Computational Analysis, Synthesis, and Design of Dynamic Systems

Discrete-Event Modeling and Simulation

A Practitioner's Approach

Gabriel A. Wainer

CRC Press
Taylor & Francis Group

Discrete-Event Modeling and Simulation

A Practitioner's Approach

Computational Analysis, Synthesis, and Design of Dynamic Models Series

Series Editor

Pieter Mosterman

The Mathworks
Natick, Connecticut

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

Discrete-Event Modeling and Simulations: Theory and Applications, *Gabriel A. Wainer and Pieter Mosterman*

Model-Based Design for Embedded Systems, *Gabriela Nicolescu and Pieter Mosterman*

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

Discrete-Event Modeling and Simulation

A Practitioner's Approach

Gabriel A. Wainer

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

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-13: 978-1-4200-5336-4 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Wainer, Gabriel A.
Discrete-event modeling and simulation : a practitioner's approach / Gabriel A. Wainer.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4200-5336-4 (hardcover : alk. paper)
1. Computer simulation. 2. Discrete-time systems. I. Title. II. Series.

QA76.9.C65W35 2009
003'.3--dc22 2008039739

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

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

Contents

SECTION 2 Building Simulation Models: The CD++ Toolkit

SECTION 3 Applications

SECTION 4 Simulation and Visualization

Foreword

Modeling and simulation (M&S) is finally coming into its own as a discipline, a technology, and an industry. The need is now evident for a textbook that can serve as an introduction to the field that is in the process of "becoming." Such a text has to lay out the foundations of M&S in a clear and concise manner without having to provide the rigor that was needed in the theory as it was developed because this theory is accessible to readers in the original works. Further, the text should offer a wide variety of applications that suggest the unique power of M&S to tackle the kinds of complex, multidisciplinary problems that are increasingly demanding global attention.

The book that you have before you admirably satisfies the preceding criteria. It is divided into sections on introductory background, applications, and technical exposition. The background section first introduces the key concepts in M&S from the point of view of the classic text *Theory of Modeling and Simulation,* as well as other foundational texts. It then goes on to present the basic concepts of discrete event system specification (DEVS) and CD++, the author's implementation of a cellular space DEVS simulation environment. The application section is notable for its wide variety of examples covering physical and life sciences, business, and engineering domains—all developed within the CD++ framework. The technical exposition covers areas such as simulator construction and execution, visualization of simulation output, and the like.

As a teacher, you will be well supported by this book. I suggest that you start with the introductory chapters and then select one application area upon which to base a detailed exposition of the concepts in a form that students can more easily grasp. The area chosen should depend on the backgrounds that students bring with them from their academic degrees. After setting this foundation, you can discuss or have students read other examples from the panoply available in the book. I would be sure to have them start on projects to apply the methodology to areas of interest to them. Finally, I suggest going on to cover some of the more technical issues to a depth that corresponds to the students' interests and backgrounds.

Although the book is primarily intended for graduate students, it can also be used in upper-level undergraduate courses. The availability of the CD++ software, with its graphical and visualization support, is a major advantage enabling students to graduate with the necessary skills to make creative and productive contributions to the exciting emergence of the modeling and simulation discipline, technology, and industry.

Bernard P. Zeigler
Tucson, Arizona

Preface

Scientists, engineers, and practitioners of many professions have long relied on the creation of models to understand the phenomena they study. Traditional mathematical methods (i.e., differential equations) have been used for centuries as the main tool for analysis, comprehension, design, and prediction for complex systems in varied areas. However, these methods appeared unsuitable for studying the complex human-made systems developed during the twentieth century. In the same vein, the many models applied to the study of natural systems have brought about scientific knowledge that, in turn, has posed new, complicated questions that could not be answered by analytical methods.

The emergence of digital computers provided alternative methods of analysis for both natural and artificial systems. Since the early days of computing, users translated their analytical models into computer-based *simulations* (i.e., the execution of those models with particular sets of data using a computing device). This approach allowed solving problems with a level of complexity unknown in earlier stages of scientific development. Computer-simulated models also have additional benefits: they can be executed safely, and experiments can be easily repeated in a cost-effective, risk-free environment and are thus well suited for training purposes.

Computational methods based on differential equations could not be easily applied in studying human-made dynamic systems (e.g., traffic controllers, robotic arms, automated factories, production plants, computer networks, VLSI circuits). These systems are usually referred to as *discrete-event systems* because their states do not change continuously but, rather, because of the occurrence of events. This makes them asynchronous, inherently concurrent, and highly nonlinear, rendering their modeling and simulation different from that used in traditional approaches.

In order to improve the model definition for this class of systems, a number of techniques were introduced, including Petri Nets, Finite State Machines, min-max algebra, Timed Automata, etc. The classic bibliography of this field (which includes, for instance, references 1–3) and other advanced literature (for instance, references 4–6) have thoroughly discussed these and related techniques in detail.

In this book we will mainly focus on one of the existing theories of modeling and simulation called DEVS (discrete-event system specification) [7,8]. Defined by B. Zeigler in the 1970s, the DEVS formalism allows the modular description of discrete-event models that can be integrated using a hierarchical approach.

Although the classic literature on discrete-event systems thoroughly covers theory and methods, there is a need to bridge theory and practice, allowing practitioners in different domains to create discrete-event models and simulations easily. Although many domain experts (and newcomers to the field of modeling and simulation) might know the basics about modeling and simulation theory (including discrete-event methodologies), they usually lack the skills and expertise to create the advanced models needed to study interesting problems. Thus, the focus of this book is to bridge this gap between theory and practice for discrete-event systems.

In order to achieve our goals, we rely on the use of the CD++ modeling and simulation environment [9], an open-source framework that enables simulation of discrete-event models (with specialized support for cellular models). Although we introduce the generics of the underlying theory, we focus on the creation of a variety of models in different areas of interest while giving details on how to create advanced simulation engines to execute these models. We also show how to build independent graphical user interfaces that have been built as open-source (and using varied three-dimensional rendering technologies). Experts in modeling and simulation can use this book as a

companion to the theoretical literature in the field, and the practitioners can focus on the practical aspects and the example applications.

The book is organized in four different parts. The "Concepts" part gives a general perspective on discrete-event modeling and simulation and a brief description of the DEVS and Cell-DEVS formalisms. Part II, "Building Simulation Models: the CD++ Toolkit," introduces the definition of DEVS models using CD++. This part starts by introducing the features of CD++, showing how to create basic models with the tool. It then concentrates on models of generic discrete-event systems, followed by a description of how to map different modeling techniques to DEVS (i.e., Petri Nets, Finite State Machines, Bond Graphs, Modelica, Layered Queuing Networks, and VHDL). We show several examples of the definition of such techniques in DEVS and discuss how to implement them in a discrete-event simulator such as CD++. Part III, "Applications," starts with a chapter on biology and medicine and then moves to defense and emergency planning, after which it discusses architecture and construction, environmental sciences, physics and chemistry, artificial systems, and urban traffic. The last part of the book, "Simulation and Visualization," elaborates on the creation of simulation software for DEVS models and analyzes the creation of three-dimensional visualization environments associated with these tools.

A variety of resources has been made available online to be used by the reader. Basic information on CD++ can be found at http://cell-devs.sce.carleton.ca, where the reader will find a complete user manual, installation tools, and the software application. It also includes links to an open-source version of the software (interested developers are encouraged to participate in the development of this project). All the examples discussed in the book can be found in a repository of models available for general use, and we encourage the reader to use such models for learning. Examples in the book are simplified versions of the actual versions found online. The examples focus on the main ideas and skip some of the details in order to make them more understandable by the reader; each model in the repository includes the complete software implementation and extended documentation for the reader.

We intend for the materials introduced to provide the reader with a wide variety of practical experiences, ranging from the creation of models, simulators, visualization tools, and theoretical analysis up to a large number of models for experimentation and open-source tools ready to use and modify. In order to show the feasibility of this approach, we include many examples developed by nonexperts. This allows readers to become familiar with the experience of others with a similar level of expertise, while providing the opportunity to improve and adapt other people's work to their own needs and interests.

This book is the result of years of collaboration with many students and colleagues. The list is so extensive that a separate section of acknowledgments has been included. Nevertheless, I would like in particular to thank Bernie Zeigler, Norbert Giambiasi, and Claudia Frydman (for their constant support) and Pieter Mosterman (for giving the initial stimulus). I owe everything to the love of my family. Carlos and Jacobo Wainer taught me the immense value of books from a very early age. Noemi and Maria Luisa taught me how to read them even earlier. Diana and Ian (who suffered the consequences of those teachings) gave me the most incredible lesson about love and patience every day I spent writing this book.

Gabriel A. Wainer
Ottawa, Canada

REFERENCES

1. Cassandras, C. G. 1993. *Discrete event systems: Modeling and performance analysis.* Homewood, IL: Aksen: Irwin.
2. Banks, J., J. S. Carson, B. L. Nelson, and D. Nicol. 2005. *Discrete-event system simulation,* 4th ed. Upper Saddle River, NJ: Prentice Hall.
3. Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis,* 3rd ed. Boston: McGraw–Hill.
4. Fishwick, P. A. 1995. *Simulation model design and execution: Building digital worlds.* Englewood Cliffs, NJ: Prentice Hall.
5. Cellier, F. E., and E. Kofman. 2006. *Continuous system simulation.* New York: Springer Science+ Business Media.
6. Toffoli, T., and N. Margolus. 1987. *Cellular automata machines: A new environment for modeling.* Cambridge, MA: MIT Press.
7. Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation,* 2nd ed. New York: Academic Press.
8. Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.
9. Wainer, G. 2002. CD++: A toolkit to develop DEVS models. *Software Practice and Experience* 32:1261.

The Author

Gabriel Wainer received the MSc (1993) and PhD degrees (1998, with highest honors) from the University of Buenos Aires (Argentina) and Université Paul Cézanne (Aix-Marseille III, France). In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, Ontario, Canada), where he is now an associate professor. He has held positions at the Computer Science Department of the University of Buenos Aires and visiting scholar positions in numerous places, including the University of Arizona, Ecole Polytechnique de Marseille, CNRS, University of Nice, and INRIA Sophia-Antipolis (France). He is the author of a book on real-time systems and another on discrete-event simulation and of more than 190 research articles. He has collaborated in the organizing of numerous conferences in the area.

Dr. Wainer has been principal investigator of different research projects (funded by the National Science and Engineering Research Council of Canada, Precarn, Usenix, the Canadian Foundation of Innovation, CANARIE, and private companies, including Hewlett-Packard, IBM, Intel, and MDA Corporation). He has been the recipient of various awards (including the Carleton University Research Achievement Award, IBM Eclipse Innovation Award, and the Leadership Award by the Society for Modeling and Simulation International—SCS, and CICC, Japan). Dr. Wainer is a member of the real-time and distributed systems lab at Carleton University and the head of the Advanced Real-time Simulation lab within Carleton University Center for Advanced Visualization and Simulation (V-Sim).

He is Special Issues editor of *Simulation Transactions of the SCS* and associate editor of the *International Journal of Simulation and Process Modeling.* He has been a member of the board of directors of the SCS and a chairman of the DEVS standardization study group (SISO). He is director of the Ottawa Center of The McLeod Institute of Simulation Sciences and chair of the Ottawa M&SNet.

Dr. Wainer's current research interests are related to modeling methodologies and tools, parallel/distributed simulation, and real-time systems.

Acknowledgments

The work presented in this book has been the result of the efforts of numerous students and collaborators. Although they have been acknowledged in the references and the authors of the models can be found on each model file, here we include the list of individuals who participated in the creation of the different tools and models.

Qi Liu, Amin Hammad, Hui Shang, Ezequiel Glinsky, Rami Madhoun, Dorin Petriu, Shaylesh Mehta, and Hesham Saadawi co-authored different sections.

Different collaborators have participated in the discussions that led to the creation of some of the work presented here, including some the applications: James Cheetham, C. Anthony Hunt, Michael Jemtrud, Ernesto Kofman, Tofy Mussivand, and Trevor W. Pearce. Other colleagues whose thoughts and discussions were of influence include Rod Bain, Olivier Dalle, Chris Herdman, Tag Gon Kim, Gabriela Nicolescu, James Nutaro, Tuncer Ören, Glen Ropella, Mamadou K. Traoré, Hans Vangheluwe, and Yuhong Yan.

Many students were fundamental in the creation of the software tools, in particular, Javier Ameghino, Amir Barylko, Jorge Beyoglonian, Shannon Borho, Wenhong Chen, Gastón Christen, Juan Cidre, Mariana D'Abreu, Alejandra Davidson, Alejandra Díaz, Alejandro Dobniewski, Ezequiel Glinsky, Marcelo Gutiérrez-Alcaraz, Christian Jacques, Kiril Kidisyuk, Qi Liu, Mariana Lo Tártaro, Alejandro López, Rami Madhoun, Alexandre Muzy, Jan Pittner, Daniel Rodriguez, Hui Shang, César Torres, Alejandro Troccoli, Verónica Vázquez, Wilson Venhola, and Yinfeng Yu. Students in our lab who collaborated in the applications include Khaldoon Al-Zoubi, Rodrigo Castro, Rachid Chreyh, Bo Feng, Rhys Goldstein, Shafagh Jafer, Leandro de San Miguel, Ayesha Khan, Mohammad Moallemi, Emil Poliakov, and Hesham Saadawi.

Finally, a large number of students created the models presented in the book and found in the model's repository. A noncomprehensive list includes T. Adams, A. Adidharma, M. Ahmed, B. Al-Aubiydy, D. Altman, B. Balya, A. Baranek, P. Barletta, J. Barrionuevo, A. Bender, P. Bendersky, X. Bing, Y. Boiko, M. Braunstein, D. Brignardello, M. Brunstein, A. Calvo, A. Campbell, J. Cao, S. Chao, J. Chazal, L. Checiu, A. Corvetto, P. Cremona, S. Daicz, F. Dellasoppa, L. De Simoni, C. Delannoy, P. Demidoff, A. Dias, W. Ding, C. Diuk, R. Djafarzadeh, M. El-Salam, A. Elshafei, S. Enrique, L. Fal, U. Farooq, E. Fernandez Rojo, J. Galaski, C. Gnanapragasam, A. Gonzalez, D. Grinberg, J. Hayes, M. Hussein, F. Iñón, R. Kirkner, R. Klett, K. Lam, S. Leon, L. Li, C. Lim, S. Lombardi, J. Long, M. MacLeod, P. MacSween, V. Mahendran, M. Mansour, S. Mehta, C. Miracola, A. Monadi, O. Muhi Kanwar, M. Núñez Cortes, J. Ogasawara, R. Ortiz, H. Pang, T. Pendergast, F. Petronio, D. Petriu, J. Pittner, M. Polimeni, L. Quinet, N. Rehman, M. Ricillo, D. Rubinstein, S. Sim, P. Sor, W. Sun, G. Thezier, S. Thurairasa, K. W. Tsui, G. Vasconcelos, D. Wassermann, P. Wu, X. Wu, P. Yeon, K. Yonis, R. Youssef, Y. Zhang, C. Zhang, X. Zhao, T. Zheng, and S. Zlotnik.

