the following extension of JavaML:

```

<deltint>
<block>
<platform-specific>
  <state-assignment op = " = ">

```

```

<devs>
  <scenario>
    <coupled name = "root" model = "LorentzAttractor" simulator =
      "xdevs" host = "127.0.0.1">
      <inputs/>
      <outputs/>
      <components>
        <atomicRef name = "fxu" model = "Function" simulator = "xdevs"
          host = "127.0.0.1"/>
        <atomicRef name = "integrator" model = "Integrator"
          simulator = "xdevs" host = "127.0.0.1"/>
        <atomicRef name = "gxu" model = "Function" simulator = "xdevs"
          host = "127.0.0.1"/>
        <atomicRef name = "scope" model = "Scope" simulator = "xdevs"
          host = "127.0.0.1"/>
      </components>
      <internal_connections>
        <connection component_from = "fxu" port_from = "out"
          component_to = "integrator" port_to = "in"/>
        <connection component_from = "integrator" port_from = "out"
          component_to = "fxu" port_to = "x"/>
      ...
    </coupled>
  </scenario>
</devs>

```

FIGURE 17.4 Code snippet for a Lorentz Attractor coupled model.

```

<lvalue>
  <var-ref name = "phase"/>
</lvalue>
<literal-string value = "standby"/>
</state-assignment>
<state-assignment op = " = ">
<lvalue>
  <var-ref name = "sigma"/>
</lvalue>
<literal-number kind = "float" value = "3.5"/>
</state-assignment>
</platform-specific>
</block>
</deltint>

```

Using these special tags, a DEVSMML model can be transformed to any supported DEVS simulation engine. The main disadvantage of using this methodology is the portability between different programming languages (for example, between Java and C++). Every programming language has its native as well as external libraries to simplify applications code (e.g., containers, numeric libraries, etc.). With DEVSMML, transformations between DEVS simulation engines developed in different programming languages would be possible only if no specialized functions or libraries were used. However, the DEVSMML reverse engineering provides full interoperability between models developed in the same programming language, which is a great advantage when models from different repositories are used to compose bigger

coupled models using DEVSMML seamless integration capabilities. Furthermore, all the DEVSMML files generated can be stored in a Library for reuse. The composed integrated model is complete in every respect as it contains behavior and is ready for simulation. Based on the information contained in the DEVSMML model description, the corresponding simulator is called; after the simulator has instantiated the model, the model executes with the designated simulator. DEVSMML provides new concepts for easily integrating reverse engineering into a DEVS/XML standard providing both interoperability and reuse. Note that in this example two DTDs were created instead of two XML Schemas (the reason is that JavaMML is defined in terms of DTDs; the DTD definition is included in the DEVS/XML standard).

17.2.3 THE DEVSMML APPLICATION

Figure 17.5 shows the Java application that implements the DEVSMML principles. It contains two simulators, namely, xDEVs and DEVsJAVA, demonstrating the validation of DEVSMML atomic and coupled models with the same Atomic and Coupled DTDs. It converts any atomic/coupled model from their Java implementation to DEVSMML and vice versa. It also validates any DEVSMML model description and integrates any coupled DEVSMML description into a composite DEVSMML coupled model ready to be simulated with the target simulator. The tool also generates a Java code library from a composite DEVSMML coupled model. It contains various web services in operation. The five web services that are publicly offered are: (1) convert Java model to DEVSMML,

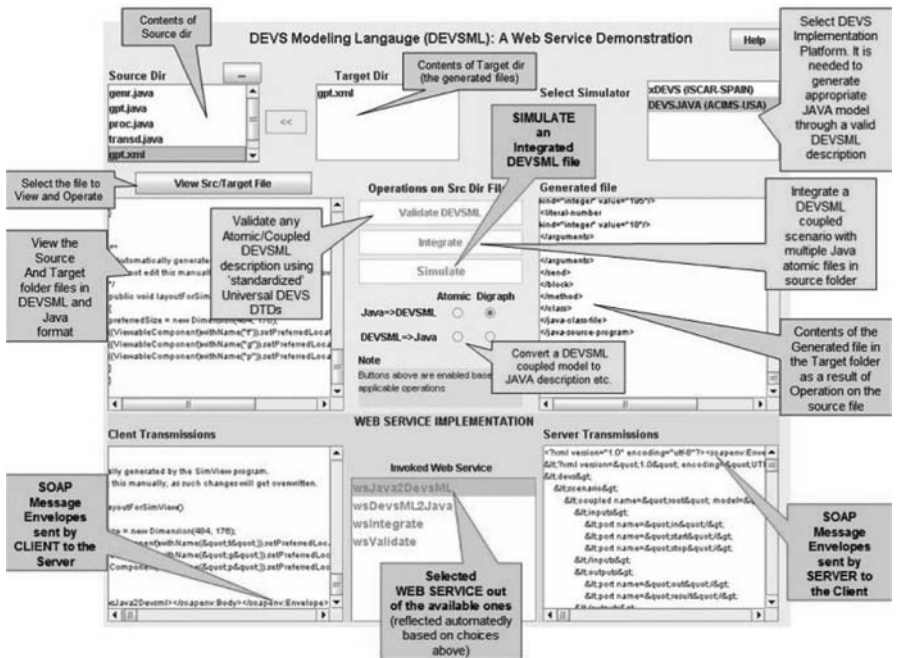


FIGURE 17.5 Client application snapshot implemented as an applet.

(2) convert DEVSMML to Java code, (3) validate the existing DEVSMML model, (4) integrate coupled and atomic DEVSMML models toward a portable “Composite” Coupled DEVSMML file that is simulatable at any remote server, and (5) simulate the Composite Coupled file and send console messages at the Server to the Client window providing evidence that the simulation is running. The users can select their own Source and Target directories and can choose their implementation; that is, Java code and Simulator compatibility. The Server application checks for compatibility as well.

To conclude, DEVSMML partially solves the interoperability in DEVS/XML, and fully supports the reuse between different simulation platforms by means of reverse engineering.

17.3 DEVS/SOA

This section presents DEVS/SOA in detail. We start by introducing how DEVS/SOA works in general terms, from a user point of view. Next, we present an overview and the software architecture of DEVS/SOA [3] and its implementation using web standards. Finally, we show how the DEVS/SOA framework is able to integrate several DEVS and non-DEVS models.

As stated before, there are many libraries for expressing DEVS models. All have efficient implementations for executing the DEVS protocol and provide advantages of Object-Oriented frameworks such as encapsulation, inheritance, and polymorphism. In order to simplify notation, we use the term DEVS/JAVA to denote a DEVS library implemented in Java (for example, DEVSJAV or xDEV) and DEVS/NET to denote a DEVS library implemented in any language supported by .NET (for example, aDEV in C, CD++ in C++, and DEVS.NET in C#).

First of all, DEVS/SOA is another DEVS simulation platform. Implemented in both Java and .NET programming environments, DEVS/SOA includes a modeling library as well as a simulation library. The main difference with respect to other DEVS simulation libraries is that the modeling package (including both atomic and coupled classes) are “adapters” of other DEVS simulation engines. For instance, as seen in Section 17.2.2, a DEVS/SOA atomic component can be written in DEVSJAV or xDEV (as well as its own DEVS/SOA native atomic component). Thus, DEVS/SOA allows interoperability among different DEVS simulation engines using diverse web services infrastructures (Figures 17.6 and 17.7).

The same situation happens in the DEVS/SOA .NET infrastructure. The “magic” of interoperability between DEVS/SOA Java and DEVS/SOA .NET resides in the message-passing interface. As shown in Section 16.2.4.2, when a simulator must send a message to another one, all the objects (or entities) included in the message are passed from one simulator to the other in SOAP messages. To this end, some intermediate libraries such as JAXB in Java and native .NET transformations are used to transform an object representation to its XML equivalent (both at the model level, not at the simulator level). Managing messages in XML format makes possible the publication of web services, giving DEVS/SOA the ability to publish simulators (and even atomic or coupled models) as web services.

How to compose a DEVS/SOA model? As stated before, each DEVS M&S library is developed using a particular programming language (Java, C, C++, C#, etc.). Since



FIGURE 17.6 Heterogeneous DEVS/SOA model.

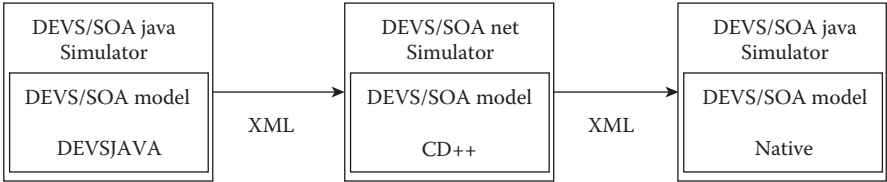


FIGURE 17.7 Interoperability in DEVS/SOA.

DEVS/SOA provides interoperability, the composition of the desired DEVS model at the client application level is completed using a platform independent mechanism. Therefore, our approach describes DEVS coupled models exploiting the XML description format. Figure 17.8 shows the DEVSMML representation of a DEVS coupled model including an Experimental Frame and a Processor [12] using three different platforms: xDEVJS, DEVJSJAVA, and DEVJS.NET. The DEVS/XML notation used to define the structure of DEVS/SOA distributed and interoperable models is based on DEVSMML (see Section 17.2.2). The root coupled element determines the root model identification and the localization of the DEVS web coordination service. The children nodes of the root element represent DEVS atomic models, connections among DEVS models, or other DEVS coupled models (in a recurrent manner). The data included for each atomic model item are: its name, the platform where the model can be simulated, the class that implements the model that must be instantiated, the localization of the DEVS web simulation service that has access to the model implementation class repository, and finally, the names of the input and output ports of a model as well as the class types of their messages (as internal elements). The connection items designate links between a model output and a model input port.

Summarizing, from a user's perspective, the use of DEVS/SOA M&S implies:

- Compose a DEVS model according to the localization of the original submodels
- Write a DEVSMML file as a result of the modeling phase
- Partition the composed model either manually or through the DEVS/SOA automated mechanism
- Deploy the models
- Run the distributed simulation
- Wait for the results

The main advantage of DEVS/SOA with respect to other distributed M&S methods is that the original models remain encoded in the native programming language related to the selected DEVS library. Therefore, from a user point of view, no middleware

```

<?xml version="1.0" encoding="UTF-8"?>
<coupled name="efp" host="http://192.168.1.2:8080/axis2/services/Coordinator">
<coupled name="ef" platform="devsjava" class="Ef" host="http://192.168.1.4:8080/axis2/services/Simulator">
<atomic name="generator" platform="xdevs" class="Generator" host="http://192.168.1.4:8080/axis2/services/Simulator">
<inport name="stop" class="Job"/>
<outport name="out" class="Job"/>
</atomic>
<atomic name="transducer" platform="xdevs" class="Transducer" host="http://192.168.1.4:8080/axis2/services/Simulator">
<inport name="solved" class="Job"/>
<inport name="arrived" class="Job"/>
<outport name="out" class="Job"/>
</atomic>
<inport name="in" class="Job"/>
<outport name="out" class="Job"/>
<connection atomicFrom="ef" portFrom="in" atomicTo="transducer" portTo="solved"/>
<connection atomicFrom="generator" portFrom="out" atomicTo="ef" portTo="out"/>
<connection atomicFrom="generator" portFrom="out" atomicTo="transducer" portTo="arrived"/>
<connection atomicFrom="transducer" portFrom="out" atomicTo="generator" portTo="stop"/>
</coupled>
<atomic name="processor" platform="devs.net" class="Processor" host="http://192.168.1.3:3158/Simulator.asmx">
<inport name="in" class="Job"/>
<outport name="out" class="Job"/>
</atomic>
<connection atomicFrom="ef" portFrom="out" atomicTo="processor" portTo="in"/>
<connection atomicFrom="processor" portFrom="out" atomicTo="ef" portTo="in"/>
</coupled>

```

FIGURE 17.8 EFP DEVS/SOA model structure (DEVSMML).

is needed (although of course DEVS/SOA is built using web services technologies), and, most important, the software engineer does not need special distributed programming skills.

17.3.1 DEVS/SOA SOFTWARE ARCHITECTURE

In DEVS/SOA, an implementation of the DEVS formalism is developed within SOA so as to provide DEVS M&S services over the WWW. The DEVS/SOA allows distributing simulations over multiple processors in the same computer, over the Internet, or over both. A DEVS coupled model is split into different submodels that run on multiple computers in a distributed manner. For this purpose, the DEVS/SOA coordinator deals with simulators hosted anywhere on the web. Every DEVS/SOA simulator is responsible for managing its corresponding DEVS atomic model. Communications between simulators and coordinators are performed using standard technologies. As a result, DEVS models can be composed in a transparent, open, and scalable way. Another advantage of this architecture is the interoperability among DEVS submodels. Since the communication among simulators is performed in a standard way such as XML, the associated DEVS submodels can be implemented using different DEVS simulation engines.

The proposed DEVS/SOA architecture, whose basics are captured in Figure 17.9, embraces three levels of interoperability between DEVS M&S elements. The outer level comprehends the interaction between the client application layer and the DEVS coordinator service. It is implemented using the SOAP communication protocol

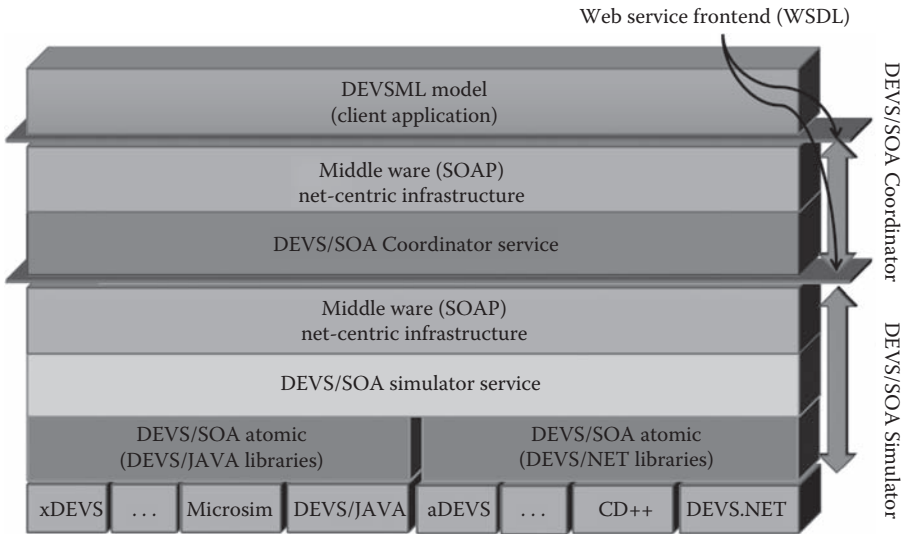


FIGURE 17.9 DEVS/SOA architecture.

according to the operations detailed at the Coordinator WSDL file. Next, a similar interoperability level is encountered as the communication between the DEVS coordinator and simulation services. Again, the SOAP protocol specification is applied for exchanging structured information between both types of services. The operations offered by the Simulator service are encoded in the Simulator WSDL file. Finally, the third level is at the junction of a DEVS simulator with its associated native DEVS model, responsible for implementing the DEVS/SOA atomic models. The messages among different native DEVS models are expressed in XML. In the following, these three levels (modeling, simulation, and client application) are described in detail.

17.3.2 DEVS/SOA MODELING LAYER

Figure 17.10 shows excerpts of the DEVS/SOA atomic interface, base class, and the DEVSJAVA atomic adapter. The former follows a classic DEVS atomic structure. In addition, the given interface includes functions to receive input data in XML and to obtain the output in XML (both implemented in the DEVS/SOA atomic model). The DEVS/SOA atomic model (class AtomicBase) implements the previous interface, including functions `getOutput` and `receive`, which send and receive data in XML. Finally, the DEVSJAVA adapter inherits from this DEVS/SOA JAVA atomic model and implements the DEVS/SOA atomic interface. By using these classes, heterogeneous DEVS/SOA and DEVSJAVA models can coexist in a simulation. Thus, to allow interoperability with other DEVS simulation platforms, a new adapter (only one class per simulation framework) must be implemented.

```

public interface AtomicInterface {
    // DEVS protocol
    void initialize();
    double ta();
    void deltint();
    void deltext(double e);
    void deltcon(double e);
    void lambda();
    // Source code is omitted for brevity ...
    Ports getInput();
    Ports getOutput();
    String[] getOutput(String portName); // Get values at output port in XML
    void receive(String portTo, String[] xmlValue) throws Exception;
    // Receive input data in XML
}

public class AtomicBase implements AtomicInterface {
    // Source code is omitted for brevity ...
    public String[] getOutput(String portName) {
        Port port = output.get(portName);
        Set<Object> values = port.getValues();
        String[] xmlValues = new String[values.size()];
        int i = 0;
        for(Object value : values) {
            StringWriter writer = new StringWriter();
            Result result = new StreamResult(writer);
            JAXB.marshal(value, result);
            String xmlValue = writer.toString();
            xmlValues[i++] = xmlValue;
        }
        return xmlValues;
    }

    public void receive(String portTo, String[] xmlValues) throws Exception {
        Port port = this.getInput().get(portTo);
        for(int i=0; i<xmlValues.length; i++) {
            if(xmlValues[i]!=null) {
                StringReader reader = new StringReader(xmlValues[i]);
                Object value = JAXB.unmarshal(reader, Class.forName(port.
                    getPortClass()));
                this.getInput().get(portTo).addValue(value);
            }
        }
    }
    // Source code is omitted for brevity ...
}

```

FIGURE 17.10 DEVS/SOA JAVA atomic interface, base class, and the DEVJSOA adapter.

```

public class AtomicDevsJava extends AtomicBase implements AtomicInterface {
    private genDevs.modeling.atomic model;
    public AtomicDevsJava(Element xmlAtomic) throws Exception { // Constructor
        super(xmlAtomic);
        // It processes the corresponding portion of the input DEVSMML file ...
    }
    // Source code is omitted for brevity ...
    public void deltext(double e) { // external transition function
        genDevs.modeling.message msg = buildMessage();
        model.deltext(e, msg);
    }
    public void lambda() {
        genDevs.modeling.message msg = model.out();
        genDevs.modeling.ContentIteratorInterface itr = msg.mIterator();
        while(itr.hasNext()) {
            genDevs.modeling.ContentInterface devsJavaPort = itr.next();
            Port port = output.get(devsJavaPort.getPortName());
            if(port!=null)
                port.addValue(devsJavaPort.getValue());
        }
    }
    // Source code is omitted for brevity ...
    public genDevs.modeling.message buildMessage() {
        genDevs.modeling.message msg = new genDevs.modeling.message();
        Map<String, Port> ports = input.ports;
        Iterator<String> itr = ports.keySet().iterator();
        while(itr.hasNext()) {
            String portName = itr.next();
            Set<Object> values = ports.get(portName).getValues();
            for(Object value : values) {
                genDevs.modeling.content con = model.makeContent(portName,
(entity)value);
                msg.add(con);
            }
        }
        return msg;
    }
}

```

FIGURE 17.10 (Continued) DEVS/SOA JAVA atomic interface, base class, and the DEVJSJAVA adapter.