Section 1

Concepts

1 Modeling and Simulation Concepts

1.1 INTRODUCTION

Human beings are resourceful and curious. The need to explore, analyze our surroundings, and solve problems is in our nature (even from a very young age). There are many different reasons why we do these things, including improvement of our quality of life (e.g., by creating devices to reduce efforts or improve our safety), thirst for knowledge (e.g., in learning how plants grow), or even just for fun.

The first technique humans use to learn about (and maybe change) their environment is *experimentation* (for instance, if we want to learn how much clay and water must be mixed to do pottery, we conduct different trials until the desired consistency is obtained). Experimentation was the only way humans had to learn about their environment for thousands of years, and it is still one of the principal methods employed in problem solving (and a crucial part of child development). Figure 1.1 shows a basic scheme for experimentation.

There are two objects under consideration: the entity under study and the experimental frame (EF), which defines the conditions for experimentation. The EF defines not only how we experiment on the entity but also how we obtain the experimental results. In our pottery example, the entity is the mix of water and clay (that we can mold and then solidify in an oven), and the EF is the set of experiments done. Each experiment would be a different trial to mix clay and water (including different percentages of each material, varied temperatures of the mix, duration of the experiment, etc.). The experiments' results would include the consistency and texture expected and obtained for each different clay mix.

EXAMPLE 1.1

Let us suppose that we want to study the best possible allocation of desks in a classroom, in order to reduce energy costs. We need to decide where to put the desks and the heating/air conditioning sources. An experimentation-based solution would take different groups of students in different positions and would use a sensor (i.e., a thermometer) to measure the temperature in different areas in the classroom. In this example, the *entity* is a classroom and its temperature under different desk configurations. The *EF* is defined by the multiple student configurations (*experiments*) and the kinds of results expected at the end of each experiment (in this case, temperature of a room between –20 and +45°C). The *results* are provided by the thermometers used to measure the temperature. As we can see, the EF allows us to consider what the objectives of the experiment are (measure temperature in the room; the number of students or their weight is not of interest) and any assumptions we have about the experiment (do we use a digital thermometer or an analog one? One or many? Are we interested in fractions of a degree?).

Unfortunately, the problems we need to tackle are usually much more complex than learning how to mix the materials for making pottery or measuring the number of students and the temperatures in a classroom. In many cases, experimentation is not a feasible solution due to ethics, risks (e.g., we cannot study spread of an epidemic or fire evacuation in the classroom), or cost (we cannot study every possible configuration in the classroom because scheduling a study with a large number of

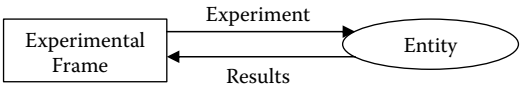


FIGURE 1.1 Problem solving through experimentation.

individuals for a long time can be costly). In other cases, experimentation is simply not possible (for instance, we cannot manipulate a star to understand its gravitational field; we cannot experiment on our classroom if the building still does not exist). However, in many cases, this kind of analysis is what we need.

Humans have found different ways of dealing with these issues. One of them is to abstract from the problem itself and then reason about it using a *model* of the problem. You might have started using this technique while thinking about Example 1.1, creating a mind picture of the room, students, sensing and heating devices, etc. You might even have started thinking about different student distributions and mechanisms to improve the heating according to their location, window positions, and building orientation. You might have sketched your ideas on paper, even using a scale model. Humans are well prepared (and trained from childhood) to create these models in a very natural way; they help us to think better about the problem we want to study.

Although different modeling techniques have been proposed, during the last 300 years, the *differential equation* formalism proposed by Newton and Leibniz has been the tool of choice for modeling and problem solving [1]. Differential equations provide a formal mathematical method (sometimes also called an analytical method) for studying the entity of interest. For instance, in Example 1.1, we could try to use Fourier's law [2] to study the temperature on the classroom's floor. These equations define the conduction of heat in a one-dimensional steady state isotropic as

$$q_x = -k\frac{\partial T}{\partial x}$$

Here,

q_x = the heat flux;
k = the thermal conductivity of the material under study (W/m°C);
T = the temperature field in the medium; and
$\partial T/\partial x$ = the temperature gradient (the minus sign indicates that the direction of heat flux is opposite the direction of increasing temperature).

When we try to solve problems through formal modeling (using, for instance, differential equations), the scheme previously presented in Figure 1.1 must be extended as shown in Figure 1.2. In this case, we can still do experimentation to obtain data about the entity under study, and we use such data to create a model of the entity (using differential equations or other analytical techniques). We use the model's experimental frame to put a context on the model's creation, indicating the kinds

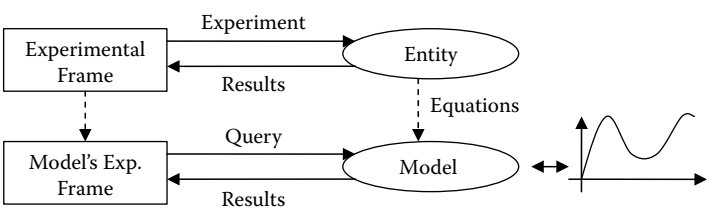


FIGURE 1.2 Problem solving through analytical modeling.

of questions we can ask and the assumptions we have made. For instance, in Example 1.1, we cannot find the average weight of the students (as we have decided to apply Fourier's equations), and we have assumed the use of a one-dimensional equation to approximate the temperature in the room. The model's EF also tries to mimic the experiments carried out on the original entity to obtain the desired results. For instance, in Example 1.1, we want to sense temperatures (we are not interested in social interactions between class students) and the effect of the number of students and their distribution in the room's temperatures. The results observed can be used, in turn, to modify the original entity (in this case, the classroom under study). Using this approach, we might even skip the initial experimentation step, just theorizing about the system's behavior through a pure model (even without any experimental data). In these cases, as soon as experimental data are available, they can be used to validate or reject the proposed theories.

These problem-solving techniques are analytical in the sense that they are symbolic and based on reasoning, and they try to provide general solutions to the problems to be solved. The idea is that we abstract what we learned about the entity into a model that represents the entity under study. This abstraction implies a loss of information, but it allows us to describe the behavior of the entities, analyze it, and prove properties of the proposed model (for instance, controllability and stability in the control system used to keep the room's temperature). The idea is that, if we are able to solve the equations, we will know the results of every possible experiment to be carried out on the entity. The solution is built using inference rules (which should be correct in the paradigm chosen to describe the model). If we need to obtain particular solutions, we just replace the symbolic values with their corresponding numerical counterparts. In Example 1.1, the Fourier equation allows us to find the solution to the problem for every value of x; if we are able to solve the equation, we will know the exact temperature in every single point (with infinitesimal scale).

Figure 1.2 also introduces an important concept: the behavior of the analytical model should match the one observed in the original entity. In this case, we say that the results given by the model are *valid*. To ensure this, a *validation* phase is required to check that the results given by the model match what we see in the original entity. (If no data are available, validation can be done by trying to match other existing theories or models or by using similar entities in a different context.)

In many cases, we need to do several simplifications in order to define and solve the equations. For instance, in Example 1.1:

- We used a single-dimensional model, although we have a three-dimensional classroom. Adding a second equation to solve the problem in two dimensions results in extra complexity. In this case, we need to add a new equation, $q_y = -k\,\partial T/\partial y$, and modify the temperature field in the medium $T = T(x, y)$ to be a function of both x and y. Solving this equation is more complex, and this is still a simplification of the three-dimensional problem.
- These equations do not consider transient behavior: what happens if students move? What happens if a window is opened or if we turn the heat on? How is the position of the heating device going to affect the equation definition?
- The equations do not consider combinations of the different possible transients, the materials being used in the room's walls and floors, influence of air flowing in the room, and many other simplifications.
- We need to be able to solve the equations of interest, which can be infeasible or complex (moreover, in many cases just finding the equations to describe the entity under study in detail is impossible).

If we are interested in solving a simple problem like the one in Example 1.1, most of the details in reality can probably be ignored. Nevertheless, if we wanted to use Fourier's equations to study the

heat conductivity of a new material that we want to use for manufacturing, some of the details cannot be ignored (otherwise, the model will lose precision, and the results of the study will be incorrect).

In order to deal with models with a higher complexity, *difference equations* and other *numerical methods* were introduced [3,4]. The idea of these methods is to approximate the equation by discretizing their behavior (which, instead of being continuous and thus computable at every point in time, is now calculated at predefined time steps). The result is not analytical but, rather, numerical, and it will have less precision than the formal model (we cannot obtain solutions for every possible combination of the model's variables). Nevertheless, such methods provide approximate values that are close enough for the problem under study. For instance, in the case of Fourier's equations, we could divide the surface of interest into small elements (assuming a linear temperature distribution along its unit length and a unit area perpendicular to heat flow direction) and obtain the following approximation to the Fourier equation:

$$T_1 = \frac{hT_\infty + \frac{K_1}{L_1} T_o}{h + \frac{K_1}{L_1}} \tag{1.1}$$

Here,

K_1 = the thermal conductivity;
L_1 = the length of the element;
T_∞ = the fluid temperature;
T_0 = the temperature inside the material on a node;
h = the heat transfer coefficient; and
T_1 = the computed surface temperature, which will replace T_0 in the next computation step [2].

The problem-solving activities using these methods can now be described as in Figure 1.3.

The cycle also starts by obtaining experimental data from the entity and then creating equations to model the observed behavior (within the corresponding experimental frames). Nevertheless, as in this case, we cannot find a solution for the equations, so we use a numerical *approximation*. The *computation* is based on a recursive method (that will calculate the values of the state variables at a given time and convert the value as the basis for the next computation). In the past, this task, in general, was carried out by a human expert. The *computation EF* in this case is derived from the model's EF; within the same frame, the computation model should be able to answer the same queries (with a loss of precision due to the approximation). In Figure 1.3 a new step is introduced, which means that we need to carry out an extra check in order to verify the correctness of the results. In this case, we will say that we need to do *verification* of the results obtained by numerical approximation. The idea is that the values we compute need to be checked against the experimental data, while trying to mimic the model's description as accurately as possible.

This approach for problem solving (which was the main technique employed in the last few centuries) made possible a tremendous advance of science and technology. A consequence of such success was the evolution of the associated mathematical techniques, which resulted in the ability to attack new and more complex problems, and new questions to answer (which might require new problem-solving techniques).

Besides the evolution in our learning about nature, the twentieth century witnessed the creation of very elaborate human-made devices (e.g., control systems, intelligent manufacturing, traffic monitoring) that made it difficult (or impossible) to continue using "pencil-and-paper" analysis methods. When we consider such problems (with a few exceptions), they are analytically intractable and numerically impossible to evaluate (unless we simplify the model, which, in most cases, results in solutions that are far from reality).

The advent of computers in the 1940s provided scientists and engineers with alternative methods of analysis. Computers are well suited to deal with approximation techniques, reducing human

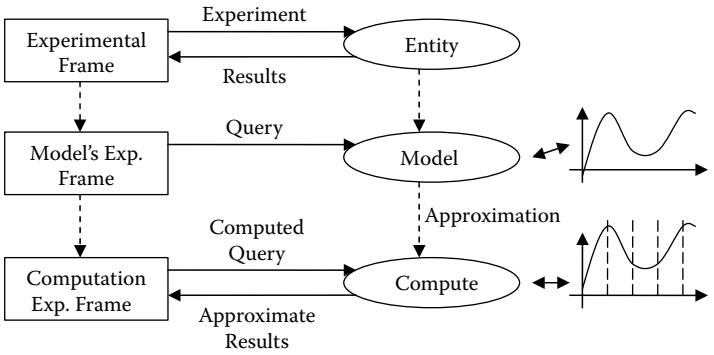


FIGURE 1.3 Problem solving through computation.

computation errors and being able to solve the problems at much higher speed. Thus, since the early days of computing, traditional numerical models were converted into computer-based solutions (and, in many cases, they were called *computer simulations*). The cycle for creating computer simulation (or simply *simulation*) studies can be also represented as in Figure 1.3, with the difference that now the computation model is executed by specialized devices (in the beginning of simulation history, analog computers were used for numerical approximation [5]; today, we use digital computers). The computation EF is now a program that generates test cases for the computational approximation of the model. In this case, the verification activities also need to check the accuracy of the results, while adding an extra step (as limited precision in computers can create erroneous results that can diverge from the expected solutions).

Computer simulation enabled scientists and engineers to experiment easily with "virtual" environments, elevating the analysis of natural and artificial applications to a new level of detail unknown in earlier stages of scientific development and providing great help in the design and analysis of complex applications. Simulated models also can be used for training because they provide cost-effective and risk-free solutions when compared to experimentation. In simulation-based techniques, we find individual solutions for particular problems (as opposed to the general solutions found by analytical methods) using a device (in general, a digital computer) for controlled experimentation and time compression. The constant reduction of the cost of computers (in hand with graphical interfaces, advanced libraries, languages, and other facilities) allowed simulation to become an easy-to-use and flexible technique. Nowadays, modeling and simulation provide a well-developed, well-proven approach to problem solving that advances steadily as more computing power becomes available at less cost.

The advantages of simulation are multiple:

- Decisions can be checked artificially.
- The same model can be reused multiple times.
- Simulations are easier to create and use than many analytical techniques, and they need fewer simplifications.
- The rules used to define the model's behavior can be modified easily.
- During execution of a simulation, we can experiment with varied special cases.
- The user can interact with the simulator, allowing analysis of such interactions.
- Simulation results in shorter design-cycle times and reduced requirements for initial resource investment.
- Simulation provides economic benefits: Research and Development cycles can be improved.
- The original entity is not affected by the study, and it can continue to be used.

In the rest of this chapter, we will discuss some basic ideas about modeling and simulation we introduce in this book. We will present basic concepts of discrete-event dynamic systems, introduce and classify different techniques for modeling such systems, and show how a simulation study is carried out.

1.2 MODELING DISCRETE-EVENT DYNAMIC SYSTEMS

As mentioned in the previous section, computer simulations began by implementing numerical approximations of the differential equations under study with the goal of solving the computation of more complex models in a fast and precise way. Unfortunately, these methods could not be adapted to most of the artificial applications developed during the twentieth century. The Industrial Revolution brought the need to model the behavior of complex apparatuses built by humans (i.e., telephone lines, avionics controllers, automated elevators, etc.), which cannot be adequately described by differential equations or their numerical approximations.

EXAMPLE 1.2

Figure 1.4 shows the observed behavior of a traffic light. Initially, the light is green for 45 s. Then, it switches to yellow for 10 s and finally switches to red for 55 s (after which the cycle is repeated). As we can see, modeling this application with a differential equation is not natural (and it is infeasible to solve in the case of complex combinations including hundreds of traffic lights in a city).

EXERCISE 1.1

Write a model of the behavior of the traffic light in Figure 1.4 using ordinary differential equations [1, 3].

In order to deal with this kind of problem in a better way, new mathematical theories (in particular, those based on automata theory) were applied in the analysis of these automated devices with discrete components [6,7]. For instance, we can model the behavior of the traffic light in Figure 1.4 with a simple untimed automaton like the one presented in Figure 1.5(a). Such a model can be used to think about correctness of the traffic light behavior (in our example, the cycle from green to yellow to red; if we see a different order when inspecting the model, we will know the model is wrong). We could also build a more complex version like the one in Figure 1.5(b), which uses a timed version of automaton, in which we include the delays for each of the lights. Using this model, we can also think about correctness of the timing properties (and, for instance, detect a wrong duration for a given cycle).

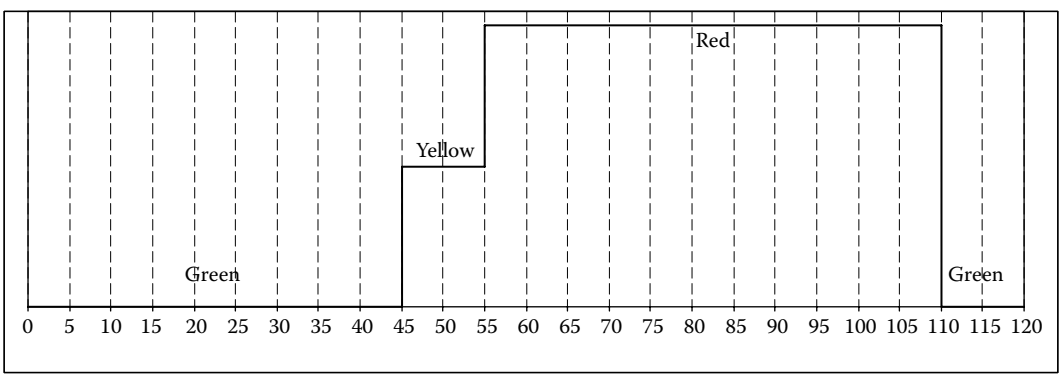


FIGURE 1.4 Observed behavior of a traffic light.

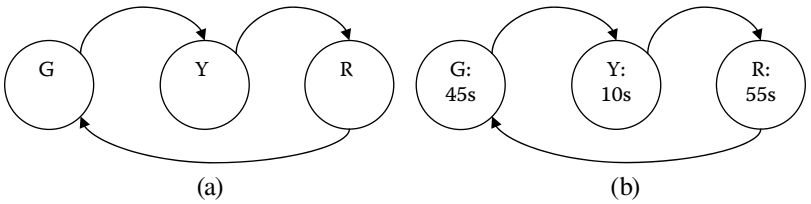


FIGURE 1.5 Simple untimed and timed automata describing the behavior of the traffic light in Figure 1.4.

```
time = 0; State = Green;
Repeat Forever {
      if (State == Green  AND (time mod 110) == 45)   State = Yellow;
      if (State == Yellow AND (time mod 110) == 55)   State = Red;
      if (State == Red    AND (time mod 110) ==110)   State = Green,
      time = time + 5;
}
```

FIGURE 1.6 An implementation of the traffic light model based on the timed automaton of Figure 1.5.

If we now use this model to create a simulator (for instance, an implementation of the timed automata in Figure 1.5(b) using a C-like programming language), we obtain a program in the style of the one introduced in Figure 1.6.

As we can see, we start at time 0 with a green light. The simulator evolves by checking the current state and time and acting according to their values. For instance, after being 45 s in green, the state changes to yellow; after 10 more units, it changes to red and then finally goes back to green. Time is incremented on each cycle using time steps of 5 s (which is the greatest common divisor—g.c.d.—of the three light periods—that is, the minimum value providing enough precision according to the data collected in Figure 1.4).

Exercise 1.2

Build a simulator based on the one in Figure 1.6 and the model in Figure 1.5(b) using a high-level programming language (like Java or C++).

Exercise 1.3

Define a composite automaton describing the behavior of two traffic lights interacting, based on the model in Figure 1.5(a) (in such a way that when one of the automata has a green light, the other is red and vice versa).

Exercise 1.4

Define the behavior of the model introduced in Exercise 1.3 using ordinary differential equations.

Exercise 1.5

Build a simulator to execute the models defined in Exercise 1.3 using a high-level programming language (like Java or C++).

Automata models were defined using discrete values to represent time and the set of states, which works well in many applications but not in others. In order to discuss some of the difficulties, let us consider now a case where the traffic light model introduced in Example 1.2 uses external sensory

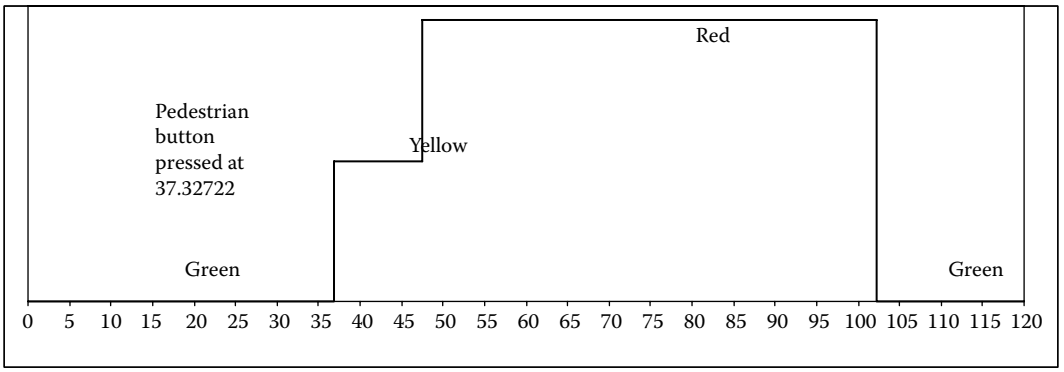


FIGURE 1.7 Observed behavior of a traffic light with a pedestrian button.

information to determine the length of the traffic cycle. We also want to model a button for pedestrians crossing. Figure 1.7 shows an example of the experimental data for such a traffic light.

Here, during the green cycle (which is 45 s long), a pedestrian arrives and presses the pedestrian crossing button, causing a change in the cycle. Such a simple action introduces several problems:

- How do we model the external sensory information in the automaton of Figure 1.5? We want to have cycles that adjust dynamically, so the resulting automaton is much more complex, and it has a larger number of states (if Exercise 1.4 was solved, try now to build the extended automaton for this new adaptation). If we now want to integrate this model for a complex crossing, the number of states might grow exponentially.
- If we need to combine this traffic light with others, how is the variable-timing behavior going to affect the combined automaton?
- When we create a simulator for this model, which would be the right time step to be used? Because neither the arrival time of pedestrians nor traffic flow can be predicted, we cannot find the g.c.d. of every possible cycle (which can be potentially very small or very large). If our requirement is to identify traffic flow changes with high precision, we might use, for instance, a timescale of 0.1 s. Nevertheless, in this case, for every simulation (including those with few or no pedestrians and those with steady traffic), our simple simulator will execute 1,200 cycles (instead of the 22 we used in Figure 1.4). If, instead, we choose a time step of 5 s as before (i.e., to improve the execution performance of the simulator), we cannot accurately model the arrival of vehicles (and in Figure 1.7, we would miss representing the pedestrian button pressed at 37.32722).
- Although this problem does not exist in differential equations (where time is continuous), they are not adequate to represent these kinds of problems (as you could experience if you tried to solve the trivial examples introduced in Exercises 1.1 and 1.3). Instead, automata are easier to understand and better represent these kinds of phenomena.

As we can see, we need different methods to model these human-made applications, in which entities change due to the occurrence of particular *events* and the model's evolution depends on the interactions of such events and their arrival times. For instance, in a computer network, the arrival of a packet is an event of interest, and its arrival time or duration can be of any variable length, depending on the current state of the system and the kind of packet. These kinds of entities, which can be represented with discrete variables and continuous time, are called *discrete-event dynamic systems* (*DEDS*). In DEDS, the states are described by *piecewise constant* trajectories (such as the ones in Figure 1.7). DEDS are naturally concurrent and highly nonlinear; because no transformation

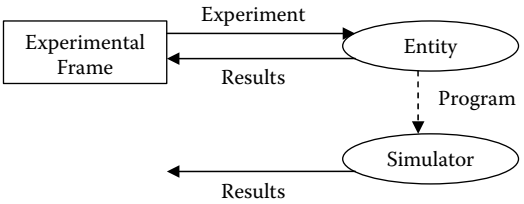


FIGURE 1.8 Building a simulator using a simulation language.

method is available, it is also difficult to find general analytical solutions [8]. The different methods for modeling and simulation (M&S) of DEDS are called *discrete event modeling and simulation*.

Discrete-event M&S assumes that, although time is continuous, only a finite number of events can occur in a given period. Therefore, a discrete-event simulator can be very efficient because we only need to represent the state changes upon occurrence of events. For instance, in order to simulate the traffic light model presented in Figure 1.7, we would only need to execute four cycles (one per color change and one for the button pressed, compared to the 22 or 1,200 previously discussed), while obtaining the maximum precision available to model this traffic light adequately.

Many of the techniques for modeling DEDS only focus on the arrival order of the events, ignoring their timing. They are called *logical DEDS models* and they are used to solve problems as the untimed automata presented in Figure 1.5. Conversely, if timing is a factor, we need *timed DEDS models*.

In order to convert the simulator presented in Figure 1.6 into a discrete-event one, we first need a discrete set of state variables and a clock indicating the current simulation time. Then we use a scheduler to keep a chronological list of events (in general, stored as messages) that represent the state changes. The time in which this will happen, $ti \in \boldsymbol{R}$, is a continuous variable. At every time, the simulator will pick the first event in the list, process it, change the value of the state variables, and cycle to the next event. As we can see, the complexity of the simulation algorithm is higher, and the management of the event list is very important.

In the 1960s, techniques for discrete-event simulation (based on the ideas just discussed) became very popular. In many cases, this resulted in the definition of advanced simulation languages like SLAM, Arena, Simula, or SimScript [9–11]. Although simulation languages can address complex problems, their use lacks the formality of previously existing modeling methods. Using a simulation language helps with problem solving and experimentation, but in most cases their foundation is not rigorous, making the resulting simulation software difficult to test, maintain, and verify. Likewise, changes in the language can produce serious effects in existing models because their semantics are usually not formally defined. Simulation languages do not provide a method abstract enough to think about the problems to solve or to prove properties of the entities under study, which could improve the final quality of the analysis while reducing end costs. Another problem faced by simulation language–based solutions is more subtle and complex to address. The source of many of these problems can be experienced by solving Exercise 1.2 or 1.5, and it is summarized in Figure 1.8: as most simulation languages were not derived from a formal modeling framework, the modeling phase, actually skipped. We start by collecting data from experiments, and we build a piece of software (the simulator), trying to mimic the problem under study (skipping the intermediate modeling phase). Although this method is still useful in many cases, the result is a single-use program approach, which can have several problems (we will use Example 1.2 to discuss some of these problems):

- What happens if we need to reuse the simulation in a different context (e.g., we want to reuse the traffic light controller simulation for a railway controller simulation)?
- How can we reuse the experiments done to test the original model on a different one?

- How do we deal with changes? If we decide to add a blinking green light for left turns (or a blinking red light for a failing controller), we need to modify the software application completely.
- Where is the abstract model we can use to organize our ideas? How do we organize the creation of a new version of the simulator in which we need to study the intersection of six streets with two-way traffic?
- How do we validate the results? What do we do if we find errors in the simulation?

It is difficult to address these issues using the same simulation languages because the model is usually mixed with the experiment and the simulation software. For instance, in Example 1.2, the traffic light controller would be mixed with the generation of pedestrians arriving at the corner and the simulation routines that make time advance and decide what to do next. Even with a simple example like this one, reuse is complex—how can we reuse the control algorithm for the traffic light? Any changes would result in going through the code for the simulation and the experiment because there is no model to use in the verification process. Building a program from the experimental data, as in Figure 1.8, would result in having the original software discarded and a new simulator built from scratch for the next simulation project.

Instead, using a model to organize our ideas can help to create a better product at a reduced cost. Although nonformal models (sometimes called *conceptual models*) can be of help in this task, a formal model like the automata in Figure 1.5 provides much better facilities for verification, reuse, modification and testing. It also provides the basis to define a process that would allow repetition of successful previous experiences, enable reuse, and introduce well-known software engineering practices in the creation of the simulation software.

With these goals in mind, different groups investigated the creation of *formal* discrete-event modeling techniques. Such *formalisms* provide a communication convention written in a mathematical language, which we can use as a guide to provide a nonambiguous specification of the system's semantics [12,13]. Formalisms can be used to represent the entities under study formally, creating an abstract model and providing means for manipulating such abstraction, while being able to translate them into executable models. A *formal model* (i.e., one built using a formalism) provides a sound mechanism to specify the entity under study that can be formally verified. This improves error detection and reduces the development and verification time of the simulation software, enhancing the security and reducing the development costs of the simulation. Going back to Figure 1.3, if we can prove properties of the model before creating its computational version, we can improve the final product at a lower final cost. Simultaneously, a formal mechanism can disambiguate communication, enhancing teamwork by providing a sound notation for the model being constructed.

The rest of this book is focused on one of these discrete-event M&S formalisms, called DEVS (discrete event system specification), a sound M&S theory that has evolved in the last 30 years [12,14–20]. In order to understand the basics of this methodology, we will first discuss some general ideas and we will provide a few general definitions.

1.3 CLASSIFICATIONS OF MODELING TECHNIQUES

Defining taxonomies and classifying methods can help us to have a better understanding of a field under study. Fishwick [13] proposed the following classification, which is based on the types of techniques used (we will use Example 1.1 to show the differences of each of the techniques):

1. *Conceptual modeling:* this technique is based on the creation of an informal *conceptual model* that communicates the basic nature of the process. It provides a vocabulary for the application (which can be ambiguous) and a general description of the entity to be modeled. Example 1.3 shows a simple version of a conceptual model for Example 1.1.

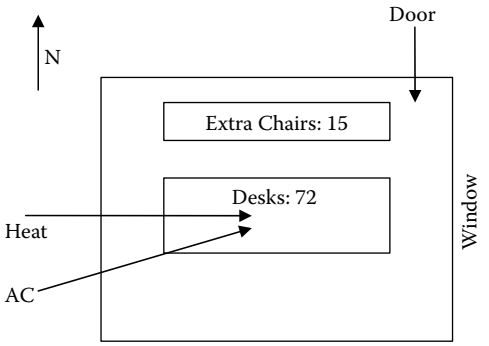


FIGURE 1.9 Classroom organization.

EXAMPLE 1.3: CONCEPTUAL MODEL FOR EXAMPLE 1.1

We are interested in studying temperatures in a classroom throughout the day. The orientation of the building is north (N)/south (S), and there are large glass windows to the east (E). The classroom is 15 × 20 × 3 m. There is a total of 72 movable desks in the room, which are set up (by default) in eight rows of nine desks each. There is also one desk in the front and 15 extra chairs available (without a desk). The classroom has one projector, a whiteboard on the front (which is heading to the S), and a blackboard on the west (W) wall.