17.4 DML: DEVS MARKUP LANGUAGE

The DEVS Markup Language DML [4] is a design that standardizes the representation of DEVS models (which can be defined in different DEVS platforms), and at the same time is as generic and flexible as possible. The idea is to use an XML representation of a specific programming language, for example the Java Markup Language, as done in DEVSMML. This approach has the advantage that many tools are available to manipulate and transform the source code and the XML representation. However,

it induces a strong coupling between the model and a particular language, therefore restricting its deployment scope.

To enlarge this scope, a more generic representation could be employed, such as Metal [5] or Object-Oriented XML (O2XML) [6]. These markup languages enable a representation of most object-oriented programming languages by specifying a common denominator between them. Thereby, an XML document can be transformed into source code in any object-oriented target language. Though this solution seems attractive, it prevents the modeler from using certain language features or libraries (e.g., pseudorandom numbers generation with the Stochastic Simulation library [7]; matrix operations using Boost uBLAS, etc.). Consequently, such a high-level language with a small set of features would reduce the universality of the standard, and make the representation of legacy models very difficult as they are usually written in a specific programming language and use advanced language functionalities and libraries.

To resolve this issue, one should let the modelers use their favorite language and delegate the task of interpreting them to simulators or proxies. That would imply developing an architecture in which a model is composed of several modules in multiple languages, this model being itself simulated by an application possibly written in yet another language. Even though this architecture is conceivable (using technologies such as JNI, RMI, CORBA, and Web Services), the complexity and the loss of performance would be high.

DML tries to reach a compromise between generality and flexibility, by defining an XML representation of source code.

17.4.1 DEVS STRUCTURE

Describing the structure of DEVS models is done by translating the set of formalism into an XML representation. Input and output ports, parameters, and state variables are characterized by a name and a type. To allow the deployment of the model on any simulator, this type should be as generic as possible. However, in real-world applications, the modeler may need to use certain types defined in particular language libraries. To satisfy these apparently contradictory requirements, three different kinds of types must be considered:

- Intrinsic types, which are natively supported by all object-oriented languages (e.g., integer, string, float, etc.).
- Custom types, which are types that are defined by the modeler in a language independent manner.
- Language dependent types, which have to be bound to a particular type, for each targeted languages. This notion, close to the one of Abstract Data Types, lets the modeler conceive algorithms using abstract types that must be matched to library or language types in order to generate the source code on the target platform.

Figure 17.11 shows an example of the use of these different types. In this example, the state variable *sigma* is a floating-point number, whereas *phase* is a more complex

```

<state>
  <variable>
    <name>phase</name>
    <type kind="custom">procPhase</type>
  </variable>
  <variable>
    <name>buffer</name>
    <type kind="languageDependent">Queue</type>
  </variable>
  <variable>
    <name>sigma</name>
    <type>double</type>
  </variable>
</state>
...
<type id="procPhase" xsi:type="enum">
  <value>busy</value>
  <value>idle</value>
</type>
<type id="Queue" xsi:type="languageDependent">
  <implementation lang="java">
    java.util.LinkedList<String>
  </implementation>
  <implementation lang="c++">
    std::queue<std::string>
  </implementation>
</type>
</state>

```

FIGURE 17.11 Sample of XML representation of state variables.

type; namely, an enumeration with two possible values: *busy* or *idle*. Finally, the variable *queue* has an abstract type, which we bind to specific types in Java and C++. Therefore, to make this model deployable on a simulator written in another language, the modeler has to specify what type can be used as a queue in this language. We can also imagine having repositories of abstract types along with their bindings in numerous programming languages, to allow their reusability in several models.

The XML Schema for coupled DEVS models includes ports, components, and coupling declarations, as well as a tie-breaking function in the case of classic DEVS. Components are described by their name and the type of model they instantiate. The couplings are pairs of ports, if necessary qualified by their component's name. Finally, the *Select* function is described by ordered sets indicating priorities among components.

17.4.2 DEVS BEHAVIOR

One of the main issues in standardizing behavior of atomic DEVS models boils down to the following question: How can we describe application logic in a language-independent manner? Several efforts have been directed toward providing an XML representation

of source code. Some of them are oriented toward a sole programming language, for example, JAVAML [8], PascalML [9], or CppML [10], while others propose vocabularies that contain a common denominator between object-oriented languages [6,10].

None of these frameworks is generic or open to extension with language dependent features. DML proposes a compromise between both approaches, by drawing inspiration from the concept of weaving used in Aspect Oriented Programming [11], or more simply from the C and C++ macros. The main concept is to use a pseudolanguage such as O2XML so that the major part of the code can be reused across different simulators, while supporting the possibility to include language specific code snippets. This concept is illustrated in Figure 17.12, where generic programming constructs are represented in XML, and the use of libraries is abstracted and tied to the specific implementations in several languages. This semigeneric language can also be used to create custom types (i.e., classes, enumerations), which become usable in the variables and ports declarations. This approach has two advantages:

- It allows the use of programming language features and libraries.
- It permits the representation of legacy models written for a specific simulator.

By limiting the language-dependent code snippets to a minimum, the migration of models between platforms becomes easier, and thereby their portability and reusability are enhanced.

17.4.2.1 Parameterization and Initialization

The above description of DEVS models is not sufficient to produce an executable specification. Indeed, models can have parameters, which need to be set prior to simulation, and their initial state must be specified in some way.

Figure 17.13 depicts how a model, regarded as a template, can be instantiated into several parameterized models depending on the values assigned to each parameter. After that, the state must be initialized to obtain a ready-to-simulate model. A simulator can then be fed with this executable model, along with an input trajectory if need be, and produce the output of the simulation.

To perform these operations, two approaches are possible depending on the degree of freedom required. The simplest is to restrict parameterization and initialization to pairs of variable identifiers and values. However, these operations can sometimes be more complex than mere assignments (e.g., for file manipulation or random number generation). Therefore, in DML these two activities are described in a similar fashion as the model dynamics, using the source code representation outlined in the previous paragraph.

Consequently, the modeler needs to define two functions for each atomic model in the hierarchy: one to set the parameters values, and the other one to initialize the state. Ideally, these functions should be kept separate from the model definition, so that it can be easily reused with different scenarios.

In order to simulate a DEVS model, four specifications are needed: structure, dynamics, parameterization, and initialization. Given these specifications and, if



FIGURE 17.12 Representation of model dynamics using a semi-generic language.

required, an input trajectory, a simulator can then produce results representing the model behavior.

17.4.2.2 Model Behavior

The behavior of a model is normally fully represented by its output trajectory. However, it is common to desire its state trajectory and, in the case of coupled models, the behavior of its components as well. These two aspects can be described

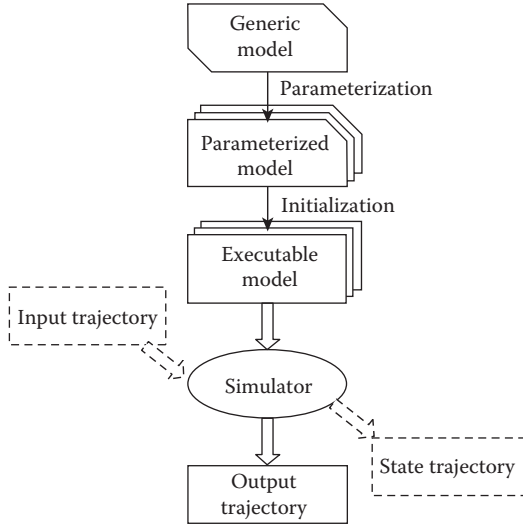


FIGURE 17.13 Parameterization and initialization of a model.

```

<behavior id="pipeSimple">
  <state>
    <snapshot time="0">
      <component id="p0">
        <phase>idle</phase>
        <queue />
        <sigma>infinity</sigma>
      </component>
      <!-- other components -->
    </snapshot>
    <snapshot time="3">
      <component id="p0">
        <phase>busy</phase>
        <queue>
          <elem>job1</elem>
        </queue>
        <sigma>10</sigma>
      </component>
      <!-- and so on -->
    
```

FIGURE 17.14 Sample state trajectory.

by sets of values annotated with a time stamp. Figure 17.14 shows a sample state trajectory of a coupled model, namely the *pipeSimple* defined in Zeigler, Kim, and Praehofer [12].

The XML schema for this representation of a trajectory can be deduced from the model structure: each variable or port becomes an XML tag accepting a value in

accordance with its type. Another solution would be to have a tag *variable* with an attribute containing the name of the variable.

In the case of variables of a complex type, the value at a given time can be obtained by serializing the object from memory, as is done when exchanging data between web services. This supposes providing an XML Schema describing the structure of the object, for later use (e.g., in a visualization tool). For custom types, this schema can be deduced from the type definition made in the model specification; for language dependent types, the schema can often be automatically generated from the source code.

This XML representation of trajectories has the valuable advantage that it can be easily processed by a machine while remaining human readable. However, because of the potentially huge size of data generated, it may be necessary to switch to a more compact representation, such as the eXternal Data Representation standard [13].

17.5 DCD++

Distributed CD++ (DCD++) [22] supports simulation over SOA in two versions: SOAP-based Web-service [26,14] and RESTful Web-services [22,23,24]. The Distributed DEVS Simulation Protocol (described in Section 17.5.2) is based on SOAP DCD++ whereas RISE (described in Section 17.5.1) is based on RESTful DCD++. Both versions provide modelers with the complete required cycle to run a simulation experiment. This includes submitting a model to be simulated (i.e., usually partitioned over different machines) and retrieving simulation results. Both versions decouple model representations from simulation, hence different simulation engines can collaborate in the same simulation run. This enables different model representations to participate in the same simulation run, hence allowing DEVS simulation engines to execute legacy models and standardized model representations. In this section, we discuss the modeling structure and partitions for both approaches followed by DEVS model behaviors.

17.5.1 RESTful INTEROPERABILITY SIMULATION ENVIRONMENT (RISE)

RESTful Interoperability Simulation Environment (RISE) provides lightweight, thin middleware based on RESTful Web-services. RISE is independent of any simulation environment and provides interoperability at three levels. First, the *interoperability framework architecture* level provides the URI templates API that allows modelers to create simulation/modeling environment (including modeling configuration, distributing simulations, starting simulation, and retrieving results). Second, the *model interoperability* level provides XML rules for binding different models together. Third, the *simulation synchronization* level provides high-level simulation algorithms and synchronization channels. The first and third levels are discussed in Chapter 18 whereas the second level is discussed in this section.

The RISE Experiment framework is a URI created by the modeler (see URI templates API in Chapter 18) to contain a domain, where a domain is a simulation environment that contains a model and simulation engine. RISE completely wraps a

domain interior, hence simplifying interoperability between heterogeneous domains. The RISE modeling interoperability level assumes an entire model is placed in a domain, hence a domain at this level can be viewed as a model wrapped within a URI (e.g., `.../myFireModel`), as shown in Figure 17.15. RISE treats all models (domains) as black boxes with input and output ports. In other words, each model views other heterogeneous models as part of its environment. It is worth noting that a model (within a domain) may be partitioned among different processors, simulated by conservative/optimistic algorithms according to DEVS/non-DEVS formalism. However, at the RISE layer the model is seen as a URI (which contains both the model and the simulation engine that simulates it).

Modelers need to configure the experiment frameworks by sending XML configuration documents via PUT channels to their URIs (e.g., `.../MyFireModel`). This represents model interconnections shown in Figure 17.15 in the XML document. Note that the model interconnections map is converted into routing tables by simulation engines (see Chapter 18), allowing them to route messages during simulation according to the model URI and message input port. Further, plug-and-play capability of RESTful Web-services takes this configuration to another level. This plug-and-play level allows modelers to overwrite model interconnection during simulation runtime, allowing simulation engines to create routing tables. Thus, models (without prior knowledge) may join or leave a simulation session at runtime. In such a case, the XML configuration is sent to the simulation engine URI (`.../MyFireModel/simulation`).

In Figure 17.15, Model-2 influences Model-1 via ports-2 and 3 of Model-2. Model-3 influences both Model-1 and Model-2 via Port -1 of Model-3. This implies that the output ports of a model can generate RISE external messages addressed to their connected input ports during simulation, thereby influencing the receiving models. Note that Port-1 and Port-3, of Model-3 and Port-4 of Model-1 are not used by RISE. These ports can still be used by modelers, for instance, to influence a model dynamically during active simulation. Figure 17.16 shows the XML configuration document for

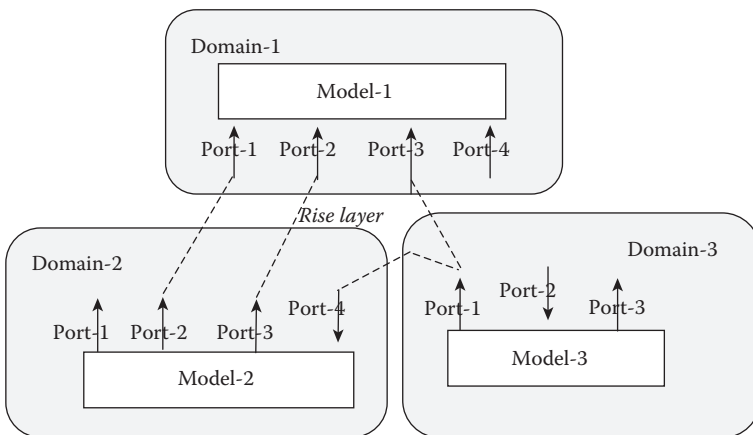


FIGURE 17.15 RISE domain models interconnections.

the example in Figure 17.15. This document needs to be submitted to each domain (typically constructed and submitted by client software). The RISE configuration must be the enclosed element <RISE> body (which starts on line 4 in Figure 17.16). The domains are free to define their own specific configuration outside the <RISE> block. Line 3 shows the RISE protocol version, and simulation type where “C” type stands for Conservative-based and “O” type for the Optimistic-based simulation. Lines 4–32 define domains (models) configurations. Line 5 defines the Main domain URI. The Main domain is needed during active simulation, discussed in Chapter 18. Lines 6–31 define RISE ports interconnections. For example, lines 7–12 define the connection from port-2 in URI <.../Domain-2> to port-1 in URI <.../Domain-1>.

```

1 <ConfigFramework>
2   ...
3   <RISE Version="1.0" Type="C">
4     <Domains>
5       <Main><URI>.../Domain-1</URI></Main>
6       <Links>
7         <Link>
8           <From><Port>Port-2</Port>
9           <URI>.../Domain-2</URI></From>
10          <TO><Port>Port-1</Port>
11          <URI>.../Domain-1</URI></TO>
12        </Link>
13        <Link>
14          <From><Port>Port-3</Port>
15          <URI>.../Domain-2</URI></From>
16          <TO><Port>Port-2</Port>
17          <URI>.../Domain-1</URI></TO>
18        </Link>
19        <Link>
20          <From><Port>Port-1</Port>
21          <URI>.../Domain-3</URI></From>
22          <TO><Port>Port-4</Port>
23          <URI>.../Domain-2</URI></TO>
24        </Link>
25        <Link>
26          <From><Port>Port-1</Port>
27          <URI>.../Domain-3</URI></From>
28          <TO><Port>Port-3</Port>
29          <URI>.../Domain-1</URI></TO>
30        </Link>
31      </Links>
32    </Domains>
33  </RISE>
34  ...
35 </ConfigFramework>

```

FIGURE 17.16 RISE model domain configuration.

RISE is independent of any modeling representation or simulation engines. For example, CD++ uses RISE to simulate different distributed model partitions. In this case, the CD++ domain is viewed as a single model that is executable by a CD++ engine. In the RESTful DCD++ (Section 18.5), the modeler partitions the DEVS or Cell-DEVS model among different RISE instances. Note that the CD++ interior domain is hidden behind a single URI, as discussed previously, even though it is using other RISE middleware instances to simulate its own domain model partitions. RISE in this case is a server with an IP address and port number, hence RISE is usually placed on a machine by itself. However, multiple instances of RISE may be placed on the same machine by assigning different port numbers, usually for testing purposes. For example, assume a modeler created experiment framework `<.../cdpp/sim/workspaces/Bob/DCDpp/Producer_Consumer>`. The RISE server then creates experiment *Producer_Consumer*. Workspace *Bob* and service *DCDpp* are created, if they do not already exist. *DCDpp* service indicates that the simulation is using the DCD++ engine to execute the model. Assume further that the modeler wants to place the atomic model “*Producer*” on a server while the atomic model “*Consumer*” is placed on a different server. In this case, the modeler needs to send the following partitioning (Figure 17.17) to the experiment URI.