There are three teaching seasons (no lectures in summer). Once every hour, an alarm will sound to indicate the end of the lectures. Once every minute, the room's temperature is sensed, and the heating/air conditioning (AC) is activated as needed. In fall and winter, if the temperature goes below 22°C, the heating is turned on, and if gets above 25°C, it is turned off. In spring, if the temperature goes above 24°C, the AC is turned on, and it is turned off when the temperature gets to 23°C (Figure 1.9).

The influence of the temperature outside and in the rest of the building will be neglected. There is a heating/AC source in the middle of the room (on the ceiling) and a door at the back. After extensive experimentation, we know that every time a person arrives in the classroom, the temperature increases 0.05°C after 5 min. Likewise, 5 min after someone leaves, the temperature decreases 0.05°C.

Parameters are room size (15 × 20 × 3 m), desks (72), chairs (87), and board positions (W, S).

State variables are number of students, activity (lecture/no lecture), desk positions (default, circle, semicircle), heating temperature, air conditioning temperature, and season (summer, fall, winter).

Exercise 1.6

Find ambiguities and missing information in this conceptual model. Which of these problems are derived from the expression in natural language? How would you solve these problems?

2. *Declarative modeling:* these techniques are focused on the evolution of the model represented as states (which describe the behavior of the entities under study) and the transitions between them. According to the particular focus of the technique, we can have *state-based* or *event-based* declarative models. *State-based* declarative models, for instance, can be represented as a graph where the vertices represent the entities and the arcs represent the state changes (transitions).

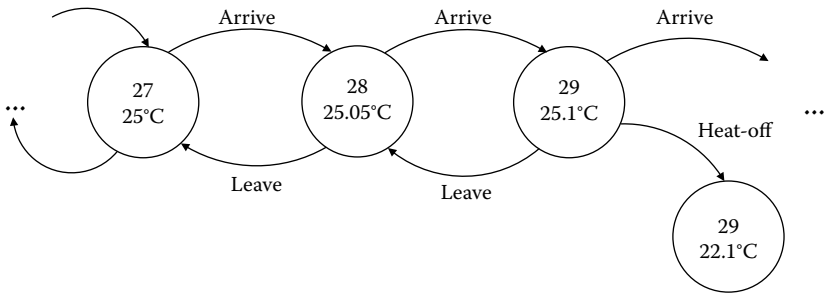


FIGURE 1.10 State-based declarative model of the classroom example.

EXAMPLE 1.4

Figure 1.10 shows a state-based declarative model representing a portion of the system's specification, in which students arrive at or leave the classroom. At a certain point, we have 27 students in the lecture room and a temperature of 25°C. If a new student arrives, we will have a total of 28 students and 25.05°C (this is an untimed model, so timing information is not included). If a student leaves, we return to the previous state; if another student arrives, we have 29 students and temperature will go up (25.1°C). We also show a state representing that the heating has been turned off, which will reduce temperature in the classroom accordingly.

In *event-based* declarative models, we use a graph-based notation with nodes representing events (i.e., the state changes that occur when there is a particular kind of event detected) and links representing the relations between those events. The state changes are associated with each event, and the links can have logical relations associated (defining the relations between events).

EXAMPLE 1.5

Figure 1.11 shows an event-based declarative model for the classroom that represents the same phenomena analyzed in Figure 1.10. If we schedule the arrival of a new student, we will have one more student in the room and temperature will increase. The model can receive more students while there is room available ($q \leq 85$). At any moment, we can schedule a student to leave (in this case, we will decrease temperature and the number of

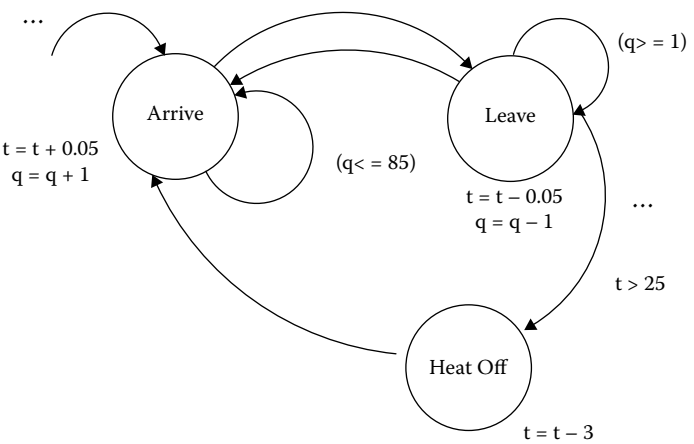


FIGURE 1.11 Event-based declarative model of the classroom example.

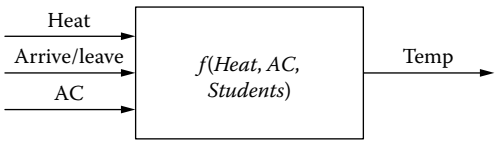


FIGURE 1.12 Functional model of the classroom example.

students in the room). We need at least one student to schedule a student departure ($q \geq 1$). If we schedule the *heat off* event, temperature goes down 3°C.

3. *Functional modeling:* in this case, the model is defined as a "black box" and the input is a signal defined over time. The output depends on an internal function, and the model can use discrete or continuous functions.

EXAMPLE 1.6

Figure 1.12 shows a black box representing the same portion of the system's specification presented in the previous two examples. The function *f* will receive information about students arriving or leaving the classroom. According to the current number of students and level of AC/heat, it will generate the room's temperature (*temp*).

4. *Spatial modeling:* in these techniques, space notions are included (i.e., the relationship between time and space is included in the model).

EXAMPLE 1.7

Figure 1.13 shows a spatial model for the classroom system. At time *t*, the students were distributed in the classroom as in the left part of the figure; at time $t + 1$, we see that a new student has arrived and is now seated in the back row. The number of students will influence the temperature in the classroom.

A different categorization classifies the different modeling techniques according how we represent the state variables and time in the model. Figure 1.14 shows such classification, organizing the various techniques mentioned earlier in this chapter according to their time base and state variables' representations.

As we can see, two criteria are used for the classification:

(a) According to the *time base,* there are **continuous time** paradigms, where time evolves continuously (time is represented a real number), and **discrete time** techniques, where time evolves by advancing in discrete portions (time is an integer number).
(b) According to the *values of the state variables,* there are **continuous** models, where the variables take their values from a continuous set represented as a real number, and **discrete**

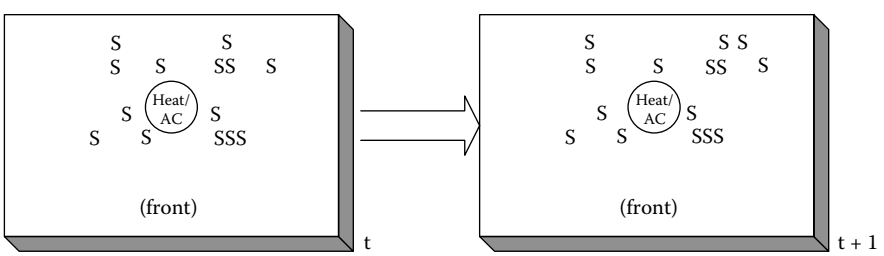


FIGURE 1.13 Spatial model of the classroom example.

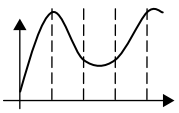
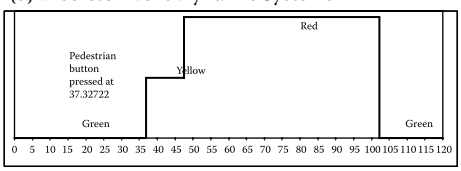
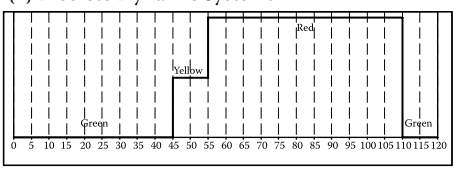
Vars./Time	Continuous	Discrete
Continuous	{1} Continuous Variable Dynamic Systems 	{2} Discrete-Time Dynamic Systems 
Discrete	{3} Discrete-Event Dynamic Systems 	{4} Discrete Dynamic Systems 

FIGURE 1.14 Classification according the representation of time bases/state variables.

models, where the variables are discrete and can be represented as a finite set of integer numbers.

Figure 1.15 classifies some of the existing modeling techniques according to these criteria.

According to Zeigler, Praehofer, and Kim [17], entities with a behavior like the one depicted in part {1} of Figure 1.14 (which are usually called *continuous variable dynamic systems*) can be defined as *differential equation systems specifications* (*DESS*). If we consider the room's temperature in Example 1.1, we can represent it as a DESS (i.e., temperature in the room can be described as in Figure 1.14{1}). We could use continuous modeling techniques like Bond Graphs, Modelica, and other techniques in Figure 1.15{1} to represent this kind of entity.

Behavior like the one in Figure 1.14{2} (usually called *discrete-time dynamic systems*) can be defined using a *discrete time system specification* (*DTSS*). For instance, the temperature sensor used in Example 1.1 (which measures temperatures at fixed periods) can be represented by difference equations (and other techniques in Figure 1.15{2}).

Figure 1.14{3} shows the behavior of *discrete-event dynamic systems*, which can be described using any of the *discrete event systems specification* (*DEVS*) techniques in Figure 1.15{3} (e.g., the arrival and departure of students in the classroom). Example 1.1 can be modeled with timed FSM, event graphs, etc. as discussed earlier.

Finally, Figure 1.14{4} represents the so-called *discrete dynamic systems*, which can be represented as a specialization of DEVS models in which the events occur at a fixed time. In our classroom, the hourly alarm can be modeled with FSMs, Petri nets (PNs), and other techniques included in Figure 1.15{3}.

Vars./Time	Continuous	Discrete
Continuous	**{1} DESS** Partial Differential Equations Ordinary Differential Equations Bond Graphs Modelica Electrical Circuit Diagrams	**{2} DTSS** Difference Equations Finite Element Method Finite Differences Numerical Methods (Runge-Kutta, Euler, DASSL, and others)
Discrete	**{3} DEVS** DEVS Formalism Timed Petri Nets Timed Finite State Machines Event Graphs	**{4} Automata** Finite State Machines Finite State Automata Petri Nets Boolean Logic Markov Chains

FIGURE 1.15 Classification of modeling techniques according the representation of time bases/state variables.

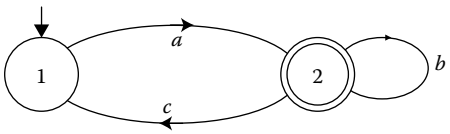


FIGURE 1.16 A simple automaton: $a\{b^*\{cab^*\}^*\}^*$.

1.4 DISCRETE-EVENT MODELING AND SIMULATION METHODOLOGIES

Modeling techniques for DEDS are relatively recent (especially if we compare them with those used for modeling CVDS). In this section, we present a noncomprehensive list of some of the formal modeling techniques created for modeling DEDS (readers interested should consult the references included here).

An *automaton* is defined as a graph representing system states and the transitions between them. The automaton receives a string of symbols as input, and it recognizes/rejects the inputs by advancing through the transitions. The input is read one symbol at a time; depending on the ending state, the automaton will accept or reject the input [6,7].

The automation in Figure 1.16 has an initial state 1 (represented by the arrow) and an ending state 2 (represented by the double circle). If the input *a* is received, the automaton transitions from state 1 to state 2. Once in state 2, it can remain there while receiving the input *b,* it can return to state 1 if it receives input *c,* or it can terminate. Thus, this automaton recognizes the strings {*a, ab, abb, abbb, abbbbb, …, aca, acab, acabb, acabbb, …, abca, abbca, abbbca…*}. This can be represented as $a\{b^*\{cab^*\}^*\}^*$, where * means 0 or more repetitions and {} are used to group strings.

Automata can be deterministic (like the automaton in Figure 1.16), but there are several extensions to the original method, including the nondeterministic automata (in which the transitions can be probabilistic), *input/output* (in which the automaton can trigger a transition when an input is received and can generate outputs, automata noted on the state), etc.

Timed automata, in particular, use *clocks* to describe the model's timing behavior [21]. The automaton is defined as a graph of states associated with clocks that determine the passage of time since the occurrence of an event. Every link is associated with a timing constraint that will define when the transition can be triggered. Whenever a transition executes, the associated clocks are reset. Timing constraints can also be associated with the model states, defining the duration of each of the states.

Figure 1.17 shows the definition of the traffic light model using a timed automaton. The initial state of the model is green, and it will be kept for 45 s. When this time arrives, it changes to yellow, producing an output (the {*yellow*} light). If, before 43 s have passed, a pedestrian presses the crossing button, we switch to yellow (this change will take 2 s). The transitions to red and to green also follow our previous definitions.

Finite state machines (FSMs) can be represented as a graph in which the system's behavior is defined as a finite set of nodes (the model's states) and links between them (transitions between states). A given state reflects the evolution of the model, and transitions are associated with a given logical condition to enable the execution of the transition. When entering a state, an entry action can be executed (and an exit action can be executed when leaving it). Likewise, an input action can be triggered based on the current state and an input [6,7]. An FSM is formally defined as

$$\text{FSM} = \langle \text{S, X, Y, f, g} \rangle \tag{1.2}$$

where

X = finite input set
Y = finite output set
S = finite state set

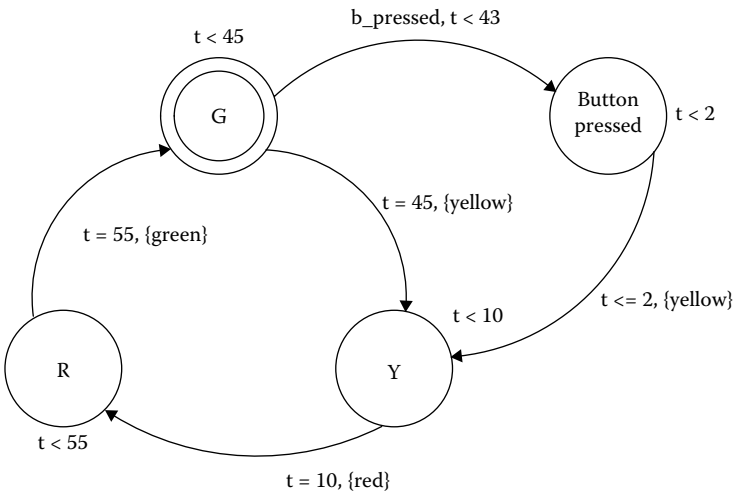


FIGURE 1.17 A timed automaton for traffic light.

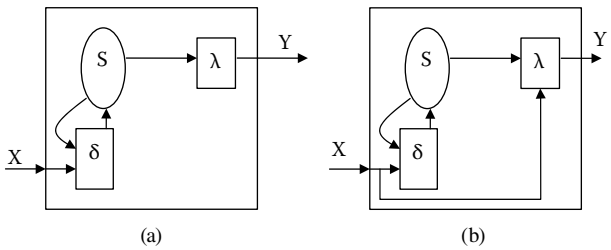


FIGURE 1.18 (a) Moore machine; (b) Mealy machine.