Though it is easier for a modeler to partition a model in terms of an IP address and port number, the simulation is conducted among URIs (which allows multiple users and experiments to be performed simultaneously). Thus, the main server must convert the modeler documents into the format shown in Figure 17.18. In this case,

```
<DCDpp>
  <Servers>
    <Server IP="10.0.40.8" PORT="8080">
      <MODEL>Producer</MODEL></Server>
    <Server IP="10.0.40.9" PORT="8282">
      <MODEL>Consumer</MODEL></Server>
    </Servers>
  </DCDpp>
```

FIGURE 17.17 RESTful DCD++ model partitions by a modeler.

```
<DCDpp>
  <Servers>
    <Server ID="0" NAME="10_0_40_8_8080">
      <URI>.../workspaces/Bob/DCDpp/Producer_Consumer</URI>
      <MODEL>Producer</MODEL></Server>
    <Server ID="1" NAME="10_0_40_9_8282">
      <URI>.../workspaces/10_0_40_8_8080/DCDpp/Producer_Consumer_Bob/</
URI>
      <MODEL>Consumer</MODEL></Server>
    </Servers>
  </DCDpp>
```

FIGURE 17.18 RESTful DCD++ model partitions by the main server.

the main server assigns an ID to every server in the distributed simulation where “0” is its own ID. It further creates a name for each server (i.e., a combination of the server IP and port number). This name is the server user account on all other servers. This is needed because all messages during simulation are authenticated according to a user name and password. Furthermore, it assigns a URI for each model partition. The model partition at the main server is assigned the URI created by the modeler while other server partitions are assigned slightly modified URIs. This is because the main server needs to create experiment partitions on other supportive servers similar to other clients. This also ensures URI uniqueness on all servers, since supportive servers on an experiment may be used as main servers by other modelers for other experiments.

17.5.2 MODELING USING THE DISTRIBUTED DEVS SIMULATION PROTOCOL

In Al-Zoubi and Wainer [14], a flexible and scalable XML-based message-oriented mechanism was developed to allow interoperability between different DEVS implementations based on the SOAP-based DCD++ (Section 18.3) simulation package. This section focuses on the modeling part while simulation synchronization is described in Chapter 18. The interoperability is achievable with minimum design changes to each DEVS implementation, mainly by hiding the detailed implementation behind a *wrapper* (i.e., a SOAP port) and focusing only on the exchanged XML messages. This method assumes the simulation is conducted among different domains where a domain contains a DEVS model (with an engine capable of executing that model). The model in this case represents a complete reusable feature that a modeler wants to plug into an overall model hierarchy. The main idea behind modeling here is wrapping all models across domains within a single DEVS coupled model, which can be simulated by the main domain simulation engine while other domains only simulate their portion of the overall coupled model. For example, Coupled 1 and Coupled 2 are wrapped within Coupled 0, as shown in Figure 17.19. This information is described in the Model structure XML document.

The Model structure XML document (shown in Figure 17.20) is initially submitted by the modeler to the main DEVS domain to describe how the overall model is structured so that each DEVS version can identify which models belong to its

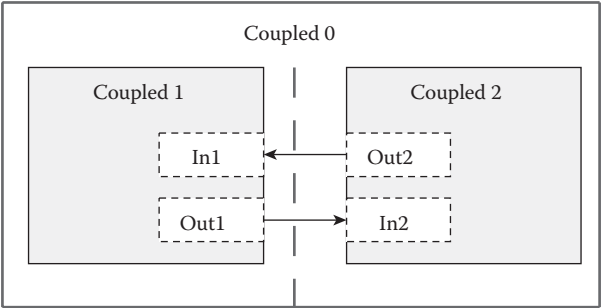


FIGURE 17.19 Coupled model partitioned across DEVS domains.

```

<MODEL_STRUCTURE ver="1.0">
  <COUPLED_SYNC>
    <scheme ver="1.0">HeadProxy</scheme>
  </COUPLED_SYNC>
  <Models>
    <Model Type="Coupled">
      <Name> Coupled0 </Name>
      <Components>
        <Name Type="Coupled">Coupled1</Name>
        <Name Type="Coupled">Coupled2</Name>
      </Components>
      <URI>http://... </URI>
      <LINKS>
        <LINK>
          <FROM>
            <Component>Coupled1</Component>
            <Port>OUT1</Port>
          </FROM>
          <TO>
            <Component>Coupled2</Component>
            <Port>IN2</Port>
          </TO>
        </LINK>
        ...
      </LINKS>
      ...
    </Model>
    <Model Type="Coupled">
      <Name> Coupled1 </Name>
      <Ports>
        <Port Type="in">IN1</Port>
        <Port Type="out">OUT1</Port>
      </Ports>
      <URI>http://... </URI>
      ...
    </Model>
    <Model Type="Coupled">
      <Name> Coupled2 </Name>
      <Ports>
        <Port Type="in">IN2</Port>
        <Port Type="out">OUT2</Port>
      </Ports>
      <URI>http://... </URI>
      ...
    </Model>
  </Models>
  ...
</MODEL_STRUCTURE>

```

FIGURE 17.20 XML model structure document example.

domain. Further, from this document, the main domain can identify the participant supportive domains. The model structure document contains enough information to allow different domains to create local models, coordinators (i.e., coupled model processor), and simulators (i.e., atomic model processor). It also includes data on how they will relate to other models in different domains. The main DEVS domain must pass this document before it starts the simulation. The model structure document contains the following information (Figure 17.20):

- Model names
- Model type (coupled/atomic)
- Model input/output ports
- Coupled models internal submodels and their ports connections
- Models domain URIs
- Coupled models synchronization algorithms used (e.g., the Head/Proxy Coordinator discussed in Chapter 18)

The DEVS models hierarchy can easily be mapped into this XML document. For example, assume two models connected with each other as in Figure 17.19 (two DEVS domains where each model is specific to its domain implementation). In this case, the two models would be enclosed within an outer model (Coupled 0), resulting in the XML document shown in Figure 17.22. This XML document also serves as an agreement contract between various implementations on the used synchronization schemes. For example, the COUPLED_SYNC keyword can be used to choose different coordination schemes to simulate a distributed coupled model across various domains. In this way, the standard can easily adopt any new schemes that may appear in the future.

A given domain can use any model representation, since each domain should contain a simulation engine capable of executing that model partition. For example, the SOAP-based DCD++ [26] partitions a model among different machines where each machine is wrapped within a SOAP port. Figure 17.21 shows an example of such partition. Lines 2–11 list all machines participating in the distributed simulation. Each machine has a rank (i.e., id) and a SOAP-port URI. Lines 12–19 describe the model partitions. In this example, a 30×30 Cell-DEVS fire coupled model (i.e., each cell is an atomic model) is split between two machines.

17.5.3 CD++ DEVS MODEL STRUCTURE AND BEHAVIOR

The CD++ DEVS modeling behavior is the same for both SOA versions. This is mainly because DCD++ separates SOA (RESTful or SOAP Web-services) from the actual modeling/simulation environment. This separation allows CD++ to be portable to other platforms other than SOA and to keep the door open for algorithms optimization and other improvements. In other words, separation between simulation environment and Web-services allows both parts to evolve independently with less complexity.

CD++ represents its coupled models in textual format by connecting ports of models together similar to what has been discussed so far. The coupled model

```

1 <Grid>
2   <MACHINES>
3     <MACHINE>
4       <MACHINE_RANK>0</MACHINE_RANK>
5       <MACHINE_URI>http://.../CDppPortType</MACHINE_URI>
6     </MACHINE>
7     <MACHINE>
8       <MACHINE_RANK>1</MACHINE_RANK>
9       <MACHINE_URI> http://.../CDppPortType</MACHINE_URI>
10    </MACHINE>
11  </MACHINES>
12  <MODEL_PARTITIONS>
13    <PARTITION machine="0">
14      <ZONE>fire(0,0)..(14,29)</ZONE>
15    </PARTITION>
16    <PARTITION machine="1">
17      <ZONE>fire(15,0)..(29,29)</ZONE>
18    </PARTITION>
19  </MODEL_PARTITIONS>
20 </Grid>

```

FIGURE 17.21 SOAP-based DCD++ cell-DEVS model partitioning.

representation is parsed and its information is loaded into a coordinator where it is simulated. For example, Figure 17.22 shows a CD++ coupled model for a barbershop. In this example, “top” coupled model is defined in lines 2–27. Lines 4–7 lists the models enclosed in the “top” coupled model. In this case, line 5 shows an atomic model instance named “reception” defined in C++ class “Reception.” This allows more instances of an atomic model to be used, if needed. Line 6 declares a coupled model named “Barber” and its definition should come later. Lines 8–13 define “top” model input/output ports. Lines 14–25 define port connections for the “top” model. For instance, lines 15–23 define a connection from input port “newcust” of the “top” model to port “newcust” of the “reception” atomic model. Lines 29–32 define the “Barber” coupled model in a similar way to the “top” model. Lines 33–39 initialize certain values for the “reception” atomic model.

The CD++ provides a template for modelers where they need to overwrite four C++ methods, as shown in Figure 17.23. Methods `initFunction`, `externalFunction`, `internalFunction`, and `outputFunction` are wrappers to initialize model specific values, handle a message from a port, handle an internal transition, and generate outputs, respectively. The new atomic model is then inherited from the shown `Atomic` class in Figure 17.23. During simulation, those methods are invoked by the atomic model “simulator” according to the DEVS formalism.

17.6 COSMOS

The design of CoSMoS is based on the separation of model specification and simulator protocol. This partitioning was used to develop the concept of the *Shared*


```

1 <Models>
2   <Model Type="Coupled">
3     <Name>top</Name>
4     <Components>
5       <Name Type="Atomic" Class="Reception">reception</Name>
6       <Name Type="Coupled">Barber</Name>
7     </Components>
8     <Ports>
9       <Port Type="in">newcust</Port>
10      <Port Type="in">next</Port>
11      <Port Type="out">cust</Port>
12      <Port Type="out">finished</Port>
13    </Ports>
14    <LINKS>
15      <LINK>
16        <FROM>
17          <Port>newcust</Port>
18        </FROM>
19        <TO>
20          <Component>reception</Component>
21          <Port>newcust</Port>
22        </TO>
23      </LINK>
24      ...
25    </LINKS>
26    ...
27  </Model>
28
29  <Model Type="Coupled">
30    <Name>Barber</Name>
31    ...
32  </Model>
33  <Model Type="Atomic">
34    <Name>reception</Name>
35    <Parameter>
36      <Name>numberOfChairs</Name><Value>3</Value>
37    </Parameter>
38    ...
39  </Model>
40 </Models>

```

FIGURE 17.22 CD++ coupled model example.

Abstract Model (which will be fully discussed in the following chapter). The concept of interoperability, however, cannot be completely divorced from model specification. One simulation engine can correctly execute a model that is developed for another simulation engine if and only if both simulation engines adhere to the same mathematical specification. The implication is to have a syntactical standard for model specification that conforms to the mathematical specification.

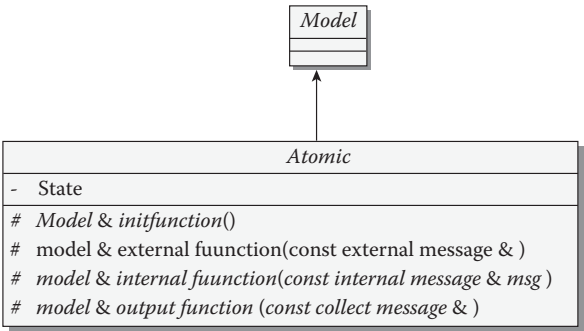


FIGURE 17.23 CD++ atomic model template.

Model specification should consist of (i) visual syntax, (ii) database syntax, and (iii) programming syntax. To achieve this objective, the Component-based System Modeling (CoSMo) framework has been developed [15,16]. The CoSMo has a unified framework that supports logical, visual, and database persistent modeling. It is aimed at discrete-event, discrete-time, continuous-time, and cellular automata specifications. The goal is to provide a mechanism for visual model development and for storing logical models in databases, while also providing support and management facilities for developing a family of interrelated models. Such capabilities are useful for individual users and crucial for teams of users, for example, for purposes of software/hardware codesign [18].

The top down concept of CoSMo has a visual syntax for logical models, which serves as the basis for storing logical specification of models in alternative database and flat files. Given the stored logical models, they can be translated to different programming languages for different target simulation engines. Other approaches to interoperability are generally defined in terms of programming syntax such as C++, Java, and XML—standardization is based on the traditional software engineering concepts and techniques. For example, simulation tools such as DEVS-Suite render a visual representation of models from implemented code that persist in flat files. In CoSMo, visual and database persistence of models are proposed to be integral to standardization.

The CoSMo environment supports developing Parallel DEVS models and generating partial code for execution using DEVS-Suite. For coupled DEVS models, it generates complete code. For atomic models, it generates partial code. For the transition, output, and time advance functions only the template code is generated. Other translators can be developed in a straightforward manner for other simulators such as ADEVS. As shown in Figure 17.24, the separation of a modeling engine and a simulation engine has important implications for standardization particularly when standardization is not at the level of XML [15] and DEVSML [19], or high-level programming languages such as Java. In effect, the CoSMo Simulator environment (called CoSMoS) is built from CoSMo and DEVS-Suite. Users can develop models using well-formed visual syntax that in turn allows generating “standardized” models stored in relational databases, which in turn can be used to generate XML-based and high-level programming code. Standardization, from the perspective of

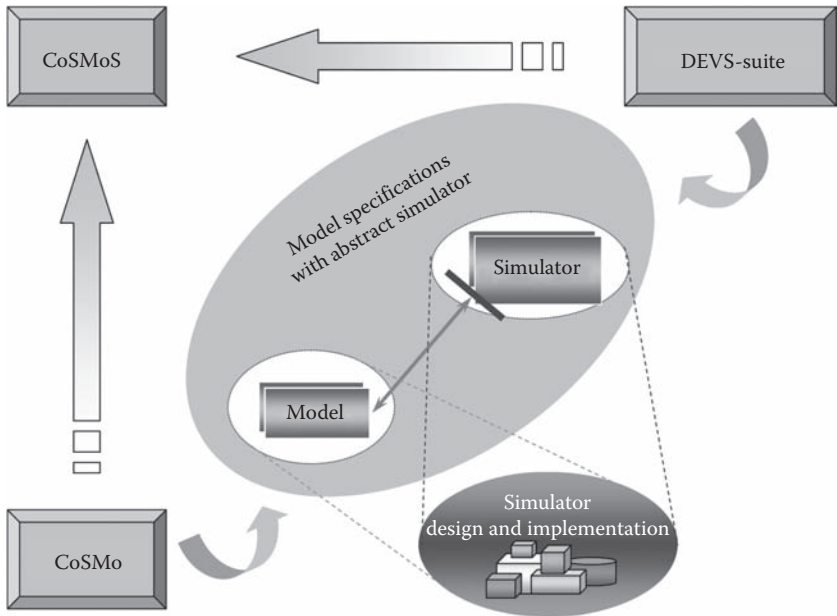


FIGURE 17.24 Separation of a modeling engine and a simulation engine.

the CoSMoS framework, enhances visual model development, model management via databases as well as reducing errors in programming code.

The underlying concept of CoSMoS can enable interoperability among multiple simulation engines. The CoSMoS architecture and realization supports adding translators as plug-ins. The CoSMoS framework also aids in supporting interoperating simulators given current and new generations of middleware technologies such as service-oriented computing. As a result, the XML Schema translator developed for CoSMo can be used to support SOA-based distributed simulation. For example, the DEVs/SOA environment [19] can be a target simulation engine and thus writing models at the level of DEVsML, one can visually develop models that are logically correct with respect to the parallel DEVs formalism. Therefore, users can automatically generate code for atomic and coupled models given target simulation engines. The structures of the atomic and coupled models as well as their relationships are formulated as a set of tables and relations in a relational database. The axioms of CoSMo also provide a set of rules for creating a family of models such that composition and specialization relationships among different model designs do not conflict with one another.

The visual, and programming elements for DEVs mathematical specifications, and the support repositories, must account for the behaviors of atomic and coupled models. A key obstacle to achieving this goal is the lack of visual notation for atomic models. Different authors have proposed and used variations of the Statecharts formalism to represent states and functions of atomic models. The visual statechart elements (states, function, and transitions) are adapted such that internal, external, and output functions can be distinguished (solid, small dashed, and large

dashed lines are used). In other work [20], a notation that is also derived from the Statecharts is developed for specifying the conditionals and distinguishing between transition functions with $at = 0$ and $ta > 0$. From the perspective of making DEVS accessible to a broader community of users, the UML visual notation is proposed for modeling the behavior of atomic models [21]. In this work, the developed DEVS/UML models are executed using a parallel DEVS simulation engine. Unfortunately, developing a simple visual notation suitable for complete behavior specification of atomic model remains challenging since Java and other such programming languages allow arbitrary logic and, compared to visual notation, can capture complete details of a model dynamics. Therefore, existing notations and tools can only support generating a partial and complete source code for atomic models. The CoSMo modeling engine can be extended once a standardized visual notation for atomic models becomes available. The extension of the relational database and model translators to the target programming code will be straightforward.

Finally, it is noted that although the use of generic modeling formalisms is important, it is equally important to provide users with user-friendly, yet formally grounded, capabilities for developing simulation models. In DEVS, as in other general-purpose M&S approaches, domain knowledge must be incorporated into the generic atomic and coupled model specifications. It is only then that the model dynamics of a system can be simulated and evaluated. For certain application domains, such as electrical systems, visual notation (and thus programming code templates) are available. However, for other application domains, such as biological and social-ecological systems, universal agreements do not exist. Nonetheless, standardization for the generic atomic and coupled DEVS models is highly important and undoubtedly necessary toward developing domain-specific standards and more generally enabling a new kind of model verification and simulation validation.

17.7 SUMMARY

This chapter has focused on various methods for standardizing DEVS model representations, which is necessary to enable platform-independent DEVS models to interoperate with each other. The concept relies on defining a generic representation that other tools can use to understand how a model is created, which allows sharing of existing models. Most of the examples in this chapter are based on XML descriptions that can be transformed and manipulated to achieve interoperability.