δ = next state function
$\delta = X * S \rightarrow S$
λ = output function
λ: $S \rightarrow Y$: Moore machine
λ: $X * S \rightarrow Y$: Mealy machine

Every FSM is supposed to have an initial state, a next-state function that defines how to obtain the next state in the system, and an output function that uses current state and inputs to generate outputs. According to the relationship between the input sets and output functions, two kinds of FSMs can be defined: Moore machines, in which outputs are independent from the inputs, and Mealy machines, in which, besides the current state variables, the current inputs are analyzed to decide the current output value. These two types are depicted in Figure 1.18.

Timed versions of FSMs associate time with the places or the transitions. Further details about untimed and timed FSMs can be found in Cassandras [6] and Papadimitriou [22].

A *Markov chain* [23] is a discrete-time stochastic model described using a graph (Figure 1.19). Models' states are defined as nodes in the graph, and transitions between states are represented by links. One important property of Markov chains is that they are memoryless; thus, no state has a cause–effect relationship with the previous state. Therefore, knowledge of previous states is irrelevant for predicting the probability of the future states.

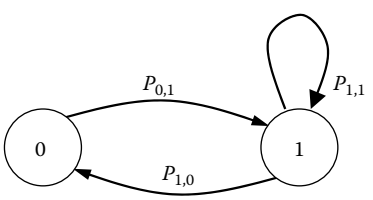


FIGURE 1.19 A simple Markov chain.

In Figure 1.19, we have a binary Markov chain, in which the probability to go from 0 to 1 is $P_{0,1}$, the probability to go back from 1 to 0 is $P_{1,0}$, and the probability of remaining in state 1 is $P_{1,1}$. Markov chains can use discrete time or continuous time, and states can take a discrete value (i.e., values from a finite or a countable infinite set). On a discrete-time chain, we compute the probability to change from one state to the next, and this will happen in discrete time. In continuous-time chains, transitions can occur at any time; thus the transition probabilities must be defined for every instant.

A *Generalized Semi-Markovian Process* (*GSMP*) is a stochastic process (i.e., a collection of random variables over a probability space indexed by time). A GSMP is based on the notion of a state that makes a transition when an event associated with the current state occurs, and the state space is generated by a stochastic timed automaton [24]. Several possible events can compete to trigger the next transition, and each of these events has its own probabilistic distribution for determining the next state.

The GSMP is defined as a set of *locations,* a finite set of *events,* a function that assigns a set of *active* events to each of the locations, a *firing time distribution* associated with each event, and a *transition* function, which takes an active event for a given source locations and returns a set of events and a probability measure. At each transition, new events may be scheduled.

For each event, we set a clock indicating the time when the event is scheduled. It is assumed that the set of scheduled events is uniquely determined by the current state and that there is a unique triggering event for each state transition. A *run* of a GSMP is a sequence of alternating timed and discrete transitions. The run starts at the initial configuration $s_0 = (q_0, e_0)$, where q_0 is the initial location for the GSMP and e_0 is the initial valuation of the events scheduled according to the corresponding firing time distributions (Figure 1.20).

$$t^*(s_0) \quad \mu_0 \quad t^*(s_1) \quad \mu_1 \quad t^*(s_2) \quad \mu_2$$
$$s_0 \longrightarrow s'_0 \longrightarrow s_1 \longrightarrow s'_1 \longrightarrow s_2 \longrightarrow s'_2 \longrightarrow \cdots$$

FIGURE 1.20 Execution of a GSMP.

Petri nets [25] define the structure of concurrent systems using a bipartite graph. One type of the graph's nodes, the *places*, represents the system states, and the second kind, the *transitions*, represents the net evolution. The PN example in Figure 1.21 represents a producer/consumer system (the producer is on the left and the consumer on the right of the figure). Here, *L1–L5* are the PN places and *t1–t4* the PN transitions. *L1* represents the producer ready, *L2* a producer that has generated an element to be consumed, *L3* a buffer between the producer and the consumer, *L4* a consumer that is ready, and *L5* a consumer active.

Because a PN is a digraph (i.e., only connections between places to transitions or vice versa are legal), the PN defines a static view of the system's structure (as we can see in Figure 1.21). If we are interested in studying the dynamics of the system, the PN needs to execute. This is done by placing a special mark (called a *token*) on the places (in this case, we have one token on each of places *L1*

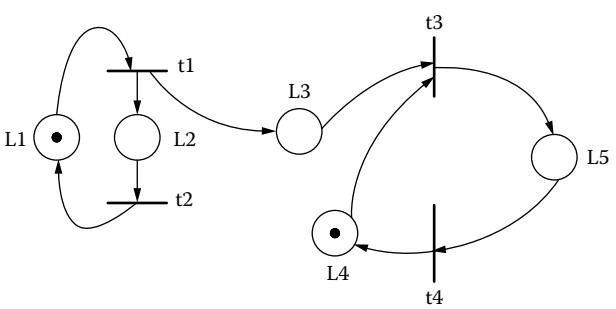


FIGURE 1.21 A producer/consumer Petri net.

and *L4,* representing that the producer and consumer are ready to start). The number of tokens per place is unlimited, and the number of tokens in the whole net defines the net's *marking.* The PN evolves by executing the transitions, which is done by taking one token per *input* place (i.e., those with a link from a place to the executing transition—for instance, *L1* → *t1*) and placing one token on every *output* place (i.e., those with a link from the executing transition, i.e., *t1* → *L2, L3*). Each execution of a transition is called *firing* the transition. We can only fire a transition that is *enabled* (i.e., one having at least one token on each input place). The execution of a transition is atomic, and if more than one transition is enabled, they execute in nondeterministic fashion.

Figure 1.22 shows the execution of the PN in Figure 1.21, using a *reachability tree*, which contains one node per marking and one link per firing. For instance, the initial marking in Figure 1.21 can be represented as the configuration: (1,0,0,1,0). With this marking, only *t1* is enabled (it is the only transition with a sufficient number of tokens in the input places). If we fire *t1*, we obtain the marking (0,1,1,1,0), which enables *t2* and *t3*. Any of them can be fired (and which one is fired is decided in nondeterministic fashion). For instance, if we fire *t2*, we obtain (1,0,1,1,0), which is a similar marking to the root of the tree but with an extra token in *L3*. At this point, if we repeat the firing sequence *t1* → *t2*, we will obtain the same marking, with one extra token in *L3*. Thus, instead of expanding this branch of the tree indefinitely, we put an asterisk in the corresponding position for *L3* (meaning that the number of tokens in this place can grow indefinitely) and stop expanding this branch of the tree.

If, instead, we fire *t3*, we obtain (0,1,0,0,1). At this point, we can fire *t2* or *t4,* etc. When we cannot fire further transitions or we obtain a marking repeated in the tree, we stop expanding the tree. For instance, after firing *t1* we obtain (0,1,0,0,1), which is repeated in the tree; thus, we stop expanding this branch.

By studying the tree, we can study concurrency problems, deadlocks (i.e., a PN that does not evolve and have resources in a cycle), unbounded buffering problems, starvation (transitions that can never execute), etc.

Exercise 1.7

Study the reachability tree in Figure 1.22 and discuss the meaning of each of the states.

Exercise 1.8

Add two extra tokens in *L1* and rebuild the reachability tree.

Exercise 1.9

Using the reachability tree in Figure 1.22, carefully check whether the occurrence of concurrent states is correct. Find cases of deadlock, starvation, or any other problems (if there are any). Repeat the exercise for the tree obtained in Exercise 1.8.

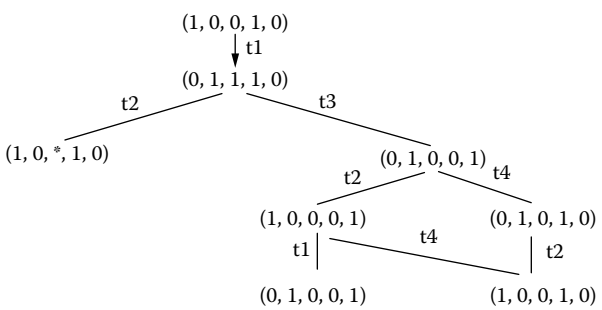


FIGURE 1.22 Reachability tree for Figure 1.21.

Exercise 1.10

Delete the token in *L4*, move it to *L3*, and repeat Exercises 1.8 and 1.9.

Queuing networks are based on a customer–server paradigm, in which customers make service requests to the servers and these requests are queued at the server until they can be serviced. The arrival time for customers and the service time at a server are described as stochastic models. By defining the number of servers and the buffering capacity on each of them, we can determine performance metrics (including the number of customers in line, throughput—number of customers serviced per time unit, turnaround times, etc.). Different policies can be used (priorities; preemption; first in, first out [FIFO]; etc.). Traditional queuing networks model only a single layer of customer–server relationships. Layered queuing networks (LQNs) allow for an arbitrary number of client–server levels [26,27]. LQNs can model intermediate software servers, and they can be used to detect software deadlocks and software as well as hardware performance bottlenecks. The layered aspect of LQNs makes them very suitable for evaluating the performance of distributed systems [28].

The formal language of *calculus of communicating systems* (*CCS*) provides primitives for concurrency and parallelism, based on synchronous communications between exactly two components. The language expressions are interpreted as a labeled transition system, and bisimulation can be used to prove equivalence of models [29].

Temporal logic is a system of rules and symbols used for representing propositions that can include the timing properties of the system [30]. It consists of a logic set of propositions that view time as a sequence of states and that can be true or false according to their state and their time of occurrence. Temporal logic has been used to verify formally timed automata. The idea is to check predictability of certain conditions according to the time that they occur, conditions that might eventually arise, or others that are guaranteed not to occur.

Communicating sequential processes (CSP) is a formal language based on process algebra that has been widely used to model concurrent systems [31]. Models are described using independent processes that interact with each other through message-passing representing the occurrence of events. The processes' primitive elements include:

- Prefixing ($\rightarrow$) connects an event with a process. For instance, if we write $x \rightarrow P$, that means that the process P will wait for the event x, after which P will execute.
- Inputs (?) represent the reception of inputs for the process. For instance, when we write $?x{:}X \rightarrow P(x)$, this represents a process P that, upon reception of x (which belongs to the set of inputs X), executes $P(x)$.
- Choice: the *external* choice permits us to select one of several actions. For instance, $P = x \rightarrow P1 \,\square\, y \rightarrow P2$ can execute either of the two processes, depending on the event received (x will trigger $P1$ and y will trigger $P2$; if both occur, the choice is nondeterministic). The *internal* choice picks one of them in a nondeterministic fashion. For instance, $P = x \rightarrow P1 * y \rightarrow P2$ can choose either of the two processes, independently from their inputs. Other operators include a choice with conditionals, Boolean operators, etc.
- Recursion is defined by having the identifier of the process at both ends of the equation—for instance, $Clock = tick \rightarrow tack \rightarrow Clock$, which will repeat the events {*tick, tack*} forever.
- Parallel: if we have $P1\ X\|Y\ P2$, $P1$ and $P2$ execute in parallel and they synchronize using the intersection of X and Y.

State transition diagrams usually suffer from a "state explosion," in which the number of states and transitions becomes unmanageable. *State charts* [32] introduce hierarchy into the model, allowing definition of varied behavior at different levels, as seen in Figure 1.23. A state is represented as a rectangle with round corners, which can be subdivided into a list of internal transitions (which are performed while the element is in the state). The start state is represented as a circle and the final

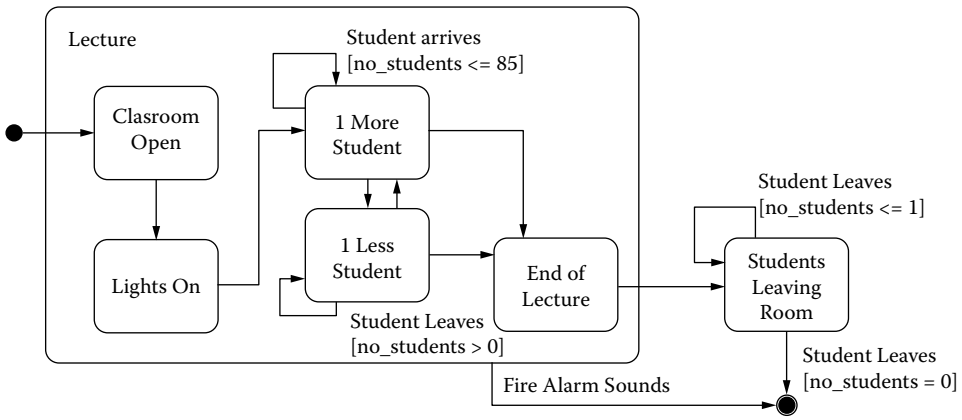


FIGURE 1.23 State chart example.

state as a double circle. The transitions between elements are represented with arrows. Numerous other constructors are available (concurrent separators, complex transitions, event messages, etc.).

In Figure 1.23, we have two states: a lecture and an end of lecture (where students leave the room). As we can see, we can hierarchically subdivide the lecture state into substates that will represent the start of the lecture, arrival and departure of students, and the end of the lecture.

State charts are also used in the UML dynamic model to show the behavior of a single class [33]. State charts introduce the concept of modeling of concurrent machines and superstates (i.e., a state that contains other states), thus making it possible to have two or more concurrent states (as opposed to classic state machines, which can only be in one state at a time). States and superstates can be nested, each with its own initial and final states.

Specification and Description Language (*SDL*) was created to specify in a nonambiguous way the behavior of real-time applications. It was originally focused on communication systems, by providing a graphical and textual representation with equivalent semantics. A system is defined as a set of extended FSMs that can be interconnected [34].

Event graphs are oriented graphs that represent the organization of the events of a discrete event system [35]. As seen earlier in Figure 1.11, events constitute nodes of the graph; that is, the vertices represent the state transition functions, and the links between nodes capture the scheduling of such events. Each link starts at the node performing the scheduling operation (which represents an event), and it ends at the node representing the event to be scheduled. Each scheduling relationship has an associated delay and condition (a Boolean function of the state), and an event is scheduled only when the condition is true.

Systems-theoretical approaches derive from systems theory [36]. Systems theory represents every entity under study using the concept of *system*, which is seen as a collection of objects and their interactions. In systems theory, the system's global behavior is seen as a composition of the individual behavior of the components, and we can find emergent behavior that is not explicitly defined in the parts of the system. Systems theory is based on the idea that every phenomenon can be viewed as a mathematical relationship among a set of entities in the system. The theory is generic and tries to find common behavior and properties in different fields of study (for instance, hydraulics, economy, biology, or social sciences), thus providing a unified view of science and engineering.