We showed how DEVSMML (an XML notation built on JAVAMML) can be used to transform different compatible DEVS platforms. DEVS coupled models are built using Document Type Definitions, and the atomic models are transformed into a combination of JavaMML code with special tags.

DEVSSOA includes a model library and a simulation library (implemented in Java and .Net). Interoperability is achieved by using standardized XML for message passing, and the ability to publish simulators, atomic, and coupled models as web services.

DML, the DEVS Markup Language, standardizes the DEVS model representation using an XML representation of specific languages (e.g., JavaMML), and provides

the common denominator of most object-oriented languages in order to allow the use of the features of language libraries. Modelers can use their favorite programming language, which is interpreted by a simulation proxy.

DCD++ supports simulation on SOAP-based or RESTful Web-services (also called the Distributed DEVS Simulation Protocol, DDSP, and the RISE). In both cases, the different simulation engines can interoperate to execute a complete simulation. DDSP introduces an XML-based environment to interoperate different DEVS implementations, using SOAP-based messages. RISE provides an interoperability framework (and API based on URIs so the modelers can create models, simulations, and experiments), a model interoperability layer (providing XML rules to combine different models together), and simulation synchronization algorithms.

Finally, CoSMoS provides a visual syntax, a programming syntax, and a database syntax that allow the models to be visually created, to be stored in a persistent data store, and overall make implementation and interoperability of components easy. CoSMoS supports the creation of models and generates code that can (partially) be executed on the DEVS-Suite environment.

These different efforts show the varied possibilities in achieving model interoperability, sharing, and reuse, based on a DEVS-based methodology.

REFERENCES

1. Chreyh, R., and G. Wainer. "CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames." *2009 Spring Simulation Conference*, San Diego, CA, March 2009.
2. Mosterman, P. J., D. Bouldin, and A. Rucinski. "A Peer Reviewed Online Computational Modeling Framework." Paper ID pmo131763. *Proceedings of the Canadian Design Engineering Network (CDEN) 2008 Conference*, July 27–29, Halifax, Nova Scotia, 2008.
3. Mittal, S., J. L. Risco-Martín, and B. P. Zeigler. "DEVS/SOA: A Cross-Platform Framework for Net-Centric Modeling and Simulation in DEVS Unified Process," *Simulation* 85, no. 7 (2009): 419–50.
4. Touraille, L., M. K. Traore, and D. R. C. Hill. "A Mark-up Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models." *Proceedings of the 2009 ACM/SCS Spring Simulation Multiconference, SpringSim 2009*, March 22–27, San Diego, CA, 2009.
5. Lemos, M. *MetaL: An XML-Based Meta-Programming Language*. <http://www.meta-language.net/>
6. Wiharto, M., and P. Stanski. "An Architecture for Retargeting Application Logic to Multiple Component Types in Multiple Languages." *Fifth Australasian Workshop on Software and System Architectures*, 2004.
7. L'Ecuyer, P., L. Meliani, and J. Vaucher. "SSJ: A Framework for Stochastic Simulation in Java." *Proceedings of the 2002 Winter Simulation Conference*, San Diego, CA.
8. Badros, G. J. "JavaML: A Markup Language for Java Source Code," *Computer Networks* 33, no. 1 (2000): 159–77.
9. McArthur, G., J. Mylopoulos, and S. K. Ng. "An Extensible Tool for Source Code Representations Using XML." *Proceedings of the Ninth Working Conference on Reverse Engineering*, Richmond, VA, 2002.

10. Mamas, E., and K. Kontogiannis. "Towards Portable Source Code Representations Using XML." *Proceedings of the Seventh Working Conference on Reverse Engineering*, Brisbane, Australia, 2000.
11. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming." *Proceedings of the European Conference on Object-Oriented Programming*, June, Finland. New York: Springer-Verlag, 1997.
12. Zeigler, B., T. Kim, and H. Praehofer. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. New York: Academic Press, 2000.
13. Vangheluwe, H., J. de Lara, J.-S. Bolduc, and E. Posse, "DEVS Standardization: Some Thoughts." *Winter Simulation Conference*, 2001.
14. Al-Zoubi, K., and G. Wainer. "Interfacing and Coordination for a DEVS Simulation Protocol Standard." *The 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, British Columbia, Canada, 2008.
15. Sarjoughian, H. S., and R. Flasher. "System Modeling with Mixed Object and Data Models." *DEVS Symposium, Spring Simulation Multi-Conference*, Norfolk, VA, 2007.
16. Sarjoughian, H. S., and V. Elamvazhuthi. "CoSMoS: A Visual Environment for Component-Based Modeling, Experimental Design, and Simulation." *Proceedings of the International Conference on Simulation Tools and Techniques*, March, 1–9, Rome, Italy, 2008.
17. Hu, H., and H. S. Sarjoughian. "Discrete Event Simulation of Network Systems Using Distributed Object Computing." *Proceedings of the International Symposium on Performance Evaluation and Telecommunications Systems*, 884–93, San Diego, CA, 2005.
18. Traoré, M. "A Graphical Notation for DEVS." *High Performance Computing & Simulation Symposium, Proceedings of the Spring Simulation Conference*, March, San Diego, CA. New York: ACM Press, 2009.
19. Mittal, S., J. L. Risco-Martín, and B. P. Zeigler. "DEVSMML: Automating DEVS Execution Over SOA Towards Transparent Simulators." *Spring Simulation Multiconference, SpringSim 2007*, 287–95, Norfolk, VA, 2007.
20. Huang, D., and H. S. Sarjoughian. "Software and Simulation Modeling for Real-Time Software-Intensive System." *The 8th IEEE International Symposium on Distributed Simulation and Real Time Applications*, 196–203, October, Budapest, Hungary. New York: IEEE, 2004.
21. Mooney, J., and H. S. Sarjoughian. "A Framework for Executable UML Models." *High Performance Computing & Simulation Symposium, Proceedings of the Spring Simulation Conference*, March, San Diego, CA. New York: ACM Press, 2009.
22. Wainer, G. "Discrete-Event Modeling and Simulation: A Practitioner's Approach." Boca Raton, FL: CRC Press, Taylor & Francis Group, 2009.
23. Al-Zoubi K., and G. Wainer. "RISE: REST-ing Heterogeneous Simulation Interoperability". *Proceedings of the Winter Simulation Conference (WSC 2010)*. Baltimore, Maryland, USA. 2010.
24. Al-Zoubi, K., and G. Wainer. "Performing Distributed Simulation with RESTful Web-Services Approach." *Proceedings of the Winter Simulation Conference (WSC 2009)*. Austin, TX, 2009.
25. Al-Zoubi, K., and G. Wainer. "Using REST Web Services Architecture for Distributed Simulation." *Proceedings of Principles of Advanced and Distributed Simulation PADS 2009*, Lake Placid, New York, 2009.
26. Wainer, G., R. Madhoun, and K. Al-Zoubi. "Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web Services". *Simulation Modelling Practice and Theory*. 16, no. 9 (2008): 1266–1292.

18 Standardizing DEVS Simulation Middleware

Gabriel A. Wainer, Khaldoon Al-Zoubi, Olivier Dalle, Saurabh Mittal, José L. Risco Martín, Hessam Sarjoughian, and Bernard P. Zeigler

CONTENTS

18.1	Introduction.....	459
18.2	DEVS/SOA.....	460
18.2.1	DEVS/SOA Simulation Layer.....	460
18.2.2	DEVS/SOA Application.....	464
18.3	Distributed DEVS Simulation Protocol (DDSP)	465
18.3.1	Introduction to DDSP	466
18.3.2	Web-Service DEVS Wrapper.....	467
18.3.3	Model Structure XML Document	472
18.3.4	Format and Content of Messages.....	475
18.4	Shared Abstract Model.....	477
18.5	RESTful Interoperability Simulation Environment (RISE)	481
18.6	DEVS Namespaces	488
18.7	Summary	491
	References.....	492

18.1 INTRODUCTION

As discussed earlier in Chapter 16, there are two different interoperability objectives that one must consider when standardizing DEVS environments: (1) *Standardizing DEVS model representation* to allow a platform-independent DEVS model representation that can be executed by any DEVS-based simulation tool. (2) *Standardizing Interoperability Middleware* to interface different simulation environments and allow synchronization for the same simulation run across a distributed network regardless of their model representation.

This chapter focuses on approaches by different groups to *standardize the simulation middleware*. All of the implementations discussed in this chapter are based on a Service Oriented Architecture (SOA) design, which employs the concept of deploying services so that they can be invoked by clients. This concept is applied in Common Object Request Broker Architecture (CORBA) and SOAP/REST Web-services.

This kind of middleware is of interest in order to overcome current distributed simulation challenges and to meet future expectations in this area [1,2]. A standardized DEVS simulation protocol should enable different DEVS implementations to simulate the same DEVS model hierarchy partitioned between various DEVS engines in distributed fashion. Moreover, each DEVS domain in this distributed system should be able to execute its legacy models and, thus, perform distributed simulation experiments between different heterogeneous models and engines. The middleware designs showed in this chapter offer simulation resources as a set of services that can be invoked by simulators, and where these simulators act as peers (i.e., clients and servers at the same time) to each other to synchronize a simulation session.

The designs presented in this chapter show that different methods can be employed for DEVS simulation synchronization. One option is to expose the actual DEVS simulators and coordinators (and to take care of synchronization at that level). A second option considers placing the DEVS coordinators and simulators of a domain behind a wrapper so that this wrapper routes all information to the appropriate coordinator/simulator. Further, the synchronization protocols can pass all simulation data between distributed entities in the form of procedure parameters (relying on CORBA or SOAP-based Web-services), or can pass simulation data between distributed entities in the form of XML messages.

18.2 DEVS/SOA

As discussed in the previous chapter, DEVS/SOA is a DEVS simulation platform implemented in both Java and .NET programming environments. It manages messages in XML format, which enables the publication of web services (including the publishing of simulators and even atomic or coupled models as web services). As discussed earlier, there are now many libraries for expressing DEVS models around the world. All have efficient implementations for executing the DEVS protocol and most of them provide the advantages of Object Orientation (such as encapsulation, inheritance, and polymorphism). In order to simplify notation, we use the term DEVS/JAVA to denote a DEVS library implemented in Java (for example, DEVSJAV or xDEV) and DEVS/.NET to denote a DEVS library implemented in any language supported by .NET (for example ADEV in C, CD++ in C++, and DEV.S.NET in C#).

18.2.1 DEVS/SOA SIMULATION LAYER

The DEVS/SOA simulator integrates the common DEVS/SOA atomic model interface pictured at the bottom of Figure 18.1 to support multiframework interoperability. This modeling interface plus a customized adapter allow the aggregation of DEVS/JAVA native models (xDEV, MicroSim, DEVSJAV, etc.) in parallel with DEVS/.NET based models (ADEV, CD++, DEV.S.NET, etc.). A noteworthy achievement of a DEVS modeling common interface is the presence of an embedded “translator” that is in charge of mapping incoming messages to XML and outgoing messages the other way around.

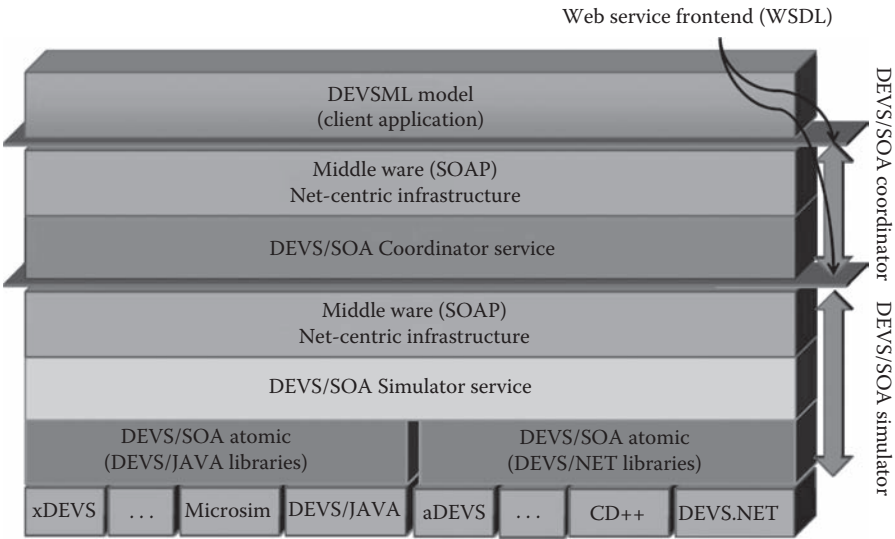


FIGURE 18.1 DEVS/SOA architecture.

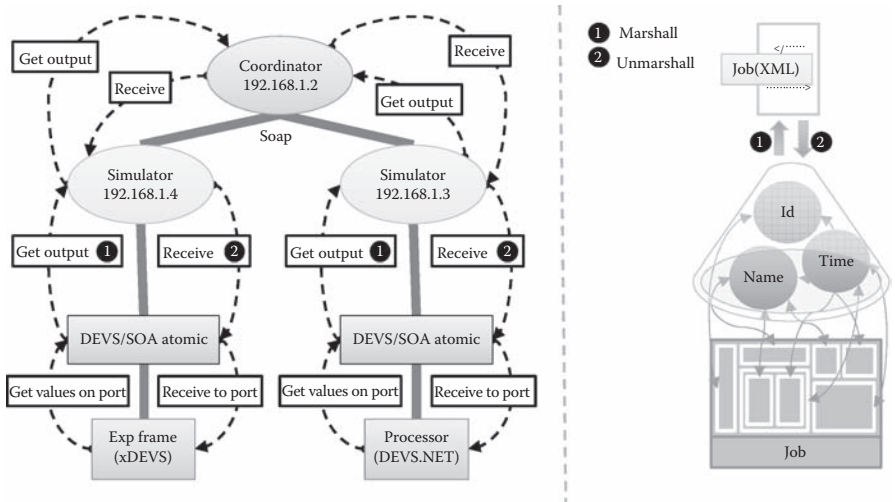


FIGURE 18.2 DEVS/SOA XML message mapping (centralized in root coordinator).

The communication between a given DEVS/SOA atomic models and its corresponding Simulator are detailed in an XML format. Therefore, the essential duty of the aforementioned DEVS modeling interface translator is to bind the platform independent XML data types to the message objects exchanged between the DEVS/SOA simulator and the DEVS atomic models.

Figure 18.2 illustrates the message passing process of the coupled model consisting of an Experimental Frame (*ExpFrame*, which generates jobs to be processed, and

it computes the performance obtained when processing those jobs) and a *Processor* (which models the activity of a server that we want to simulate) using DEVS/SOA. Dashed lines represent the model dataflow.

- i. The *ExpFrame* DEVS/SOA atomic model requests an output message from a platform specific model (xDEVS or DEVS.NET) using *GetValuesOnPort* and *GetOutput*
- ii. A simulator requests an XML formatted output message from the DEVS/SOA atomic model using *GetOutput*
- iii. The Coordinator requests an XML formatted output message from a Simulator and sends it to another simulator through the corresponding ports coupling
- iv. A simulator receives an XML formatted message using *Receive*
 - v. The DEVS/SOA atomic model receives a platform specific message
 - vi. The platform specific *Processor* model receives the value of the message

It should be mentioned that each operation embodies a specific input or output port. The right-hand side of Figure 18.2 symbolizes the translation procedure that converts a DEVS message (DEVS model input or output) built from class objects into an XML-based document (marshall), and vice versa (unmarshall). The corresponding steps (1) and (2) can be seen on the left-hand side of the figure. Because of this XML serialization, DEVS model inputs and outputs are sent through the web within SOAP.

Figure 18.3 depicts the DEVS/SOA simulator interface. The DEVS/SOA simulator is implemented as a classic DEVS simulator, with the difference that data are managed in XML. Thus, the input and output of a function can be declared using standard data types (string and double) and the publication of such interfaces as web services can be easily performed.

Technically, these operations, expressed in the WSDL file of the DEVS/SOA Simulator service, wrap the classic DEVS simulation protocol with the purpose of achieving a qualified and standardized DEVS simulation.

```
public interface Simulator{
    public void setModel(String XmlModel); // Receive the
    corresponding DEVXML part
    void initialize(double t); // Typical DEVS initialization function
    double getTN(); // Returns the time of the next event
    void deltfcn(double t); // Encapsulate the call to delttint,
    deltext and deltcon in the model
    void lambda(double t); // Output function
    String[] getOutput(String portName); // Returns the values at
    output ports as XML
    void receive(String portTo, String[] xmlMessage); // Receive
    messages in XML format
}
```

FIGURE 18.3 Simulator interface.

```
public interface Coordinator{  
    public void setModel(String XmlModel); // Initialize the model  
    with the DEVSML file  
    public String[] simulate(int numIterations); // Simulate a given  
    number of cycles  
}
```

FIGURE 18.4 DEVS/SOA coordinator interface.

Finally, Figure 18.4 shows the DEVS/SOA Coordinator interface. Likewise, as pictured in Figure 18.1, the Coordinator service layer together with a web service frontend invokes DEVS simulation operations against the Simulator service layer. Again, the communication is driven by the standard SOAP protocol and the operations are detailed in the Simulator service WSDL file allowing platform-independent interactions. Moreover, assuming that a coordinator is already provided by the client application with the root DEVS model based on DEVSML syntax, the initial coordination task comprehends parsing the DEVSML document looking for DEVS atomic or coupled models and their connections. Each atomic or coupled model supplies the coordinator with the associate remote simulator location in order to enable communications among them. The connections among the models let the coordinator be aware of the entire distributed circuit to carry out proper performance analysis. After the communication between the coordinator (coordination service layer) and the respective simulators (simulation service layer) is set up, the coordinator proceeds to activate and initialize all simulators by feeding them with their proper DEVSML-based model definition. Next, the coordinator continues with the simulation initialization within the DEVS/SOA simulation protocol.