Systems theory distinguishes the structure (internal constitution) of a system and its behavior (external manifestation). Each component is seen as having inputs, outputs, and internal structure that dictates how inputs and states are transformed into outputs. If the model only allows observing the external manifestation, we call it a **black box** (or **behavioral**) model; when we analyze it, we only consider the input/output trajectories observed. These models contain a representation of observable and unobservable variables, and the system's behavior is described as a set of input/

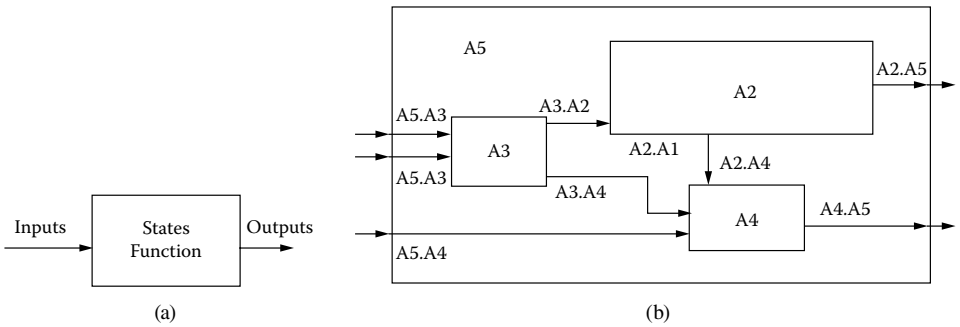


FIGURE 1.24 (a) Behavioral model; (b) structural model.

output data that we can observe about the system of interest. When the model represents the internal constitution of the system, we call it a **white box** (or **structural**) model. Structural models introduce concepts of *decomposition* and *coupling,* and they refer to the component elements in the model (as opposed to the behavior they generate). Decomposition focuses on how to divide a system into components, and coupling focuses on how to combine the components to reconstruct it. Using these concepts, we must explicitly define the components of the real system, using decomposition and coupling concepts (Figure 1.24).

In the case of Example 1.1, we can build a behavioral model of the temperature of the whole room by defining a transition *function* that will be activated every time a student arrives/leaves, updating the temperature and number of students in the room (*states*). On the other hand, a structural model would consider the different heat/cold sources, the student's seating area, and the connection to the rest of the building. In this case, we are not interested in the behavior but, rather, in the decomposition of the model into subcomponents and their interrelation.

Systems theory also focuses on different views of the same system. For a given system, we can have many models, which will be valid with respect to the objective for which it was created (that is, it is going to be able to answer to queries according to the objective and the experimental frame under which it was created). The bottom line is that a *complete* model never exists because there is an indefinite number of models that can be created for the same system. Simultaneously, every model can have many different implementations (simulations), and a variety of experiments can be carried out. In Example 1.1, we can use the source system and create an experimental frame to study the temperature (as described in the different examples we have discussed). The same source system could also be used for studying weight of the student population, its gender, distribution of students in classrooms, scheduling of lectures, construction of the building based on occupancy, evacuation under emergency, etc.

Discrete-Event Systems Specifications (DESS) formalism [12,15–17,18] is a mathematical modeling technique derived from systems theorythat allows the construction of hierarchical and modular models, providing a well-defined coupling of components. Given its hierarchical nature, DEVS allows the coupling of existing models modularly, allowing us to build complex systems from simple ones. DEVS theory provides a rigorous methodology for representing models, and it presents an abstract way of thinking about the world independently of the simulation mechanisms.

The formalism is based on general system theory, and it allows us to represent all the systems whose input/output behavior can be described by sequences of events. The generality of DEVS is derived from the fact that it permits the modeling of systems with a set of infinite possible states where the new state after an event arrival may depend on the (continuous) time elapsed in the previous state.

Exercise 1.11

Classify each of the previous techniques according to Fishwick's taxonomy.

1.5 SOME DEFINITIONS

Having discussed the basic ideas we will cover in the rest of the book, we are now ready to introduce some definitions.

According to systems theory, a *system* is a natural or artificial entity, real or abstract, that is a part of a given reality constrained by an environment. It can be seen as an ordered set of related objects that evolve through different activities, interacting to achieve a goal. It is also called the *real system* (or the *system of interest*), and it is seen as a source of observational data, which is viewed through an *experimental frame* (EF) of interest to the modeler [17]. The system's observational data are used to define the model's *behavior*, which is defined as a specific form of data observable in a system over time within an experimental frame.

A *model* is an understandable representation (abstract and consistent) of a given system that we use to understand it better. Models can be built in a variety of ways, and they have different meanings according to the individual doing the modeling. For architects, a model can be the drawing of a floor map or a maquette; for a biochemist, a model can be a three-dimensional representation of a molecule; etc. In this book, we are not interested in such static models. Instead, we are interested in models that have *dynamic behavior* (i.e., models that exhibit time-varying changes), we want to study how the model organizes itself over time in response to imposed conditions and stimuli. The objective is to predict how the system will react to external inputs or structural changes. The model's behavior is generated using specific rules, equations, or a modeling formalism with the goal of generating behavior that should be indistinguishable from the one of the system within one or more models' EFs. The EF defines the conditions under which a system or a model is observed or experimented with; thus, problem solving is related to the EF within which the model is analyzed [17].

The process of thinking and reasoning about a system in order to abstract the description of the model from reality is called systems *modeling.* We will use the term *paradigm* to refer to the concepts, laws, and mechanisms that are used to define a set of models. It is important to keep a clear separation between the systems of interest and the models that we use to think about them. (A model is an abstract representation of the system rather than the system itself, although it is easy to confuse them because we are used to thinking about models for reasoning about real systems.)

Discrete-event modeling is based on the notion of *event,* which is defined as a change in the state of the model. An event occurs at a given *instant* (called the *event time*) and causes the model to *activate* in order to produce a *state change* (e.g., at least one attribute in the model will change). Finally, a model's *state* is the set of values of all the attributes of the model at a given instant. The model's attributes are usually stored in variables; *state variables* are those that will influence the evolution of the model's behavior [37].

A model is an *abstract* representation of the system of interest because the model is an aggregated elaboration of the information provided by the system (with a format based on rules, equations, and relations between components). In modeling, we define *abstraction* as the basic process used to extract a set of entities and relations from a complex reality [17]. During this abstraction process, information is lost; however, a higher level of abstraction allows us to better define the model's behavior and to prove properties of the system by manipulating the abstract model definition while addressing the problem at the right level of complexity. Figure 1.25 shows an example of a variety of abstraction levels in a real system.

In Figure 1.25, we start with a continuous signal in the plant that is read by sensors. In order to model the behavior at this level of abstraction, we need to use a continuous system methodology (for instance, bond graphs). If we want to feed information from the plant to a digital computer, we sample the input data (and in order to model this sampled data we need to use a discrete time technique). The sampled data information from the plant is stored in an image of the plant, and we use intelligent agents within a supervisory control system to process the inputs and react accordingly (for instance, to activate alarms under emergency). The intelligent agents need to identify

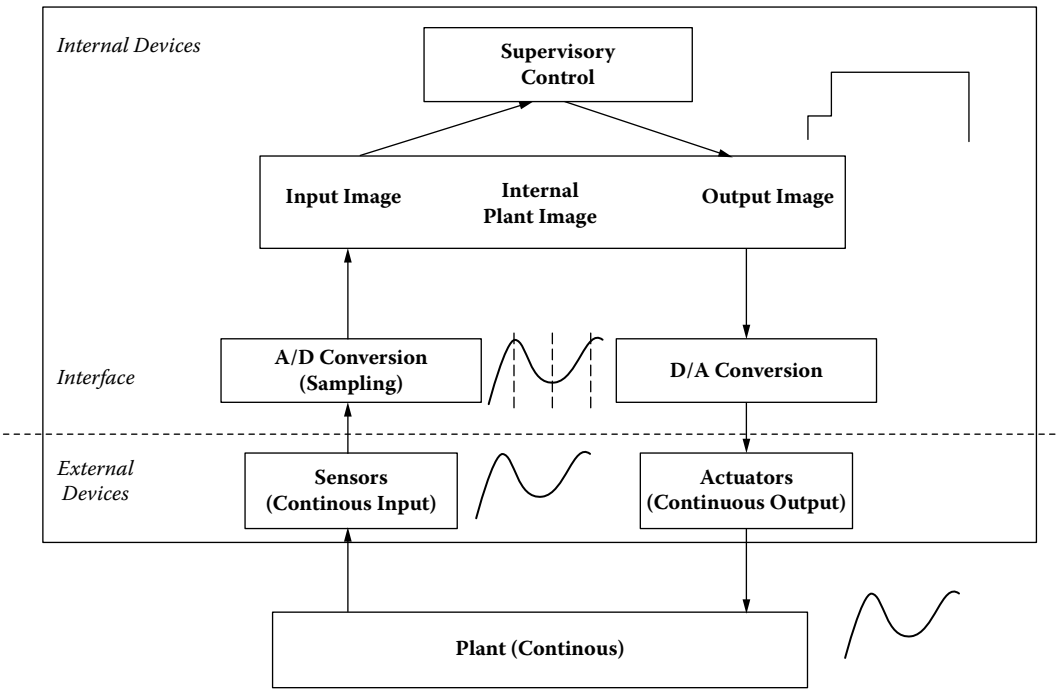


FIGURE 1.25 Different abstraction levels in a plant model.

meaningful events to react to; thus, a discrete-event model, detecting important events to be processed by the supervisory system, would be the best choice. As is clear in this process, we lose information at each step, but we are able to make decisions that are more precise at each level by using the right level of abstraction. Current systems of interest usually include components defined with multiple techniques (also referred to as *multimodeling*). Likewise, each of these models might have different levels of *resolution* according to the needs. (For instance, under emergency, we might downgrade the sampling rate to better utilize the computer resources in dealing with the emergency, thus reducing the quality of the input data while the emergency lasts.) These *multiresolution* models are needed more and more frequently as they provide multiple versions of a model that can be easily adapted and reused.

We can now define *simulation* as the reproduction of the dynamic behavior of a system of interest with the goal of obtaining conclusions that can be applied to the system.

A *simulation study* refers to a set of simulation-based experiments. Typically, the model associated with a simulation study is implemented as a computer program (also called a *simulation model* or a *simulator*). The *simulation experimental frame* contains information about the experimental conditions, parameter values, and behavior generation mechanisms. A simulator may interface with real-world entities (e.g., a moving platform for a driving simulator) or with humans (e.g., a driver in the simulator). We call the first a simulator with *hardware in the loop* and the second a simulator with *human in the loop.*

In order to be able to create a model and a simulator that represent the system with precision, we need to carry out verification and validation (V&V) activities. We use the term *validation* to refer to the relationship of the model, the system of interest, and the EF. A model's validation answers the question of whether it is impossible to distinguish the behavior of the model and the system within the EF. *Verification* is the process of verifying that a simulator of a model correctly generates the desired behavior [17].

EXAMPLE 1.8

Let us use Example 1.1 to explain some of the definitions just presented. The *experimental frame* consists of different classroom configurations for the study (students entering/leaving and varied heating/cooling levels). We created different models: a *spatial* one in Figure 1.13 and a *declarative* one (finite state machine) in Figure 1.10. A student arriving in the classroom is an *event* of interest for this model. Two *state variables* will change upon occurrence of such event: the number of students in the room and the room's temperature (the composite change of these two attributes is the room's *state change*). This event will occur at the *instant* when the student enters the room (represented by a real number). The FSM and the spatial models introduced in previous figures are *abstract* representations of the actual system (for instance, the FSM in Figure 1.10 does not contain any information about the student's age or gender, which are attributes of the real system that we are not interested in modeling within our EF). A *simulator* for this model would implement the FSM in software, and a simulator's EF would generate a variety of independent tests to run the program. A different simulator would implement a software version of the spatial model in Figure 1.13. In order to *verify* the simulator, we need to check that the trajectories generated in runtime coincide with those defined by the model (for instance, if a new student arrival is simulated, we see the number of students increasing, and the temperature increasing, as described in the model). Finally, *validation* activities should ensure that what we see in the model and in the simulation results coincides with reality. (For instance, if we see temperatures decreasing when a new student arrives, we know that this simulated behavior is not valid; we have first to check the model and, if its specification is erroneous, we need to fix it. Otherwise, we need to modify the model's simulator.)

Once the study has been finished and the results are obtained, we might need to use them on the original system of interest. According to the objectives for the study and the decisions to be taken on the system of interest once it has finished, we can consider the following kinds of simulation models:

- *Exploration* models are models that are used to understand the operation of the system better.
- *Prediction* models are models that are used to predict the future behavior of the system.
- *Improvement* models are models that are used to optimize the performance of the system through the analysis of different alternatives.
- *Conception* models are models that are used when the system does not exist yet, and the model is used to test different options prior to construction.
- *Engineering design* models are models that are used to design devices in engineering applications (ranging from bridges to electron devices). Simulation permits exploring different design options and helps to choose the best.
- *Rapid prototyping* models are models that permit obtaining quickly a working model that can be used to test ideas and get early feedback from stakeholders.
- *Planning* models are models that provide a risk-free mechanism for thinking about the future in different fields of application (ranging from manufacturing to governance).
- *Acquisition* models are models that involve choosing proper equipment. Very large pieces of equipment (e.g., helicopters, airplanes, submarines) are extremely expensive. M&S can help us to decide during the purchasing process, enabling the customer to explore different alternatives without the need of constructing the equipment prior to taking the decision.
- *Proof of concept* models are models that can be used to test ideas and put them to work before creating the actual application.
- *Training* can be used to provide controlled experiments to enhance decision-making abilities and teach fundamental concepts for defense applications (also called *constructive* simulation). Other training examples include *business gaming* and *virtual* simulators (which are usually human-in-the-loop simulators to learn and enhance motor skills when operating complex vehicles).

- *Education* models are models that can be used in different sciences to provide insight into the nature of dynamic phenomena as well as the underlying mechanisms.
- *Entertainment* models are models that involve games and animation, the two most popular applications of simulation.