As Figure 18.1 illustrates, the client application requires a web service frontend to provide access to the Coordinator service layer. The baseline communication is settled within the SOAP protocol and the operations are detailed in the Coordinator WSDL file. This arrangement provides platform independent interactions. Assembled with a web service framework, the client application acquires the capacity to look over the web for the coordinator service hosted at the remote location stated at the DEVSML root model (see Section 17.3.2). Once the connection is established, the web client application activates the remote coordinator by supplying a model specified using DEVSML, and it subsequently executes the simulation according to specific parameters. As soon as the simulation is completed, the client application receives a summary of the overall simulation performance and cumulative results. The aforementioned operations are specified inside the DEVS/SOA Coordinator WSDL file.

When composing a DEVS/SOA model, one important simplification, which is not mandatory, is that every coupled model can work as an atomic model [3]. For example, Figure 18.5 shows that the root model can be simulated in two different ways: with each atomic model on a different computer (as seen on the left-hand side of the figure), or with the Experimental Frame on one computer (i.e., the coupled model *ExpFrame* shown on the right-hand side figure, which is the composition of *Generator* and *Transducer* atomic models on the left) and the *Processor* on another (as seen on the right-hand side).

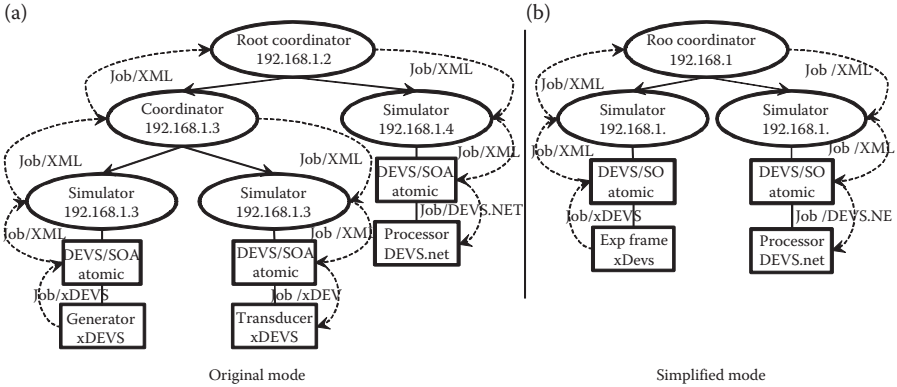


FIGURE 18.5 Simplification of DEVS models.

18.2.2 DEVS/SOA APPLICATION

As stated above, to build a web service-based infrastructure, both the coordinator and simulators are published as web services. Thus, we have two Coordinators (DEVS/SOA Coordinator and stub) and two Simulators (DEVS/SOA Simulator and stub).

On the server side, all the Coordinators and the Simulators must be placed in a directory accessible by the selected web server (Apache Tomcat and Axis2 to allow web service development, or MS Internet Information Server with .NET), as well as all the models to share by every particular server and their corresponding DEVS libraries to allow interoperability. For example, for a particular case where Apache Tomcat and Axis 2 are being used, the server side content comprises:

- TOMCAT_HOME/webapps/axis2/WEB-INF/services:
 - Simulator.aar: Axis2 archived web service and stub of DEVS/SOA Simulator
 - Coordinator.aar: Axis2 archived web service and stub of DEVS/SOA Coordinator
- TOMCAT_HOME/shared/lib:
 - xdevs.jar: xDEVS M&S (modeling and simulation) API
 - devsjava.jar: DESJAVA M&S API
 - Several xDEVS and DEVSJAVA models

Since both web services and stubs are installed in the server, they are able to act both as a simulation services and as clients to another server. For a fully Symmetrical Service architecture design, refer to Mittal, Risco-Martín, and Zeigler [3]. By this symmetrical design, when a distributed DEVS/SOA is executed on multiple machines (say on a server farm), each machine can serve as a DEVS coordinator as well as a DEVS simulator facilitating the recursive DEVS hierarchical design principles.

Similarly, the client application contains just two executable classes (the DEVS/SOA coordinator and simulator stubs), namely *DevsSoaSimulator.jar* and

DevsSoaCoordinator.jar. A graphical UI client for DEVS/SOA was implemented in Mittal, Risco-Martín, and Zeigler [3]. To execute the model, the user must compose an XML file describing the distributed architecture (as depicted in Figure 18.5) and run the simulation as follows (in the Java client application):

```
// xmlCoupledModelAsString is a DEVSMML file, which has
// been previously loaded
CoordinatorServiceInterface service=new
    CoordinatorService("http://localhost:8080/devsoa/
        Coordinator",true);
service.setCoupledModel(xmlCoupledModelAsString);
String[] response = service.simulate(numIterations);
```

or using the client Coordinator as a executable file:

```
java -jar DevsSoaCoordinator.jar -file=ef-p.xml
-numIter = 110.
```

To this end, the software required in a DEVS/SOA Java Axis2 client application is:

- Java Development Kit (JDK): Version 1.4 or later
- Axis2: Version 1.4.1 or later standard binary distribution

To conclude, the purpose of DEVS/SOA is to support distributed simulation and interoperability. The distributed arrangement of the ongoing simulations enables the user to partition the original model and distribute it between several processors or cores in the same computer, among several computers connected through the Internet, or between both. In the same manner, the user is able to compose a complex DEVS model using different submodels that may be hosted on different computers. Furthermore, the interoperability quality allows the user to compose a complex DEVS model with different submodels implemented using distinct DEVS M&S frameworks or libraries. Every computer involved in the M&S process must act as a repository providing DEVS models (implemented for a specific DEVS library), as a server (providing Simulators and Coordinators as web services), and as a client (communicating with the Coordinator on any machine).

18.3 DISTRIBUTED DEVS SIMULATION PROTOCOL (DDSP)

In Al-Zoubi and Wainer [5], a flexible and scalable XML-based, message-oriented mechanism was developed with the goal to allow interoperability between different DEVS implementations. The main objective of the protocol is to enable different DEVS implementations to interface and coordinate among each other to simulate the same model structure across their diverse domains. To do so, the developed simulation protocol uses SOAP-based Web-services technology as the communication framework to exchange control and standardized simulation XML messages.

18.3.1 INTRODUCTION TO DDSP

The idea of DDSP is to provide interoperability with minimum design changes to each DEVS implementation, mainly by hiding the detailed implementation behind a *wrapper* (i.e., a SOAP-engine port) and focusing only on the exchanged XML messages. This point is important because various DEVS implementations are different (even if they are implemented with the same programming language). Interfacing the same tool implementation in a parallel/distributed environment can require weeks of programming and debugging by programmers who understand that tool implementation very well. One cannot expect interfacing different DEVS implementations that were developed by different independent teams to be internally structured the same. Further, different teams have extended their tools over the years to accommodate different optimizing algorithms or modeling techniques. For example, the Cell-DEVS extension [6] allows for representing each cell in the cell space as a DEVS model that is only activated when it receives external inputs from its neighboring cells. The CD++ [7] provides an environment for DEVS and Cell-DEVS models. However, it extends the software design into different C++ classes to implement both DEVS and Cell-DEVS. Figure 18.6 shows a fragment of the design of distributed CD++ (DCD++). The figure clearly shows that this version of the simulation software uses a specific implementation to simulate Cell-DEVS models (using the *AtomicCell* and *CoupledCell* methods). The various DEVS versions have two things in common: That coordinators synchronize coupled models and simulators execute atomic models where the simulation is advanced according to the DEVS theory rules. However, each DEVS version provides different software design and implementations. In fact, the internal implementation for a DEVS coordinator, for instance, can vary between parallel, stand-alone, and distributed for the same DEVS tool because each of these DEVS coordinators can use a different algorithm to coordinate its children.

Hiding the internal implementation increases the protocol chances of success since various DEVS teams are not expected to change their internal design and software implementation in a way that jeopardizes their existing DEVS tools integrity. Further, they can have full freedom to extend/change their own internal software implementation. This is because a DEVS tool is always in conformance

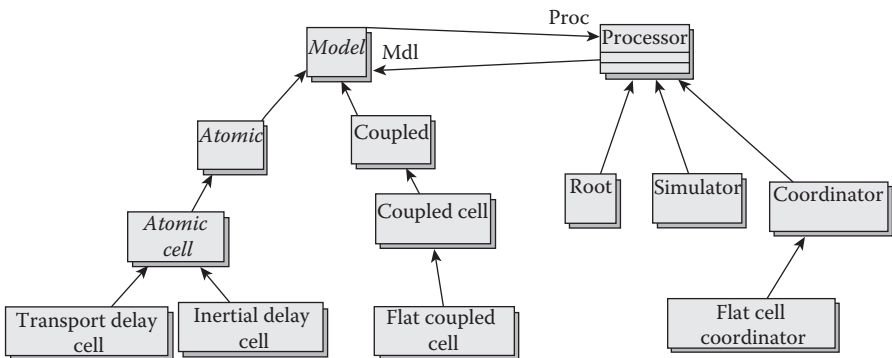


FIGURE 18.6 Snippet of the DCD++ model/simulators hierarchy.

with the standards as soon as it handles the standardized XML simulation messages correctly.

The DEVS Distributed Simulation Protocol also supports interfacing DEVS legacy models. This allows a modeler to assemble and simulate heterogeneous DEVS models that were originally intended to run in a specific DEVS environment. The protocol expects each tool to react to expected messages (with a standardized format constructed as XML documents) in order to correctly synchronize and carry out simulation of the overall model (which is spread over different domains). Having a message-oriented protocol that hides implementation detail (behind wrappers) and focuses only on the information needed (within exchanged messages) has many advantages. To summarize a few:

- **Maintainability:** Protocol changes are only applied to the protocol messages rather than to every DEVS implementation.
- **Scalability:** The contents of the XML message being exchanged are easy to add (or remove) by adding (or removing) the Remote Procedure Calls (RPCs) interface.
- **Testing:** Local testing is easy to perform by each group before executing integration testing between different DEVS domains. The general rule is that if a DEVS implementation can interface with itself via exchanging XML standardized messages, it should be able to interface with a different DEVS implementation using the same standardized messages (in case both implementations conform to the standardized messages and rules).

18.3.2 WEB-SERVICE DEVS WRAPPER

Each DEVS implementation should execute its own specific models. This requirement enables both the utilization of hundreds of legacy models for each DEVS tool as well as the integration with other models in different DEVS tools. This requirement is essential to make DEVS standards attainable because we can never expect all legacy models to be rewritten. This requirement is satisfied by enclosing all models in a single outer model and making each DEVS tool responsible for simulating its specific models. For example, in Figure 18.7, *Coupled1* can be in DEVSJAV

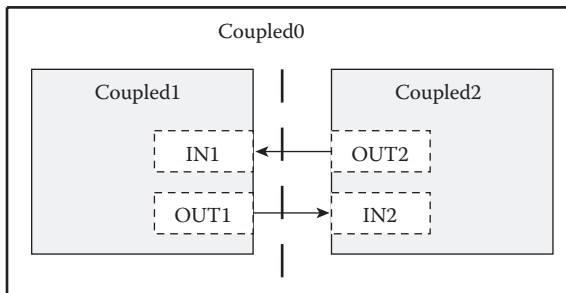


FIGURE 18.7 Coupled model partitioned across DEVS domains.

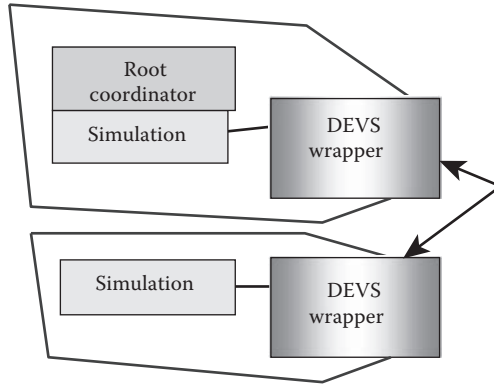


FIGURE 18.8 Connecting two DEVS domains.

Coupled2 can be in DCD++. In this case, the main DEVS domain owns the Root coordinators and simulates both heterogeneous models, giving the impression of simulating a single distributed heterogeneous DEVS model.

As shown in Figure 18.7, both coupled models interface without worrying about how the other implementation performs the simulation internally. Therefore, coupled models are viewed as black boxes with input/output ports. However, it is still possible for a DEVS implementation to know more details about the model structure in other domains, depending on the level of detail that is made available to the domains when the structure is distributed, as described in the XML structure document.

The concept is to have each DEVS implementation use a single communication entry point, implemented as a DEVS-Wrapper (Figure 18.8). Therefore, a coupled model may physically be partitioned among different machines within a DEVS implementation domain, but other DEVS domains “believe” the coupled model actually exist on the machine through which they communicate with the coupled model. The DEVS-Wrapper is actually a web-service port that exposes a number of stubs (interfaces), allowing other DEVS domains to invoke them in an RPC-style mechanism, as discussed in the communication framework section. Therefore, the DEVS-wrapper interfaces are described in the WSDL document allowing domains to construct the necessary stubs.

This requirement simplifies the coordination via one single web-service port described in a single WSDL document. This approach does not require the DEVS-wrapper operations to be exposed in a separate web-service port or to be merged with other existing ports, but it leaves this decision to individual teams since it is a software design issue rather than a standardization one. Further, the DEVS-wrapper port needs to be described by a WSDL document where other domains can use the standardized interfaces. The DEVS-Wrapper component is expected to perform the following tasks:

- Translate incoming standardized simulation messages to specific domain simulation messages

- Transmit simulation messages to other DEVS domains according to the DEVS standards
- Route incoming simulation messages to the correct models/ports within its domain

The protocol should minimize its dependency on the communication framework, requiring few (or no) changes to the standardized simulation messages if one needs to move the simulation protocol to other communication engines in the future. In DCD++, this requirement is implemented by sending all simulation messages as XML documents in SOAP attachments via exposed DEVS-wrapper interfaces. Therefore, if the communication mechanism changes, those same XML documents can still be transmitted without changes.

Although SOAP messages are standardized XML documents, they are hidden from the web-service applications and are only seen as programming stubs with input/output parameters. Consequently, a DEVS standard would need to define its own XML messages. Eventually, the standard is realized by the programming code, hence if it only relies on SOAP XML messages, all DEVS interfaces become a matter of simply gluing programming remote procedures together. This makes the standard extremely sensitive to changes since programming language procedures by nature are sensitive to many factors such as the order, type, and number of parameters that are passed into them. Therefore, the standard can never claim to be using XML messages for communication because the SOAP messages are only handled by the SOAP engine (i.e., the communication layer below a web-service application that is responsible for SOAP messages handling), and this engine is enabled to invoke the appropriate service stub.

The DDSP implementation uses web-services technology to transfer standardized simulation messages between different domains. All messages are transmitted through SOAP/HTTP engines, hence wrapped within SOAP and HTTP envelopes, as shown in Figure 18.9, where a DEVS Wrapper communicates with other DEVS domains by invoking the deployed-service stubs in a remote procedure call style.

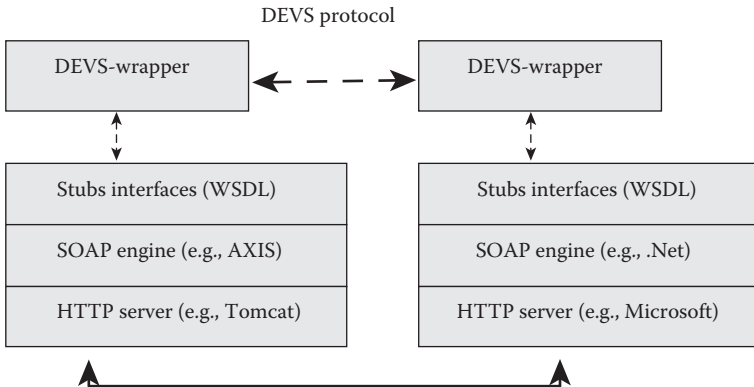


FIGURE 18.9 Connecting domains using web-services.

```

1  import javax.activation.DataHandler;
2
3  public interface DEVSWrapperType extends java.rmi.Remote {
4
5      public DataHandler retrieveResultFile(int SupportiveSession);
6      public boolean startSimulation(int SupportiveSession);
7      public boolean isSimRunning(int session);
8      public boolean StopSimulation(int session);
9      public boolean setDEVFXML(int session, String filename,
10                               DataHandler file);
11     public boolean deleteSession(int SupportiveSession);
12     public int createSupportiveSession(int MainSession);
13 }

```

FIGURE 18.10 DEVS-wrapper AXIS-port services.

Simulation messages are passed into those stubs as SOAP attachments in the form of XML documents.

Stubs are constructed from the deployed WSDL document by the service provider (other DEVS domains). To support SOAP-based Web-services, each DEVS domain should have the following engines:

- HTTP Server (e.g., Tomcat [8])
- SOAP Engine (e.g., AXIS [9])
- XML parser: The proposed protocol is not making any assumptions regarding a specific XML parser

A DEVS-Wrapper is actually a web-service port connected to the AXIS SOAP engine that encapsulates the necessary operations, allowing different DEVS domains to communicate with each other and clients to activate DEVS simulation services. The standard here assumes that individual DEVS domains provide their own interfaces for clients allowing them to invoke the services of the individual DEVS domains (such as user authentication, file submission, etc.). The remaining discussion in this section focuses only on the communication between various DEVS domains.

Figure 18.10 shows the AXIS Java interface for the DEVS-wrapper operations. They are described in WSDL documents, so that other AXIS Java communication classes may be constructed by various DEVS domains (these operations are stubs that allow DEVS domains to communicate with each other).

The DEVS-Wrapper services in Figure 18.10 can be summarized as follows:

- *retrieveResultFile*: It is used to retrieve simulation result files from a support DEVS domain.
 - Input: The support DEVS domain session number
 - Output: The results file
- *startSimulation*: It starts the simulation on a support DEVS domain. In this case the simulation engine starts and waits for the first simulation message from the main DEVS domain.