1.6 PHASES IN A SIMULATION STUDY

There have been different kinds of life cycles proposed for studies in modeling and simulation [17,38–41]. In this section, we summarize the basic steps that should be considered in doing a simulation study. The life cycle does not have to be interpreted as strictly sequential; it is iterative by nature, and sometimes transitions in opposite directions can appear. Likewise, some of the steps can be skipped, according to the complexity of the application. It is highly recommended to use a spiral cycle with incremental development for steps 2–8, which can cause a revision to earlier phases. Each phase in the spiral cycle should end with a working prototype including more functionality than the previous cycle:

1. *Problem formulation:* the simulation process begins with a practical problem that requires solving or understanding. It might be the case of a cargo company trying to develop a new strategy for truck dispatching or an astronomer trying to understand how a nebula is formed. At this stage, we must understand the behavior of the system of interest (which can be a natural or artificial system, existing or not), organizing the system's operation as objects and activities within the experimental framework of interest. Then we need to analyze different alternatives of solutions by investigating other previously existing results for similar problems. The most acceptable solution should be chosen (omitting this stage could cause the selection of an expensive or wrong solution). We also must identify the input/output variables and classify them into **decision** variables (controllable) or **parameters** (noncontrollable). If the problem involves performance analysis, this is the point at which we can also define performance **metrics** (based on the output variables) and an **objective function** (i.e., a combination of some of the metrics). At this stage, we can also do risk analysis and decide whether to follow or discard the project.
2. The *conceptual model* must be defined. This step consists of building a high-level description of the structure and behavior of the system and identifying all the objects with their attributes and interfaces. We also must define what the state variables are, how they are related, and which ones are important for the study. In this step, key aspects of the requirements are expressed (if possible, using a formalism, which introduces a higher degree of precision). During the definition of the conceptual model, we need to reveal features that are of critical significance (e.g., possibility of instability, deadlock, or starvation). We must also document nonfunctional information—for instance, possible future changes, nonintuitive (or nonformal) behavior, and the relation with the environment.
3. In the *collection and analysis of input/output data* phase, we must study the system to obtain input/output data. To do so, we must observe and collect the attributes chosen in the previous phase. When the system entities are studied, we try to associate them with a timing value. Another important issue during this phase is the selection of a sample size that is statistically valid and a data format that can be processed with a computer. Finally, we must decide which attributes are stochastic and which are deterministic. In some cases, there are no data sources to collect (for instance, for nonexisting systems). In those cases, we need to try to obtain data sets from similar systems (if available). Another option is to use a stochastic approach to provide the data needed through random number generation.
4. In the *modeling* phase, we must build a detailed representation of the system based on the conceptual model and the I/O data collected. The model is built by defining objects, attributes, and methods using a chosen paradigm. At this point, a specification model is

created, including the set of equations defining its behavior and structure. After finishing this definition, we must try to build a preliminary structure of the model (possibly relating the system variables and performance metrics), carefully describing any assumptions and simplifications and collecting them into the model's EF.

5. During the *simulation* stage, we must choose a mechanism to implement the model (in most cases using a computer and adequate programming languages and tools), and a simulation model is constructed. During this step, it might be necessary to define simulation algorithms and to translate them into a computer program. In this phase, we also must build a model of the EF for the simulation.
6. *V&V:* During the previous steps, three different models are built: the conceptual model (specification), the system's model (design), and the simulation model (executable program). We need to verify and validate these models. Verification is related to the internal consistency among the three models (is the model correctly implemented?). Validation is focused on the correspondence between model and reality: are the simulation results consistent with the system being analyzed? Did we build the right model? Based on the results obtained during this phase, the model and its implementation might need refinement. As we will discuss in the next section, the V&V process does not constitute a particular phase of the life cycle, but it is an integral part of it. This process must be formal and must be documented correctly because later versions of the model will require another round of V&V, which is, in fact, one of the most expensive phases in the cycle.
7. *Experimentation:* We must execute the simulation model, following the goals stated in the conceptual model. During this phase, we must evaluate the outputs of the simulator, using statistical correlation to determine a precision level for the performance metrics. This phase starts with the design of the experiments, using different techniques. Some of these techniques include sensitivity analysis, optimization, variance reduction (to optimize the results from a statistical point of view), and ranking and selection (comparison with alternative systems).
8. In the *output analysis* phase, the simulation outputs are analyzed in order to understand the system behavior. These outputs are used to obtain responses about the behavior of the original system. At this stage, visualization tools can be used to help with the process. The goal of visualization is to provide a deeper understanding of the real systems being investigated and to help in exploring the large set of numerical data produced by the simulation.

The rest of this book will concentrate mostly on phases 4, 5, and 6, using the DEVS formalism as the chosen paradigm. We will explain how to do advanced visualization for phase 8, and we will show how to conduct some experiments (phase 8). The focus here is on the practitioner's point of view, and those interested in understanding the details of the underlying theories should consult references 12, 17, 38, 39, and 41. Input/output analysis is exhaustively covered in references 6, 39, 42–44, and others. Likewise, we will not cover random number generation because it is covered in the previous references. We will focus on how to create an advanced simulation toolkit based on DEVS modeling and simulation methodology, concentrating on how to reduce the development costs of a simulation (by merging phases 2 and 4 and reducing phase 5) while providing good facilities for incremental development, V&V, experimentation, and maintenance.

1.7 VERIFICATION AND VALIDATION (V&V)

The credibility of the results of the simulation depends not only on the correctness of the model, but also on how accurate the formulation of the problem is expected to be [45]. Thus, we must use various V&V techniques throughout the life cycle of the simulation study. As discussed earlier, we call *validation* the process of determining that a model is a correct representation of the real world and that its behavior corresponds to the requirements of the model (i.e., validation is related to the

correct formulation of the model). We say a model is *valid* when it is impossible to distinguish system and model within the experimental frame. We can recognize different types of validity [17]:

- *Replicative validity:* For every possible experiment within the experimental frame, trajectories of model and system agree within acceptable tolerance.
- *Predictive validity:* Within an experimental frame, it is possible to initialize the model to a state such that, for the same input trajectory, model output trajectory predicts system output trajectory within acceptable tolerance.
- *Structural validity:* The model is able to replicate the data observed from the system but also mimics step by step and component by component the way in which the system does its transitions.

Verification, on the other hand, is the process of checking that a simulator of a model correctly generates its behavior according to the model's specification. A *correct simulation* faithfully generates model behavior in every simulation run.

In large organizations (mostly related to M&S in the defense industry), an *accreditation* phase can be required. This is the official process of gaining approval for model use, which certifies that the model satisfies the specifications. *Specific* accreditation includes the model and its environment (input data, aptitude of the personnel, performance of the simulator, etc.).

The result of the activities of V&V must not be considered in absolute form (e.g., right/wrong). A simulation model is constructed with certain objectives described in the conceptual model and the experimental frame. As in any software project, it is recommended that V&V should be executed by an independent team to prevent biased decisions [45]. Likewise, the software cannot be tested completely, and standard testing techniques should be used, trying to cover the largest percentage of cases within the domain [46]. The objective is to increase confidence in the model, based on the study of the objectives. Likewise, the satisfactory test of each submodel (unit test) does not imply the correctness of the whole (integration test).

Following the life cycle described in Figure 1.26, we need to carry out the following activities:

1. Verification of the conceptual model and of the problem under analysis: We must determine that the problem we are solving corresponds to the real system. It is essential to understand the requirements of the system of interest, justifying all the assumptions made during the definition of the conceptual model. We have to check that such assumptions are appropriate and that the conceptual model represents the real system based on the proposed objectives.
2. Verification of the design: The purpose of design verification is to ensure that the specifications reflect the conceptual model accurately.
3. Validation of the system's model: We must verify that the model corresponds to the system of interest within the EF. We must also verify that the model is represented with a sufficient degree of accuracy. As part of this process, we must ensure the quality of the data used in the different phases of the model.
4. Verification of the simulator: We must ensure that the software implementation of the model reflects its specification. The model's behavior must be tested to cover the maximum percentage of cases possible.
5. Validation of the experimental results: The credibility of the model is the result of having completed each of the different activities of V&V mentioned earlier satisfactorily and ensuring that the results given by the simulator are compatible with those of the real system.

A variety of V&V techniques can be employed (discussed in detail in references 40, 43, and 45–47), which can be categorized as the following:

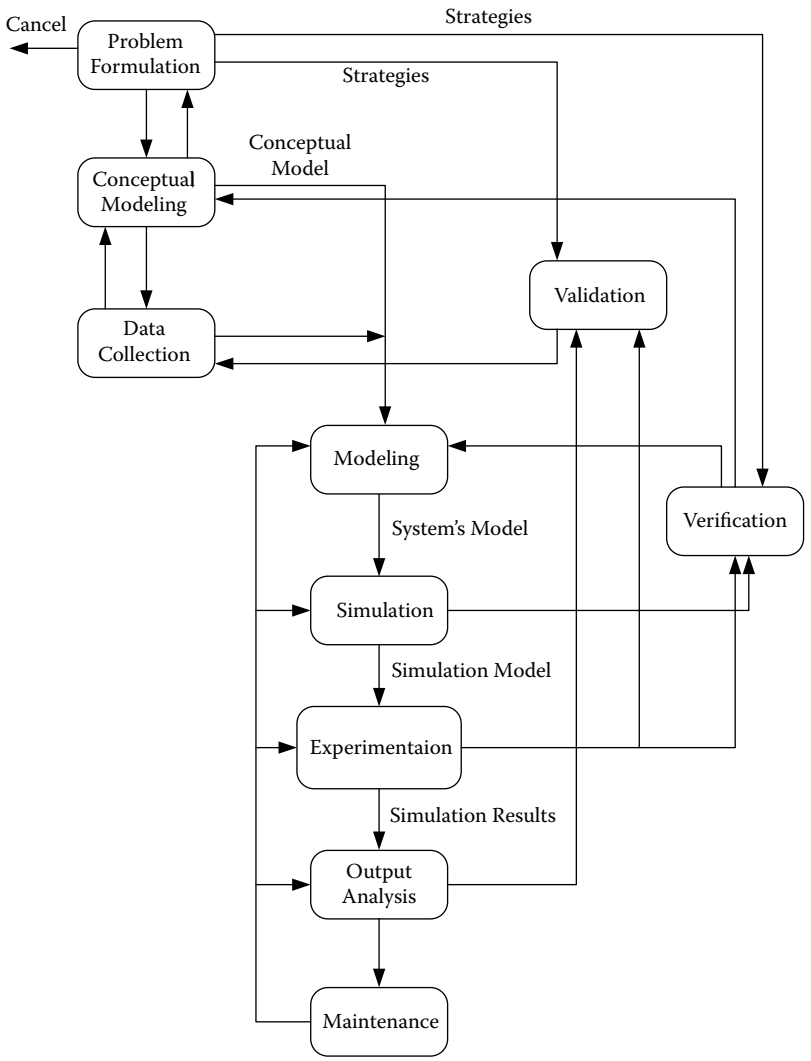


FIGURE 1.26 Steps in M&S studies.

- *Informal* techniques lack mathematical foundations and are based mainly on human reasoning. These are the most common techniques, including inspections, Turing test (i.e., making an expert interact with the simulator, trying to make sure the expert cannot recognize the differences between the simulator and the real system), and animation (in which the behavior of the model can be represented using animated graphics). Other informal techniques are the comparison with other models (i.e., the experimental results are compared with the results from other existing models) or the opinions of experts.
- *Static* techniques are used to validate the design of the model and the source code (they do not execute the model). These techniques include the analysis of data flow, analysis based on graphics, and syntactic and semantic analyses.
- *Dynamic* techniques require the execution of the model and validation of its behavior. Some of the techniques include the symbolic execution of the model, cause–effect graphs, regression tests, and stress testing. Other techniques include reachability analysis, degenerative testing (inaccurate or limited input data are used to test the model), and sensitivity analysis (which consists of changing the values of the input as well as those of some

internal parameters in order to observe the reaction of the model to these variations). Varied statistical techniques (including histograms, confidence intervals, or hypothesis tests) are commonly used.

- *Formal* techniques try to prove the correctness of the model using mathematical techniques. They present a high degree of difficulty and serve as a base for other techniques of V&V. This can include induction techniques, inference models, deduction logic, predictive calculus, correction proofs, bisimulation, and model checking.

1.8 SUMMARY

In this chapter, we have given basic ideas and concepts on modeling and simulation. We have presented a brief historical perspective in the field, showing different techniques used for problem solving. We clearly showed the concepts of a system (a formalization of entities under study), the separation between them, and the models we use to reason about the system itself. We briefly introduced different methods to define simulators that will drive the execution of the models (in algorithmic fashion or using specialized devices, including digital computers).

We introduced the concept of the experimental framework, which gives an environment for defining the model, conducting experiments, collecting data, and executing the simulation analysis. The clear separation between the system of interest, the model that represents it, the simulator that executes it, and the experimental frames to conduct tests will make the creation, modification, and use of the simulation tools much easier.

Finally, we introduced varied discrete-event modeling techniques and discussed different classifications, focusing on discrete-event modeling and simulation methodologies (the main focus of this book). We then introduced some term definitions, presented a life cycle proposal for M&S studies, and gave a brief introduction to ideas on verification and validation activities.

REFERENCES

1. Taylor, M. 1996. *Partial differential equations: Basic theory.* New York: Springer–Verlag.
2. Bacon, D. H. 1989. *BASIC heat transfer.* London: Butterworth & Co. Ltd.
3. Lapidus, L., and G. F. Pinder. 1982. *Numerical solution of partial differential equations in science and engineering.* New York: Wiley.
4. Brenan, K. E., S. L. Campbell, and L. R. Petzold. 1989. *Numerical solution of initial-value problems in differential algebraic equations.* New York: Elsevier.
5. Coward, D. Analog computer museum and history center. http://dcoward.best.vwh.net/analog/. Accessed: September 1, 2007.
6. Cassandras, C. G. 1993. *Discrete event systems: Modeling and performance analysis.* Homewood, IL: Aksen: Irwin.
7. Hopcroft, J. E., R. Motwani, and J. D. Ullman. 2007. *Introduction to automata theory, languages, and computation,* 3rd ed. Boston: Pearson/Addison–Wesley.
8. Ho, Y. C. 1989. Introduction to special issue on dynamics of discrete event systems. *Proceedings of the IEEE,* 77:3–6.
9. Birtwistle, G. M. 1979. *A system for discrete event modeling on SIMULA.* London: Macmillan.
10. IFIP Technical Committee 2—Programming. 1968. Simulation programming languages. *Proceedings of the IFIP Working Conference on Simulation Programming Languages,* Oslo, Norway.
11. Pritsker, A. B., and C. D. Pegden. 1979. *Introduction to simulation and SLAM.* New York: Wiley (distributed by Halsted Press).
12. Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.
13. Fishwick, P. A. 1995. *Simulation model design and execution: Building digital worlds.* Englewood Cliffs, NJ: Prentice Hall.
14. Zeigler, B. 1984. *Multifaceted modeling and discrete event simulation.* New York: Academic Press.
15. Zeigler, B. P. 1990. *Object-oriented simulation with hierarchical, modular models: Intelligent agents and endomorphic systems.* Boston: Academic Press.