- Input: The support DEVS domain session number
- Output: True on success; false otherwise
- *isSimRunning*: It checks if a simulation is running on a DEVS domain.
 - Input: The support DEVS domain session number
 - Output: True if simulation is running. Otherwise false
- *StopSimulation*: It stops the simulation normally on a support DEVS domain.
 - Input: The support DEVS domain session number
 - Output: True on success. Otherwise false
- *setDEVSEXML*: It sends an XML document to a DEVS domain. The XML document is either a configuration file or a simulation message.
 - Input:
 - A DEVS domain session number
 - XML document file name
 - The actual XML file
 - Output: True on success; false otherwise
- *deleteSession*: It deletes a simulation session on a support DEVS domain. This operation releases all resources used for this session.
 - Input: The support DEVS domain session number
 - Output: True on success. Otherwise false
- *createSupportiveSession*: It creates a simulation session on a support DEVS domain. This operation allocates all necessary resources needed for this session. The support and main DEVS domains are accepted to bind both session numbers together. This gives each DEVS domain the freedom to allocate its session numbers without worrying about possible conflict with other DEVS domain session numbers.
 - Input: The main DEVS domain session number
 - Output: The support session number or -1 on failure

Initially, the main DEVS domain creates support simulation sessions and establishes full connections with all support domains. Each DEVS domain should know each participant DEVS-Wrapper port URI and its associated session number for all participant DEVS domains. This allows DEVS domains to have multiple concurrent simulation sessions using the same web-service port.

The main DEVS domain opens a session with all relevant support domains, and it broadcasts this information to support domains in one XML document (using the method *setDEVSEXML*). The simulation session document contains the main domain session number (which all support DEVS domains know upon invoking *createSupportiveSession* by the main domain), and the support URIs paired with their session number, as shown in Figure 18.11. After receiving the XML document in Figure 18.11, each domain should be able to send messages on a session to any other domain.

The main principle followed here is to enclose all various DEVS domain heterogeneous models within a single coupled model. This simplifies the simulation coordination, as each DEVS domain hides its internal activities and coordinates with other DEVS domains. This approach has been adopted in the variant of DEVS/

```

<Sessions ver = "1.0">
  <Session Type = "Main">
    <Number>123</Number>
    <URI>http://...</URI>
  </Session>
  <Session Type = "Supportive">
    <Number>1000</Number>
    <URI>http://...</URI>
  </Session>
  ...
</Sessions>

```

FIGURE 18.11 Domain-simulation sessions XML binding document example.

SOA developed by Seo and Zeigler [11] and Seo [12], which supports interoperability across different web-service platforms using the XML namespace concept to be described next.

18.3.3 MODEL STRUCTURE XML DOCUMENT

The Model structure XML document (shown in [Figure 18.12](#)) is initially submitted by the modeler to the main DEVS domain to describe how the overall model is structured so that each DEVS version can identify which models belong to its domain. Further, from this document, the main machine can identify the participant support domains.

The model structure document contains enough information to allow different domains to create local models, coordinators (i.e., coupled model processor), and simulators (i.e., atomic model processor). It also includes data on how they will relate to other models in different domains.

The main DEVS domain must pass this document before it starts the simulation (i.e., before invoking service *startSimulation* on support domains). The model structure document contains the following information (see [Figure 18.12](#)):

- Model names
- Model type (coupled/atomic)
- Model input/output ports
- Coupled models internal submodels and their ports connections
- Models domain URIs
- Coupled models synchronization algorithms used (e.g., the Head/Proxy Coordinator discussed earlier)

The DEVS models hierarchy can easily be mapped into this XML document. For example, assume two models connected with each other as in [Figure 18.7](#) (two DEVS domains where each model is specific to its domain implementation). In this case, the two models would be enclosed within an outer model (*Coupled*), resulting in the XML document shown in [Figure 18.12](#). This XML document also serves as an agreement contract between various implementations on the used synchronization schemes. For example, the coordination scheme that is used can be set by the COUPLED_SYNC

```

<MODEL_STRUCTURE ver = "1.0">
  <COUPLED_SYNC>
    <scheme ver = "1.0">HeadProxy</scheme>
  </COUPLED_SYNC>
  <Models>
    <Model Type = "Coupled">
      <Name>Coupled0</Name>
      <Components>
        <Name Type = "Coupled">Coupled1</Name>
        <Name Type = "Coupled">Coupled2</Name>
      </Components>
      <URI>http://...</URI>
      <LINKS>
        <LINK>
          <FROM>
            <Component>Coupled1</Component>
            <Port>OUT1</Port>
          </FROM>
          <TO>
            <Component>Coupled2</Component>
            <Port>IN2</Port>
          </TO>
        </LINK>
        ...
      </LINKS>
    </Model>
    <Model Type = "Coupled">
      <Name>Coupled1</Name>
      <Ports>
        <Port Type = "in">IN1</Port>
        <Port Type = "out">OUT1</Port>
      </Ports>
      <URI>http://...</URI>
    </Model>
    <Model Type = "Coupled">
      <Name>Coupled2</Name>
      <Ports>
        <Port Type = "in">IN2</Port>
        <Port Type = "out">OUT2</Port>
      </Ports>
      <URI>http://...</URI>
    </Model>
  </Models>
  ...
</MODEL_STRUCTURE>

```

FIGURE 18.12 XML model structure document example.

field to simulate a distributed coupled model across various domains. In this way, the standard can easily adopt any new schemes that may appear in the future.

The simplest way of structuring a DEVS model is to have one coupled model at each of the DEVS domains connected to one other via their input/output ports, where each coupled model views the coupled models in other domains as “black boxes.” Even with this simple scenario, another top-coupled model should then be created to wrap all coupled models across the various domains. Therefore, there will be at least one coupled model partitioned across the DEVS domains. By having one Coordinator simulating a single coupled model distributed over the network, it becomes a performance bottleneck (because of the number of messages exchanged between the parent Coordinator and its children). For this reason, we propose adopting a Head/Proxy Coordinator structure. Other algorithms can be adopted if they are scalable and it is needed. The Head/Proxy extends the coordinator concept, as follows:

- **Head Coordinator:** It is in charge of simulating the entire coupled model. It coordinates the internal models that exist in its domain and (via Proxy Coordinators) the other internal models that exist in other domains.
- **Proxy Coordinator:** It acts as an agent on behalf of the Head Coordinator to simulate the internal submodels of a coupled model that exist in its DEVS domain. A Proxy Coordinator passes *all* the unknown messages to its Head Coordinator; however, a Proxy Coordinator usually passes *only one* message to its head Coordinator on behalf of the coupled model internal partitions its domain (which is possibly distributed among different machines in the same domain).

Note that the domain that owns the first internal model as structured in the XML model structure document will create the Head coordinator for the parent and other domains will create proxy coordinators. For example, the domain that owns Coupled 1 in Figure 18.12 creates the Head coordinator for the outer model Coupled while the others create Proxy coordinators. Note further that the main domain always owns the Root coordinator and drives the overall simulation. This is not related to the Head/Proxy algorithm. Therefore, it is possible for the top-level model to have its Head coordinator in a support domain since this depends on how the modeler described it in the XML structure document. However, the modeler should structure the top-level model to have its Head coordinator in the main DEVS domain to be near the Root coordinator in the main DEVS domain for performance reasons.

Using a single Coordinator adds unnecessary overhead if two child simulators want to exchange messages and are running on a machine different from their coordinator. As shown in Figure 18.13, Simulator 3 sends an output message that is to be translated into external message to Simulator 2, which resides on the same machine as its sibling Simulator 3. Therefore, sending this message to the coordinator, it ends up being transmitted twice as remote messages because the coordinator is running on a machine different from the source and destination of the message.

The above-described problem could have been avoided if there is a coordinator responsible for message routing locally in each machine. Therefore, having a proxy coordinator on Machine 2 (in [Figure 18.13](#)) causes the message from

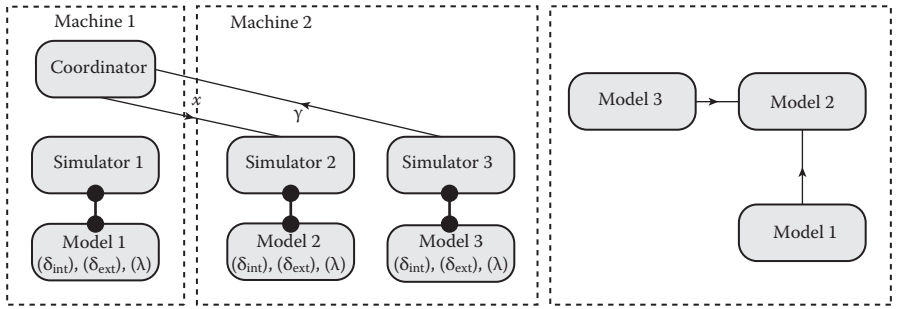


FIGURE 18.13 Superfluous messages exchange in distributed simulation.

Simulator 3 to Simulator 2 to be sent locally, thereby improving the performance of the simulator. Further, one DONE message is sent to the Head Coordinator (on Machine 1) from the Proxy Coordinator (on Machine 2) on behalf of Simulator 2 and Simulator 3.

18.3.4 FORMAT AND CONTENT OF MESSAGES

Simulation messages are constructed as XML documents and sent to other domains as SOAP attachments (using the AXIS stub setDEVSEXML where the receiver session number is one of its parameters). Therefore, any changes in the simulation messages will be made to the message XML document rather than to the input/output parameters of the AXIS stub, thereby increasing scalability and portability.

The simulation message types are listed as follow (note that the specific simulation phases are discussed in the next section):

- *Init (I)*: Simulation starts when the Init message is passed to the top-coupled model Coordinator, which then pushes it downward to its children.
- *Collect (@)*: It is used to start the collection phase. The top model Coordinator propagates it downward.
- *Internal (*)*: It is used to start the transition phase.
- *Done (D)*: It is used by Coordinators to identify which children need to be simulated at this phase. It is used by the Root Coordinator to advance the simulation time and switch simulation phases.
- *External Message (X)*: Messages from the environment or as a result of output messages.
- *Output Message (Y)*: Generated during the collection phase.

Table 18.1 shows all possible fields in an XML message document. All fields are not required to be sent with each message type. However, if the sender chooses to send all fields in a message the required fields (based on the message type) are the only ones that the receiver must consider. The Next-change-time element is used by DONE messages to inform the parent Coordinator about the next expected internal change (in turn, the parent Coordinator passes a DONE message to its parent including the

TABLE 18.1
Simulation Message XML Fields

Element	Format	Allowed Values	Comments
MessageType	Character	I, @, D, X, Y, *	I = INIT, @ = Collect, D = Done, X = External, Y = Output, * = Internal.
Time	String Hours:Minutes:Secs:mSec	Numbers separated by colon (":")	Example: 08:50:00:00
SrcModel	String	Known Model Name	Source Model
DestModel	String	Known Model Name	Destination Model
Port	String	Known Port Name	Destination Port.
Value	C++/Java double	N/A	Mandatory only for External and Output messages.
NextChange	See Time element	See Time element	Next Change Time. Mandatory only for DONE messages.
IsFromProxy	Java boolean	True or False	Mandatory only for DONE messages if Head/Proxy Algorithm is used. This allows Head to synchronize its Proxies.

```
<Message ver = "1.0">
  <MessageType>I</MessageType>
  <Time>00:00:00:00</Time>
  <SrcModel>Coupled0</SrcModel>
  <DestModel>Coupled2</DestModel>
  <Port>IN</Port>
  <Value>-1.0</Value>
  <NextChange>00:00:00:00</NextChange>
  <IsFromProxy>false</IsFromProxy>
</Message>
```

FIGURE 18.14 Initialization simulation message with all fields example.

minimum next change of its model children, whether local or in other domains). Eventually only one DONE message is received by the Root Coordinator (in the main domain), which then starts another simulation phase. All Coordinators (including Root) use this message to know which children branches should be involved in each simulation cycle. This prevents many unnecessary message transmissions across the network.

Figure 18.14 shows an example of an INIT message from the model *Coupled0* to port IN of model *Coupled2*. In this example, the sender domain chose to send all fields; however, the receiver must only use the fields relevant to the INIT message.

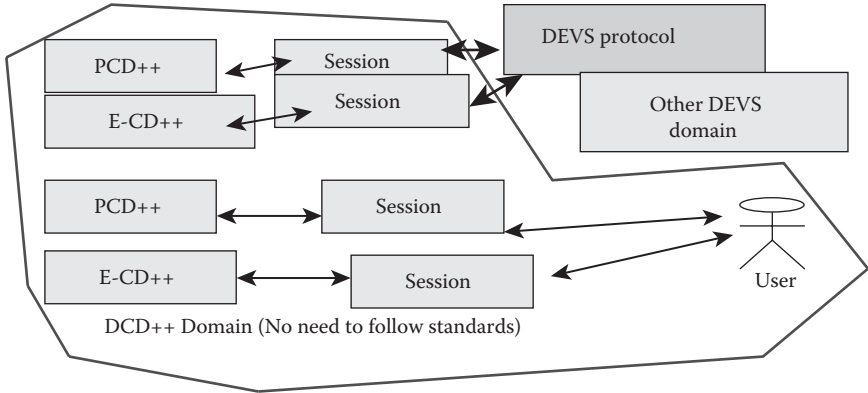


FIGURE 18.15 An internal look of a DEVS domain.

The developed protocol in this section has simplified the simulation by wrapping all distributed models across various DEVS domains in one single coupled model; hence, it becomes the responsibility of coupled Coordinators to locate their children (i.e., internal models) in order to pass them the needed simulation messages (perhaps by having a database that stores each model description along with its domain URI). Further, simulation messages can be specific to a certain domain when they are exchanged within the domain itself, but when they must exit to another domain, the DEVS-Wrapper (discussed in the communication section) translates them to the standardized XML message documents and passes them as SOAP attachments using the AXIS stub `setDEVSEXML` to other domains DEVS-Wrappers. For example, as shown in Figure 18.15 a DEVS domain does not need to use the standards within its domain. However, when a message must travel to another domain, it has to be translated first to the standard format so that it can cross the DEVS protocol bridge.

18.4 SHARED ABSTRACT MODEL

In Wutzler and Sarjoughian [13], an approach for online model-based interoperability named the *Shared Abstract Model* (SAM) was defined. The SAM is used to specify an *Abstract Model Interface*, shown in Figure 18.16. The specification of the Abstract Model Interface is based on the DEVS Atomic Model formalism. The interface is specified in terms of OMG-IDL, and it is executed using CORBA. Based on the Abstract Model Interface definition, the SAM approach requires a Model Proxy (corresponding to the atomic stub of Figure 18.16). Then, models have to be wrapped into model adapters that will make their interfaces match the standard one. This approach is predominantly aimed at integrating existing legacy models or models specified in different DEVS implementations (and for simulation engines responsible for executing a nonnative model implementation). Writing the model adapters for the SAM can be a tedious task, notably because of the use of generic messages that have to be converted before being processed by the model. Fortunately, this task of writing the two adapters (one for atomic and one for coupled models) must be done only

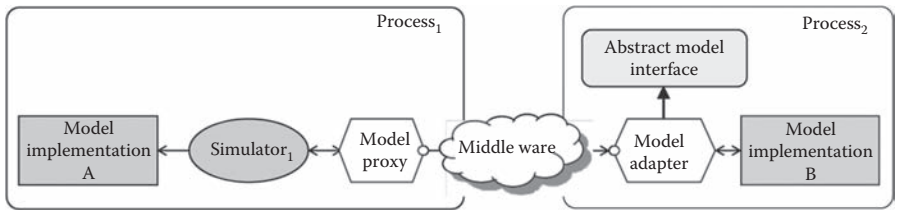


FIGURE 18.16 The shared abstract model concept for a simulator executing nonnative models using proxy model, adapter model, and the abstract interface model.

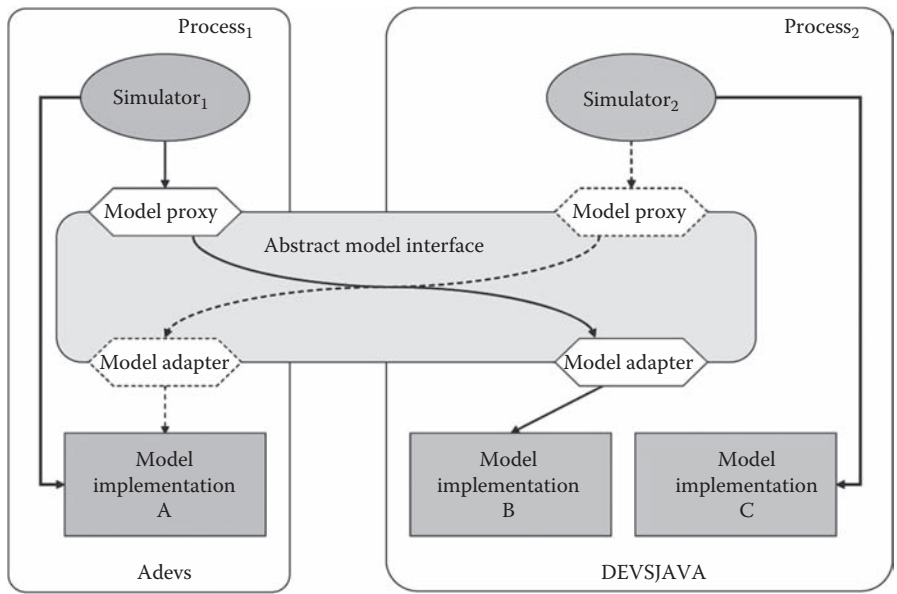


FIGURE 18.17 Example of the shared abstract model for ADEVS and DEVS-suite.

once for each DEVS engine implementation. The two adapters can then be used for all models specified within the DEVS engine implementation.

Considering the DEVS-Suite simulation engine, its simulator (Simulator₁ in Figure 18.17) can directly execute Model Implementation A (i.e., there is no need for syntactic translation from one programming language to another). However, the same simulator cannot execute the Model Implementation B, which is implemented for direct execution using the ADEVS simulator (Simulator₂ in Figure 18.17). The Model Proxy and Model Adapter are used to overcome the syntactical differences between the Java and C++ programming languages that also necessitates Process₁ and Process₂ to communicate with one another (i.e., send and receive messages). The Model Proxy translates the method invocations of the Simulator₁ to those of the Abstract Model Interface. The Simulator₁ then can use the Model Adapter to execute the transition, time advance, and output functions defined for Model Implementation

B. The inheritance relationship from Model Adapter to the Abstract Model Interface allows Model Implementation B and other models that are developed in other programming languages to be uniformly executed using Simulator₁. Therefore, a simulator can execute its own models (e.g., the arrow from Simulator₂ to Model Implementation C), models that are developed for other variants of parallel DEVS simulators (e.g., the arrows from Simulator₁ to Model Implementation B), and any model that can be wrapped inside an atomic DEVS model but does not have its own simulator.

The specifications for the Model Proxy and Model Adapter are straightforward since their operations have a one-to-one relationship to those that are defined for the Abstract Model Interface. However, to support message mappings between different simulation engines, it is necessary to develop modules that can translate one kind of message to another kind. Figures 18.18 and 18.19 show the implementations of the Model Proxy and Model Adapter for the DEVS-Suite. The listings exclude implementations for error handling.

Figure 18.19 shows an example where an Abstract Model Interface is defined for DEVS-Suite and ADEVS simulation engines. A pair of Model Proxy and Model Adapter is defined (shown in solid lines) such that ADEVS Model Implementation B can be simulated using the DEVS-Suite Simulator₁. Using the same Abstract Model Interface with another pair of Model Proxy and Model Adapter (shown in dotted lines), Model Implementation A can be simulated using the ADEVS Simulator₂.

```
interface DEVS { // OMG-idl (CORBA)
    // start of simulation
    double doInitialize()

    // time of next internal transition without input messages
    // value also returned by doInitialize and state
    // transition functions.
    double timeAdvance()

    // produce outputs for current simulation time
    // output does not have any side effect (no state change)
    Message outputFunction()

    // internal state transition without input messages
    double internalTransition()

    // external state transition with input messages
    double externalTransition(in double e, in Message msg)

    // input message is received at the time of internal state
    // transition
    double confluentTransition(in Message msg)
};

Message: bag { inputPort ->value }
```

FIGURE 18.18 The abstract model interface specification.

```

void initialize() {
    ta = devsMod.doInitialize();
    if(ta == devsBridge.DEVS.TA_INFINITY)
        passivate();
    else
        holdIn("active",ta);
}

//External Transition Function
void deltext(double e, MessageInterface x){
    MsgEntity[] msg = trans.devs2CorbaInputs(x);
    ta = devsMod.externalTransition(e, msg);
    if(ta == devsBridge.DEVS.TA_INFINITY)
        passivate();
    else
        holdIn("active",ta);
}

//Internal Transition Function
void deltint() {
    ta = devsMod.internalTransition();
    if(ta == devsBridge.DEVS.TA_INFINITY)
        passivate();
    else
        holdIn("active",ta);
}

//Confluent Transition Function
void deltcon(double e, MessageInterface x){
    MsgEntity[] msg = trans.devs2CorbaInputs(x);
    ta = devsMod.confluentTransition(msg);
    if(ta == devsBridge.DEVS.TA_INFINITY)
        passivate();
    else
        holdIn("active",ta);
}

//Output Function
MessageInterface out() {
    MsgEntity[] msg == devsMod.outputFunction();
    MessageInterface devsMsg = trans.corba2DevsOutputs(msg);
    return devsMsg;
}

```

FIGURE 18.19 Proxy implementation for the DEVS-suite atomic model.

In order for one simulation engine to execute a coupled model that is implemented for by another simulator, it is necessary to also account for coupled models and so Model Adapters for coupled models are needed as well. Rather than specifying the Model Adapter as the coordinator, the Model Adapter is defined based on the Abstract Interface Model (see [Figure 18.18](#)). Examination of the Model Adapter for the DEVS-Suite coupled model enforces the coordinator's logic (see [Figure 18.20](#)). The comments in [Figure 18.18](#) relate the association defined between the Abstract Interface Model and the Model Adapter. This formulation uses the closure under coupling property that allows treating an atomic and coupled model as a basic DEVS model component. Thus, the correctness of the simulation cycle of every

```

//Initialize simulator
double doInitialize() {
    coord.initialize();
    return timeAdvance();
}

//query for time to next event (1. nextTN and 2. outTN)
double timeAdvance() {
    return coord.tN() - coord.tL();
}

// ComputeIO is called (5. applyDelt)
double internalTransition() {
    coord.DeltFunc(coord.tN(), [empty set]);
    return timeAdvance();
}

// ComputeIO is not called (5. applyDelt)
double externalTransition(e, x) {
    coord.DeltFunc(coord.tL()+e, x);
    return timeAdvance();
}

// ComputeIO is called (5. applyDelt)
double confluentTransition(x) {
    coord.DeltFunc(coord.tN(), x);
    return timeAdvance();
}

// 3. getOut and 4. returnOut
MsgEntity[] outputFunction() {
    coord.ComputeIO(coord.tN());
    return coord.getOutputs();
}

```

FIGURE 18.20 Model adapter implementation for the DEVS-suite coupled model.

DEVS coupled model remains legitimate; that is, the executor (either a simulator for atomic models or a coordinator for coupled models) guarantees the correct ordering of events and transmission of events among hierarchical models in concert with the method invocations of the Abstract Model Interface.

18.5 RESTFUL INTEROPERABILITY SIMULATION ENVIRONMENT (RISE)

Interoperating applications that have been developed independently and that interact with each other is not a trivial task, since this interaction involves not only passing remote messages, but also synchronizing them (interpreting messages and reacting to them correctly). This fact further applies to interoperating DEVS-based tools in order to synchronize the same simulation run. The value proposition, however, of such interoperability is that it enables a plug-and-play middleware approach, which

is an appropriate method to interface independently developed software applications [14]. The plug-and-play type of interoperability is already applied by the World Wide Web (WWW) network. The principles of the Web interoperability have been recently called the Representational State Transfer (REST) style [15]. These RESTful Web Services [16] have been gaining attention with the advent of Web 2.0 [17] and the concept of mashups (grouping various services from different providers presented as a bundle).

The RESTful Web-services lightweight approach hides internal software implementation (in “black boxes” called resources). Each resource exposes *uniform* channels (connectors) and describes connectivity semantics between resources in the form of *messages* (usually XML). RESTful services are distributed across a set of connected resources where each resource is named with a URI (similar to a Web site). Service consumers connect with those resources via standardized virtual uniform channels where semantic messages and the corresponding methods are assigned to those resources. In RISE, the channels are the HTTP methods shown in Figure 18.21: GET channel (to read a resource entirely or partially), PUT channel (to create a new resource or update existing data), POST channel (to append new data to a resource), and DELETE channel (to remove a resource). Resources use those channels to transfer their data (or potentially, their data representation) among each other, hence transferring their representational state, as specified by the name of the Representational State Transfer style [15]. REST exposes all services as URIs, hides internal implementation, employs message-oriented synchronization semantics (i.e., XML), and accesses each service (URI) via standardized channels. These are the ingredients for plug-and-play interoperability even at runtime, and they are being used on the WWW every day. A detailed study of the current and future challenges of distributed simulation algorithms and middleware is provided in Wainer and Al-Zoubi [14].

Other approaches, such as CORBA or SOAP-based Web-services, expose functionalities in heterogeneous RPCs that often reflect internal implementation and describe semantics as procedure parameters. The RPC style literally splits software implementation across the distributed environment. It is worth noting that the

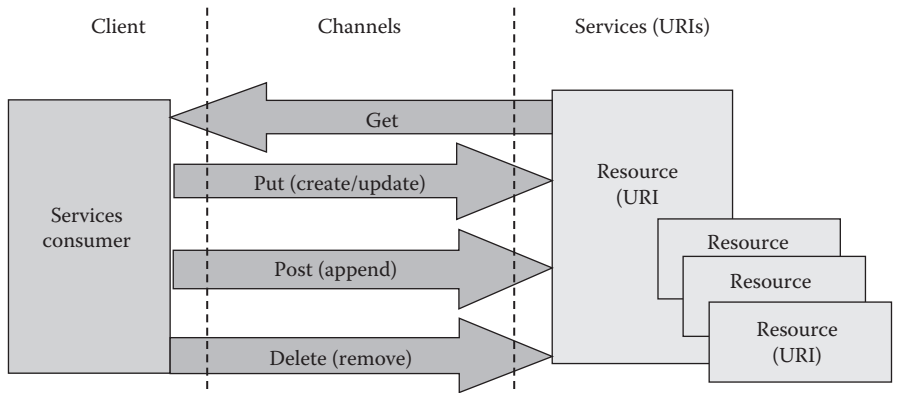


FIGURE 18.21 Uniform channels for RESTful resources.

SOAP-based Web-services transfer all RPC representations (as SOAP XML messages) via the HTTP POST channel. This overloading of the POST channel has resulted in making connectors that were once standardized uniformly into a more heterogeneous interface, which is more complex to use. RPC-style is heterogeneous in a sense that they are programming procedures invented by different programmers. Of course, the XML SOAP standard is powerful enough to describe those RPCs. However, applications interoperability is realized as RPC-style in another software layer above the SOAP handling layer (usually called SOAP engine) that converts RPCs from/to SOAP messages. For example, Figure 18.9 shows a typical SOAP-based Web-services protocol stack while Figure 18.10 shows RPCs exposed within a port.

In recent years a RESTful middleware application has been developed called RESTful Interoperability Simulation Environment (RISE), formally known as RESTful-CD++ [18,19], which has provided promising results in this area. RISE also allows any application or device attached to the Web to be in the simulation loop at runtime, using Web 2.0 mashup concepts. RISE middleware serves as a container to support concrete services; hence, concrete services are plugged into the middleware. In this case, concrete services are wrapped and accessed through URIs at the middleware level, rendering the middleware independent of any specific service. This allows additional services to be plugged into the middleware without affecting other existing services. This is similar to adding additional services or links to a regular Web site. The distributed CD++ (DCD++) simulation package was plugged into the middleware. In this case, multiple CD++ instances can perform distributed simulation session across the Web where the simulation model is split among those CD++ distributed instances, enabling each to simulate its portion of the model, as shown in Figure 18.22. The simulation manager, shown in Figure 18.22, manages a CD++ instance by handling, for instance, the geographic existence of model partitions, XML synchronization simulation messages, and synchronization algorithms. The simulation manager is seen externally as a URI (e.g., similar to Web site URIs). The distributed CD++ instances synchronize among each other via sending simulation XML messages (wrapped in HTTP envelopes) to each other's URIs via an HTTP POST channel. RESTful DCD++ is described in Al-Zoubi and Wainer [18,19].

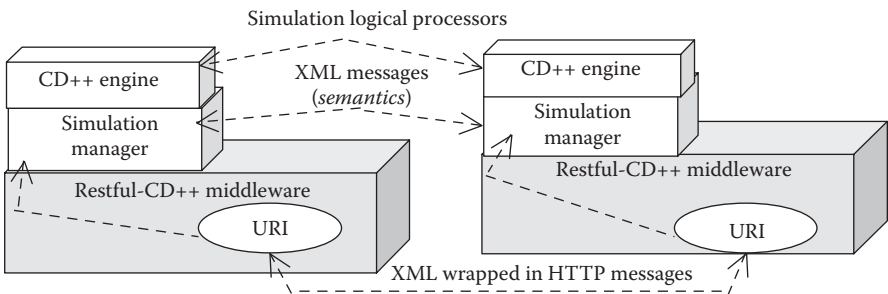


FIGURE 18.22 DCD++ simulation session between two online simulation engines.

RESTful applications APIs, including RISE, are expressed as URI templates [20] that can be created at runtime. Variables in URI templates (written within braces {}) are assigned at runtime by clients before a request is sent to the server, enabling clients to name their URIs at the server side. For example, username in template `<../users/{username}>` can be substituted with any string to obtain the actual URI instance (such as `<../users/user1>` or `<../users/user2>`). Further, URIs may include query variables to define the request scope by appending them to a URI after the question mark “?”. For instance, a request via the GET channel to URI `<http://www.google.com/search?q=DEVS>` would instruct the Google search engine to return information only about keyword “DEVS.” As another example, RISE middleware [18,19] defines the simulation framework URI template as `/cdpp/sim/workspaces/{userworkspace}/{servicetype}/{framework}`, where `{userworkspace}` is a specific workspace. The workspace allows users to define their specific URI hierarchy while avoiding naming conflicts. The `{servicetype}` is the selected simulation service (e.g., DCD++), allowing a client to use different services simultaneously. The `{framework}` is the simulation experiment framework; hence, a user may create multiple experiment frameworks that use the same simulation service. The experiment is configurable by its owner, for instance, to have different simulation partitions conduct the same simulation session. To further illustrate, the `<../>/cdpp/sim/workspace/Bob/DCDpp/MyModel` URI indicates that the user workspace belongs to user Bob, and the servicetype is DCDpp (which selects the distributed CD++ engine). The framework is named MyModel, which is the name of the simulation experiment. In this case, the modeler may select a different simulation engine (instead of DCDpp) or a different framework (instead of MyModel), because these variables are assigned at runtime according to the API URI template. Therefore, URI templates enable modelers to name their URIs without being in conflict with other users. The RESTful-CD++ API is fully described in Al-Zoubi and Wainer [18].

The RISE standards approach is derived from the lessons learned of the RISE middleware. The RISE standards approach divides the entire simulation space into domains. Each domain wraps a DEVS model and DEVS-based simulation engine to simulate that model. Each domain is accessed via three URIs (i.e., API) to exchange semantics (i.e., synchronization and configuration) as standardized XML messages. Thus, a domain’s interior is fully hidden, which makes the standard easier to understand and to support. This is because each domain only needs to be able to transmit/handle the standardized XML messages according to the approved rules while they are free to change whatever they need within their domain without affecting other domains. The RISE approach achieves this at three levels: (1) the *interoperability framework architecture* level, (2) the *model interoperability* level, and (3) the *simulation synchronization* level. These aspects are summarized next.

The *interoperability framework architecture level* provides the URI template (API) that allows modelers to create a simulation environment (including distributing simulations, starting simulation, and retrieving results). RISE requires three RESTful resources (URIs) for each domain so that other domains and modelers can use them to setup and conduct simulations. The focus here is on the parts of the URI template that are relevant to the RISE standard. The main functionality of those URIs is left to design for specific domains (the RISE standard may be part

of different services provided by a specific domain). These resources (URIs) are described as follows:

1. **.../{framework}**: Represents a simulation environment in a domain. It is named by the modeler upon creation. The modeler uses this URI to submit all necessary information to execute simulation in that domain such as the simulation model and the RISE XML configuration. This URI is the parent of the other two needed resources described next. This resource uses HTTP channels as follows: The PUT channel is used to create and/or update the resource with the XML configuration document (for instance, interconnections of the different simulation model ports across domains). The DELETE channel is used to remove this resource. POST is used to submit, as a zip file, all necessary scripts related to the model that is supposed to run on this domain. GET is used to read a simulation status on that domain as an XML document.
2. **.../{framework}/simulation**: Represents active simulation in a domain. The modeler uses this URI to start/abort simulation, and to manipulate simulation during runtime. This resource uses HTTP channels as follows: The PUT channel (with a null message) is used to create this resource. DELETE is used to abort simulation. POST is used by simulation engines in domains to exchange simulation XML synchronization messages.
3. **.../{framework}/results**: Is automatically created by a domain upon successfully completing the simulation, allowing retrieval of the simulation results.

The *model interoperability level* provides XML rules for combining different models. This XML document is provided via the PUT channel to resource **.../{framework}**. This is a straightforward step, because of the assumption that each domain contains an entire model with external ports. In this case, the modeler defines an interconnection between ports analogous to a DEVS coupled model. It is worth noting that this is different from RESTful DCD++ in the sense that DCD++ partitions a single model across the distributed environment. On the other hand, the developed approach here is placing an entire model in each domain. This is because it aims at interoperating heterogeneous environments with many implementation differences, and, therefore, the more flexible, practical, and powerful interoperability is achieved when hiding implementation. This makes sense because the heterogeneity devil resides in the software design and implementation details. For example, Figure 18.23 shows two models placed at two different domains. In this case, the model is wrapped in URI **.../{framework}**: The first model URI is **.../Domain1** and the second model URI is **.../Domain2**. In order to conduct different simulation session experiments, different URI frameworks are needed in a given domain. Each model in Figure 18.23 has two external ports connected to the other model ports. This interconnection is shown in the XML document in Figure 18.24. For example, lines 7–10 show the connection link of port OUT1 (at **.../Domain1**) to port IN1 (at **.../Domain2**). The XML document also shows other configuration such as “Type” at line 3 is set to “C,” indicating that the simulation will be synchronized according to a RISE conservative based

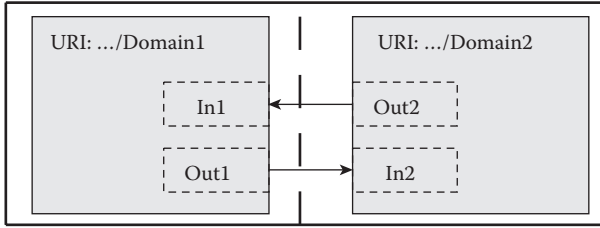


FIGURE 18.23 RISE models interconnection across DEVS domains.

```

1  <ConfigFramework>
2  ...
3  <RISE Version="1.0" Type="C">
4  <Domains>
5  <Main><URI>.../Domain1</URI></Main>
6  <Links>
7  <Link>
8
<From><Port>OUT1</Port><URI>.../Domain1</URI></From>
9  <TO><Port>IN2</Port><URI>.../Domain2</URI></TO>
10 </Link>
11 <Link>
12
<From><Port>OUT2</Port><URI>.../Domain2</URI></From>
13 <TO><Port>IN1</Port><URI>.../Domain1</URI></TO>
14 </Link>
15 </Links>
16 </Domains>
17 </RISE>
18 ...
19 </ConfigFramework>

```

FIGURE 18.24 RISE XML configuration corresponding to Figure 18.23.

algorithm. Likewise, the “Type” attribute can be set to “O” to conduct optimistic synchronization. Line 5 selects the main domain, which is mainly needed to manage the conservative-based simulation.

The *simulation synchronization level* provides high-level simulation algorithms (i.e., conservative/optimistic) and synchronization channels in order to carry out simulation among different domains. The modeler starts the simulation via the main domain (i.e., using the PUT channel to create URI .../framework/simulation). Consequently, the main domain starts simulation, in the same way, on all other domains, as shown in Figure 18.25. Afterward, all simulation engines at different domains are ready to exchange XML simulation messages to synchronize the simulation session. All of the simulation messages are sent to a domain via URI .../framework/simulation using the POST channel.

The conservative-based approach expects the main domain to create the RISE Time Manager (RISE-TM) to manage time advancement of the entire space. This is not required for the optimistic-based approach where domains may directly send

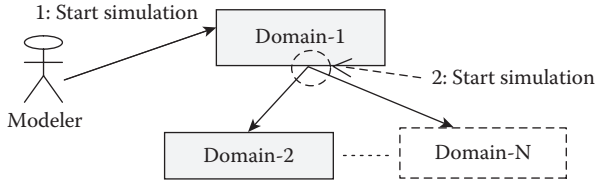


FIGURE 18.25 Starting simulation overview.

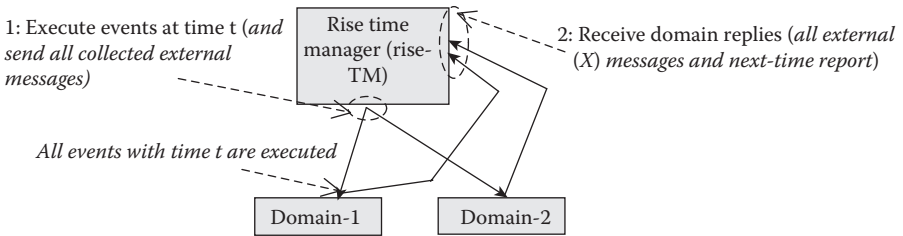


FIGURE 18.26 RISE conservative-based simulation cycle at time t .

messages to each other, provided domains would detect and correct any error because of receiving a *straggler* message. Hiding this detail allows moving algorithm complexity to the interior of domains, while the RISE standard layer simply comprises channels to exchange simulation messages. On the other hand, the conservative approach requires more handling at RISE, since it owns the RISE-TM component. Note that the RISE-TM URI is the same as the main domain URI. Thus, the synchronization between the main domain simulation and the RISE-TM is specific to the internal implementation of the simulation software. However, they are separated in the discussion here for clarity. RISE-TM executes a simulation cycle in the following steps, as shown in

Figure 18.26: (1) Execute all events in all domains at the current RISE time. This starts a new simulation cycle with the current or newly calculated RISE time. RISE-TM always starts the first phase with time zero. The domains must always execute all events with current RISE time, if any, and respond to the RISE-TM with the following information: all external messages generated for other domains stamped with RISE time (or larger) and its next time. The next time is the time of the next event in a domain larger than RISE time. If no more events exist, this value is then set to “-1,” indicating infinity. (2) Once RISE-TM receives all replies from relevant domains, it calculates the next RISE time and starts a new simulation cycle. Further, the RISE-TM merges all generated external messages and passes them to all relevant domains at the beginning of a simulation cycle. Note that the new simulation cycle may be a continuation of the current simulation cycle since external messages may be stamped with current RISE time. Note further that the RISE-TM stops simulation if it calculates a new RISE time to be infinity.

Figure 18.27 shows a domain-2 response message to RISE-TM (i.e., step #2 at Figure 18.26). Line 2 indicates the message source domain URI, hence allowing

```

1  <RISE Version = "1.0">
2  <URI>.../Domain2</URI>
3  <XEvents>
4    <MessagesCount>2</MessagesCount>
5    <XEvent>
6      <Time>00:00:01:000</Time>
7      <Port>IN1</Port>
8      <Value>9</Value>
9      <URI>.../Domain1</URI>
10   </XEvent>
11   <XEvent>
12     ... ..
13   </XEvent>
14   <Time>00:00:01:000</Time>
15 </XEvents>
16 <Next>00:00:03:000</Next>
17 </RISE>

```

FIGURE 18.27 RISE domain XML document response to RISE-TM example.

```

1  <RISE Version = "1.0">
2  <Time>00:00:01:000</Next>
3  <XEvents>
4    <MessagesCount>1</MessagesCount>
5    <XEvent>
6      <Time>00:00:01:000</Time>
7      <Port>IN1</Port>
8      <Value>9</Value>
9      <URI>.../Domain1</URI>
10   </XEvent>
17 </XEvents>
18 </RISE>

```

FIGURE 18.28 RISE-TM message to start a cycle.

RISE-TM to wait for all replies. Lines 3–15 contain all newly generated simulation events by the source domain. For instance, lines 5–10 show a generated event at port *IN1* in *Domain1* with value 9. Line 14 specifies the minimum time of all enclosed external messages from the source domain. RISE-TM must include this time when calculating the next RISE time. Line 16 specifies the time of the next event of *Domain1*. RISE-TM must include this time when calculating the next RISE time. Figure 18.28 shows an example of a message sent by RISE-TM to all relevant domains. In this case, the new RISE-TM is calculated (i.e., line 2), hence all events with this time must be executed at this cycle. Lines 3–17 forward all generated external messages. At this point, it becomes the responsibility of a domain to forward events to appropriate models through specified ports.

18.6 DEVS NAMESPACES

The WSDL for a DEVS simulator service defines data types used by each operation. When the web service communicates with a user, the operations of the web service

receive an argument as an XML document encapsulated in a SOAP message. The XML document is created in conformance with a type of schema in WSDL. The data types in WSDL are only defined for operations of a DEVS simulator not a DEVS model. In the view of simulation, the structure of a DEVS message consists of a set of contents, each of which includes a port name and a value. In the DEVS formalism, values are defined as abstract sets that are not further constrained. Therefore, different DEVS simulation environments can have different class representations and associated object instance representations for values. To overcome this problem, a DEVS message is converted to an XML document at the web service level. This approach requires that different DEVS environments can translate back and forth between their internal value representations and a common XML representation. The *namespace* is the concept in XML that enables services to access a Schema employed by other services and thereby to parse documents to extract data corresponding to the instances of the Schema.

Thus, in order to interoperate DEVS simulator services in different platforms or languages, the namespace concept can be used to provide information about DEVS model messages. This gives rise to the *DEVS namespace* to support interoperability of DEVS simulator services on different web service platforms [11].

The DEVS namespace is an indicator of a schema document for types of messages that are used in DEVS models. The types are expressed in an element of XML Schema that describes the structure of the XML document. XML Schema assigns a unique name to each element. For example, if the name of the element is *Job*, then *Job* element is unique in the schema document. The uniqueness of a type provides clarity for message passing between systems that need to interoperate. Figure 18.29 illustrates the conversion of a language class to a schema type. If a *Job* class is used in the DEVS model, the *Job* class should be expressed as a corresponding schema data type. In the example, the class *Job* has two variables named *id* and *time* that are assigned to *int* and *double* type, respectively. The schema data type represents all variables in the class. The name of the class is the name of a data type and variables become subelements of the data type. The subelements are assigned to primitive data types like variables in the class.

Conversion of a class to a schema is performed by a service provider. For example, in a Java environment, the JAXB library performs this conversion and in addition supports dynamic invocation in which data are bound from a class instance to a corresponding document. The schema document resulting from the conversion is registered into a DEVS namespace storage to access through the network.

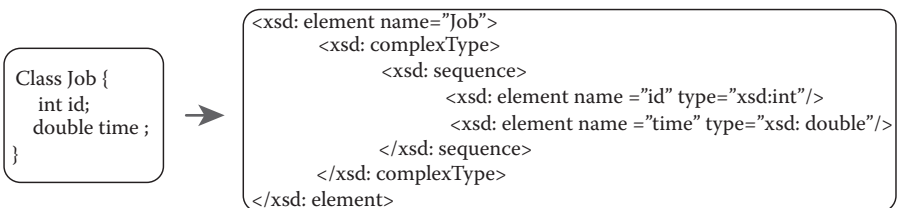


FIGURE 18.29 Conversion of *Job* class to schema data type.

DEVS messages are defined as pairs consisting of a port and a value in the DEVS modeling and simulation. Implementations of the DEVS theory use these pairs to express DEVS messages. That means that the DEVS messages can be converted to a common expression in XML. A common XML message is designed to cover generic DEVS messages.

Figure 18.30 represents the structure of the XML message starting with a *Message* tag. The *Message* tag consists of *content* tags whose elements are a *port* and an *entity* tag. The *entity* tag expresses any object as a message used in the DEVS model. It has a *class* tag containing an identifier for the object. Tags under the *class* tag are created according to the number of variables of the object. The tags have an attribute called *type* describing the type of the variable.

Figure 18.31 represents the conversion of DEVS messages to XML messages and vice versa. A DEVS simulator service consists of DEVS modeling and simulation (DEVS M&S), DEVS interface, and web service. The DEVS M&S handles the DEVS messages, and the DEVS interface converts DEVS messages to XML messages. The web service then generates a SOAP message including the XML messages. This procedure is called serialization. The opposite procedure converts XML messages to DEVS messages and is called deserialization.

Seo [12] created a web service called *NamespaceService* through which a Schema of a DEVS simulator service is registered and browsed. A service provider has the responsibility of registration of a schema. When the provider registers the schema, the provider uses a GUI called schema data register. The GUI has client code for

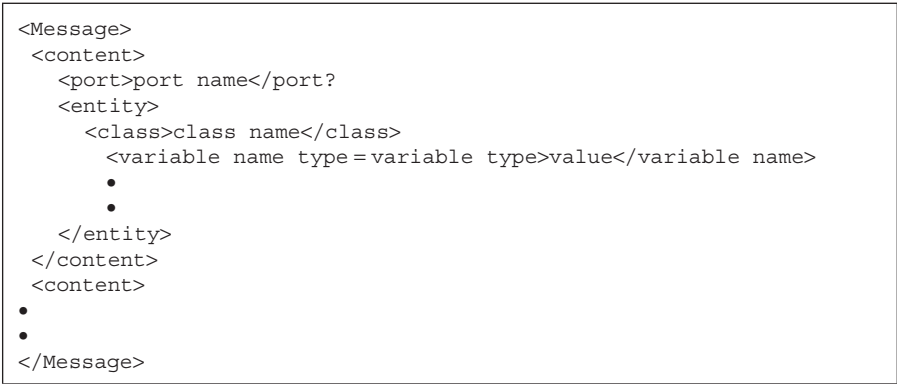


FIGURE 18.30 The structure of the XML message.

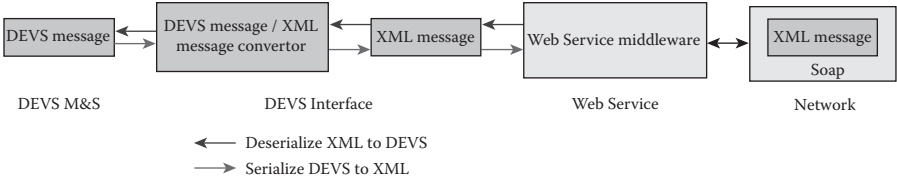


FIGURE 18.31 The DEVS message and XML message in the web service.

NamespaceService web service, which can help easily invoke operations. It displays the response of the operations. Any DEVS developer who uses a Java-based environment or .Net based environment can use the GUI to register a schema. If a developer wants to browse the DEVS namespace storage, the developer can use a browsing GUI consisting of two parts. One part is to display all schema documents in the DEVS namespace storage and the other part is to show the schema document corresponding to the name of the document chosen by the user. This concept can be extended to support browsing based on search criteria such as metadata about which applications have used the schema, the kinds of operations supported, and so on.

18.7 SUMMARY

This chapter introduced different methods to standardize DEVS simulation middleware. This middleware supports interfacing different simulation environments along with synchronization for distributed simulations. Most of the software introduced in this chapter is based on a SOA, with Web-services deployed such that they can be easily invoked by clients.

The DEVS/SOA distributed simulation platform manages distributed simulation through the interchange of XML messages and the publication of web services. The DEVS/SOA simulator is based on a net-centric infrastructure that permits coordinating different atomic models written in varied tools and libraries. The environment provides a well-defined interface for atomic model Simulators and coupled model Coordinators that can be remotely invoked. Client applications are based on DEVSMML models.

DDSP provides interoperability with minimum design changes to the relevant DEVS engines by providing a wrapper SOAP interface and XML messages for simulation synchronization. DDSP hides the internal implementation of the simulation tools (improving interoperability and supporting DEVS legacy models). The protocol implementation uses Web-services to interchange simulation messages, and a DEVS wrapper is employed to call service stubs using a remote procedure call.

The Shared Abstract Model defines an abstract Model Interface based on DEVS atomic models and executed using CORBA. The system uses a model Proxy and a model Adapter to distribute the workload depending on the source and destination of the simulation messages, using the abstract model interface to standardize this communication.

RISE is a RESTful Interoperability Simulation Environment based on Representational State Transfer (REST). RISE uses RESTful Web-services and messages (in XML) to transfer information between simulation engines. Uniform channels are used to expose the simulation resources, based on the methods that are widely used for the World Wide Web (and the HTTP protocol). This approach hides the internal implementation and it uses message-oriented synchronization based on XML.

Finally, we introduced DEVS namespaces, which permit finding the service data types and operations available quickly. The namespace enables the services to access an XML Schema employed by other services and to parse documents to extract data for that Schema. The namespace indicates the type of messages used in the model,

and it assigns a unique name to each element. In order to browse the namespace storage, a namespace service is available to register and browse Schemas for DEVS simulation services.

REFERENCES

1. Strassburger, S., T. Schulze, and R. Fujimoto. "Future Trends in Distributed Simulation and Distributed Virtual Environments: Results of a Peer Study." *Proceedings of Winter Simulation Conference (WSC 2008)*, Miami, FL, 2008.
2. Boer, C., A. Bruin, and A. Verbraeck. "A Survey on Distributed Simulation in Industry," *Journal of Simulation* 3, no. 1 (2009): 3–16.
3. Mittal, S., J. L. Risco-Martín, and B. P. Zeigler. "DEVS/SOA: A Cross-Platform Framework for Net-Centric Modeling and Simulation in DEVS Unified Process," *Simulation* 85, no. 7 (2009): 419–50.
4. Seo, C., and B. P. Zeigler. "Automating the DEVS Modeling and Simulation Interface to Web Services." *Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, CA. Vista, CA: Society for Computer Simulation International, 2009.
5. Al-Zoubi, K., and G. Wainer. "Interfacing and Coordination for a DEVS Simulation Protocol Standard." *The 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, British Columbia, Canada. New York: IEEE, 2008.
6. Wainer, G., and N. Giambiasi. "Timed Cell-DEVS: Modeling and Simulation of Cell Spaces," in *Discrete Event Modeling & Simulation: Enabling Future Technologies*. New York: Springer-Verlag, 2001.
7. Wainer, G. "Discrete-Event Modeling and Simulation: A Practitioner's Approach." Boca Raton, FL: CRC Press, Taylor & Francis Group, 2009.
8. Apache Tomcat. Available via <http://tomcat.apache.org/> [Accessed July 2008].
9. Apache Axis. Available via <http://ws.apache.org/axis/> [Accessed July 2008].
10. Zeigler, B., and P. Hammonds. "Modeling & Simulation-Based Data Engineering: Pragmatics into Ontologies for Net-Centric Information Exchange." New York: Academic Press, 2007.
11. Seo, C., and B. P. Zeigler. "DEVS Namespace for Interoperable DEVS/SOA." *Proceedings of the 2009 Winter Simulation Conference*, Austin, TX, 2009.
12. Seo, C. "Interoperability Between DEVS Simulators Using Service Oriented Architecture and DEVS Namespace," Doctoral Dissertation, ECE department. Tucson, AZ: University of Arizona, 2009.
13. Wutzler, T., and H. S. Sarjoughian. "Interoperability Among Parallel DEVS Simulators and Models Implemented in Multiple Programming Languages," *Simulation* 83, no 6 (2007): 473–90.
14. Wainer, G., and K. Al-Zoubi. "An Introduction to Distributed Simulation." Chapter 11 in *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*. Edited by C. Banks and J. Soklowski. New York: Wiley, 2010.
15. Fielding, R. T. "Architectural Styles and the Design of Network-Based Software Architectures," Doctoral Dissertation. Irvine, CA: University of California, 2000. Available at: <<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>> [Accessed October 2008].
16. Richardson, L., and S. Ruby. *RESTful Web Services*. Sebastopol, CA: O'Reilly Media, Inc., 2007.
17. O'Reilly, T. *What is Web 2.0?* <<http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>> [Accessed May 2009].

18. Al-Zoubi, K., and G. Wainer. "Performing Distributed Simulation with RESTful Web-Services Approach." *Proceedings of the Winter Simulation Conference (WSC 2009)*. Austin, TX, 2009.
19. Al-Zoubi, K., and G. Wainer. "Using REST Web Services Architecture for Distributed Simulation." *Proceedings of Principles of Advanced and Distributed Simulation PADS 2009*, Lake Placid, New York, 2009.
20. Gregorio, J. *URI Templates*. <<http://bitworking.org/projects/URI-Templates/>> [Accessed October 2008].