16. Zeigler, B. P. 1998. DEVS theory of quantization. Technical report, DARPA contract N6133997K-0007, ECE Department, the University of Arizona, Tucson.
17. Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation,* 2nd. ed. New York: Academic Press.
18. Zeigler, B. P. 2005. Continuity and change (activity) are fundamentally related in DEVS simulation of continuous systems. *Proceedings of AIS 2004, Artificial Intelligence, Simulation and Planning,* Jeju Island, Korea, 1–17.
19. Kofman, E. 2004. Discrete event simulation of hybrid systems. *SIAM Journal of Scientific and Statistical Computing* 25:1771–1797.
20. Cellier, F. E., and E. Kofman. 2006. *Continuous system simulation.* New York: Springer Science+ Business Media.
21. Alur, R., and D. L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126:183–235.
22. Papadimitriou, C. H. 1994. *Computational complexity.* Reading, MA: Addison–Wesley.
23. Cao, X. R., and Y. C. Ho. 1990. Models of discrete event dynamic systems. *IEEE Control Systems Magazine* 10:69.
24. Glynn, P. W. 1989. A GSMP formalism for discrete event systems. *Proceedings of the IEEE* 77:14–23.
25. Peterson, J. L. 1981. *Petri net theory and the modeling of systems.* Englewood Cliffs, NJ: Prentice Hall.
26. Woodside, C. M. 1988. Throughput calculation for basic stochastic rendezvous networks. *Performance Evaluation* 9:143–160.
27. Rolia, J., and K. C. Sevkik. 1995. The method of layers. *IEEE Transactions on Software Engineering* 21:682–688.
28. Woodside, C. M., S. Majumdar, J. E. Neilson, D. C. Petriu, J. Rolia, A. Hubbard, and G. Franks. 1995. A guide to performance modeling of distributed client–server software systems with layered queuing networks. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada.
29. Milner, R. 1982. *A calculus of communicating systems.* New York: Springer–Verlag.
30. Manna, Z., and A. Pnueli. 1992. *The temporal logic of reactive and concurrent systems.* New York: Springer–Verlag.
31. Hoare, C. A. R. 1985. *Communicating sequential processes.* Englewood Cliffs, NJ: Prentice Hall International.
32. Harel, D., and M. Politi. 1998. *Modeling reactive systems with statecharts.* New York: McGraw–Hill.
33. Rumbaugh, J., I. Jacobson, and G. Booch. 1999. *The unified modeling language reference manual.* Essex, UK: Addison-Wesley Longman Ltd..
34. Belina, F., and D. Hogrefe. 1989. The CCITT-specification and description language SDL. *Computer Networks ISDN Systems* 16:311–341.
35. Schruben, L. W., T. M. Roeder, W. K. Chan, P. Hyden, and M. Freimer. 2003. Advanced event scheduling methodology. *Proceedings of WSC '03: Proceedings of the 35th Winter Simulation Conference,* New Orleans, 159–165.
36. von Bertalanffy, L. 1969. *General system theory: Foundations, development, applications.* New York: G. Braziller.
37. Nance, R. E. 1981. The time and state relationships in simulation modeling. *Communications of the ACM* 24:173–179.
38. Sargent, R. G. 2005. Verification and validation of simulation models. *Proceedings of WSC '05: Proceedings of the 37th Winter Simulation Conference,* 130–143, Orlando.
39. Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis,* 3rd ed. Boston: McGraw–Hill.
40. Balci, O. 1994. Validation, verification, and testing techniques throughout the life cycle of a simulation study. *Proceedings of WSC '94: Proceedings of the 26th Winter Simulation Conference,* 215–220, Orlando.
41. Lutz, R. 1999. FEDEP V1.4: An update to the HLA process model. *Proceedings of WSC '99: Proceedings of the 31st Winter Simulation Conference,* Phoenix, AZ, 1044–1049.
42. Banks, J., J. S. Carson, B. L. Nelson, and D. Nicol. 2005. *Discrete-event system simulation,* 4th ed. Upper Saddle River, NJ: Prentice Hall.
43. Kleijnen, J. P. C. 2005. Forty years of statistical design and analysis of simulation experiments (DASE). *Proceedings of WSC '05: Proceedings of the 37th Winter Simulation Conference,* Orlando, 1–17.
44. Leemis, L. M., and S. K. Park. 2005. *Discrete-event simulation: A first course.* Upper Saddle River, NJ: Prentice Hall.
45. Pace, D. K. 1993. Naval modeling and simulation verification, validation, and accreditation. *Proceedings of WSC '93: Proceedings of the 25th Winter Simulation Conference,* Los Angeles, 1077–1080.

46. Beizer, B. 1990. *Software testing techniques,* 2nd. ed. New York: Van Nostrand Reinhold Co.
47. Sargent, R. G., P. A. Glasow, J. P. C. Kleijnen, A. M. Law, I. McGregor, and S. Youngblood. 2000. Strategic directions in verification, validation, and accreditation research. *Proceedings of WSC '00: Proceedings of the 32nd Winter Simulation Conference,* Orlando, 909–916.

2 Introduction to the DEVS Modeling and Simulation Formalism

2.1 INTRODUCTION

As discussed in our previous chapter, the theory of differential equations has a long history, while modeling techniques for discrete-event systems have only appeared recently. Discrete-event simulation techniques were developed hand in hand with the creation of the computer. Consequently, the first discrete-event modeling and simulation (M&S) approaches were tightly coupled to the computer hardware and languages, while formal modeling techniques with a solid mathematical background are more recent. Discrete-event systems specification (DEVS) M&S theory is one of such techniques that were based on systems theory concepts [1–3]. Although in this chapter we will briefly introduce some of the basic ideas behind this theory of modeling and simulation, the reader should refer to Zeigler [1]; Zeigler, Praehofer, and Kim [4]; and Zeigler [5] to understand the details behind the mathematical background of this technique. Our goal here is to provide the basis to discuss the application examples introduced in the following chapters. As we will briefly discuss here, DEVS also provides a formal background for a common methodology for modeling both discrete and continuous worlds (the interested reader should consult Zeigler et al. [4] and Cellier and Kofman [6].

According to DEVS theory, the system of interest is seen as a source of behavioral data for the study within a given experimental frame (EF). The EF is a restricted set of the elements observed in the system and the conditions under which they are observed. These data are used to create an abstract representation of such a system (a model). Using a set of instructions, rules, or mathematical equations, the model tries to replicate the behavior of the system of interest under the experimental conditions. Figure 2.1 shows these basic entities in M&S and their relationships [4].

This separation of concerns and the use of a formal mechanism to describe each of the components allowed DEVS methodology to improve the creation of models and the execution of simulations. The formal specifications provide means for mathematical manipulation and it permit independence of the language chosen to implement the models.

The model represents a simplified version of reality and its structure. The model is built considering the conditions of experimentation of the system of interest, including the work conditions of the real system and its application domain. Thus, the model is restricted to the experimental framework under which it was developed (which influences its construction, experimentation tasks, and validation).

The model is subsequently used to build a simulator (i.e., a device capable of executing the model's instructions) generating the model's behavior. When we implement the model, there is an extra reduction in the precision. If the model is implemented in a computer, the programming languages and the computer used are limited, including the duration of the simulation, memory availability, precision of the variables involved, etc.

Figure 2.1 also shows the two fundamental relationships discussed in Chapter 1: verification and validation. Validation links the model with the real system within the experimental frame (i.e.,

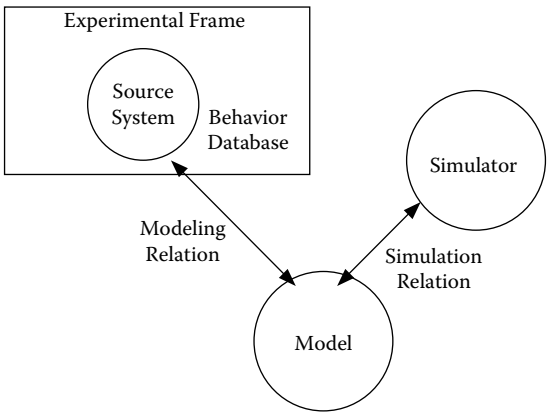


FIGURE 2.1 Basic entities in M&S and their relationships. (From Zeigler, B. P. et al. 2000. *Theory of modeling and simulation,* 2nd. ed. New York: Academic Press.)

how well the model's behavior agrees with the system's behavior under the conditions specified in the experimental frame). Verification links a simulator and its model, dealing with the accuracy of the simulator to execute the instructions of the model (this can be related to the precision of the computer, the simulation algorithms, etc.).

DEVS was created for modeling and simulating discrete-event dynamic systems (DEDS); thus, it defines a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. In order to attack the complexity of the system under study, the model is organized hierarchically (i.e., it is organized in a way such that every element is higher than its precedent), and higher-level components of the system are decomposed into simpler elements. The second tool used to attack complexity is information hiding, through the provision of a modular interface for each of the models.

The separation between model and simulator and the hierarchical and modular nature of the formalism have enabled carrying out of formal proofs on the different entities under study. One of them is the proof of composability of the subcomponents (including legitimacy and equivalence between multicomponent models). The second is the ability to conduct proofs of correctness of the simulation algorithms, which results in simulators rigorously verified. The proofs are based on formal transformations (*morphisms*) between each of the representations, trying to prove the equivalence between the entities under study at different levels of abstraction [4,7–9]. For instance, we can prove that the mathematical entity *simulator* is able to execute correctly the behavior described by the mathematical entity *model*, which represents the system under the experimental framework (which can also be represented formally).

Different mechanisms are used to prove this, including the mathematical manipulation of the abstraction hierarchy, observation of the I/O trajectories (in order to ensure that different levels of specification correctly describe the system's structure), and decomposition concepts (DEVS is closed under composition, which means that a composite model integrated by multiple components is equivalent to an atomic component).

2.2 THE DEVS FORMALISM

A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components [1,4]. Here, we will use the definition of *DEVS with ports* [4] (instead of *classic* DEVS [1]). An *atomic* model is specified as

$$M = <X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta> \tag{2.1}$$

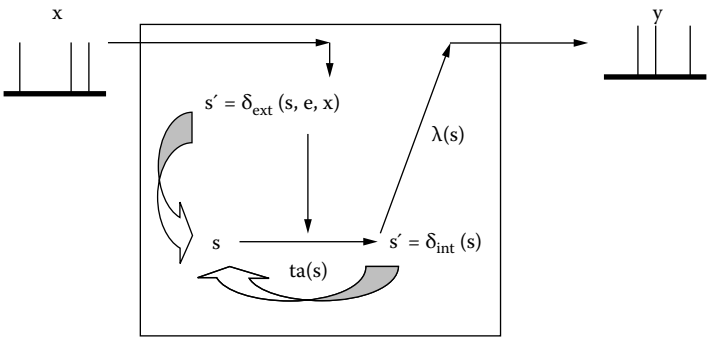


FIGURE 2.2 DEVS atomic model semantics.

where

$X = \{(p,v)|p \in IPort\text{s}, v \in X_p\}$ is the set of *input* events, where *IPorts* represents the set of input ports and X_p represents the set of values for the input ports;

$Y = \{(p,v)|p \in OPort\text{s}, v \in Y_p\}$ is the set of *output* events, where *OPorts* represents the set of output ports and Y_p represents the set of values for the output ports;

S is the set of sequential *states*;

$\delta_{ext}: Q \times X \rightarrow S$ is the *external* state transition function, with $Q = \{(s,e)/s \in S, e \in [0, ta(s)]\}$ and e is the elapsed time since the last state transition;

$\delta_{int}: S \rightarrow S$ is the *internal* state transition function;

$\lambda: S \rightarrow Y$ is the *output* function; and

$ta: S \rightarrow R_0^+ \bigcup \infty$ is the *time advance* function.

Figure 2.2 shows an informal depiction of DEVS atomic models.

At any given moment, a DEVS model is in a state $s \in S$. In the absence of external events, it remains in that state for a lifetime defined by $ta(s)$. When $ta(s)$ expires, the model outputs the value $\lambda(s)$ through a port $y \in \gamma$, and it then changes to a new state given by $\delta_{int}(s)$. A transition that occurs due to the consumption of time indicated by $ta(s)$ is called an internal transition. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$, where s is the current state, e is the time elapsed since the last transition, and $x \in X$ is the external event that has been received.

The time advance function can take any real value between 0 and ∞. A state for which $ta(s) = 0$ is called a *transient* state (which will trigger an *instantaneous* internal transition). In contrast, if $ta(s) = \infty$, then s is said to be a *passive* state, in which the system will remain perpetually unless an external event is received (can be used as a termination condition).

A DEVS *coupled model* is composed of several atomic or coupled submodels. It is formally defined by

$$CM = <X, Y, D, \{M_d|d \in D\}, EIC, EOC, IC, select> \tag{2.2}$$

where

$X = \{(p,v)|p \in IPort\text{s}, v \in X_p\}$ is the set of *input* events, where *IPorts* represents the set of input ports and X_p represents the set of values for the input ports;

$Y = \{(p,v)|p \in OPorts, v \in Y_p\}$ is the set of *output* events, where *OPorts* represents the set of input ports and Y_p represents the set of values for the output ports;

D is the set of the component names and for each $d \in D$;

M_d is a DEVS basic (i.e., atomic or coupled) model;

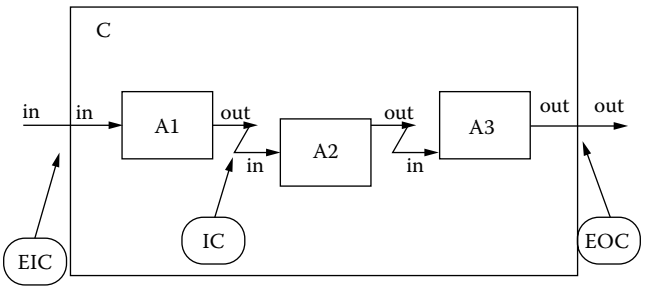


FIGURE 2.3 A coupled model.

EIC is the set of external input couplings, $EIC \subseteq \{ ((Self, in_{Self}), (j, in_j)) | in_{Self} \in IPorts, j \in D, in_j \in Iports_j \}$;

EOC is the set of external output couplings, $EOC \subseteq \{ ((i,out_i), (Self,out_{Self})) | out_{Self} \in OPorts, i \in D, out_i \in OPorts_i \}$;

IC is the set of internal couplings, $IC \subseteq \{ ((i,out_i), (j,in_j)) | i,j \in D, out_i \in OPorts_i, in_j \in IPorts_j \}$; and

select is the tiebreaker function, where $select \subseteq D \rightarrow D$, such that, for any nonempty subset E, $select(E) \in E$.

Figure 2.3 shows an example of a DEVS coupled model with three subcomponents, A1–A3. These basic models are interconnected through the corresponding I/O ports presented in the figure. The models are connected to the external coupled model through the EIC and EOC connectors. Keep in mind that A1–A3 are *basic* models (i.e., they can be atomic or coupled components).

The model depicted in Figure 2.3 can be formally defined as

$$C = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select> \tag{2.3}$$

where

X = {(in, v)|in $\in$ *IPorts*, $v \in \boldsymbol{R}$};
Y = {(out, v)|out $\in$ *IPorts*, $v \in \boldsymbol{R}$ };
D = {A1, A2, A3};
M_d = { M_{A1}, M_{A2}, M_{A3} };
EIC $\subseteq$ { ((Self, in), (A1, in)) }; (or *EIC* $\subseteq$ { ((C, in), (A1, in)) });
EOC $\subseteq$ { ((A3, out), (Self, out)) }; (or *EOC* $\subseteq$ { ((A3, out), (C, out)) });
IC $\subseteq$ { ((A1, out), (A2, in)); ((A2, out), (A3, in)) };
select = {A3, A1, A2}.

The coupled model definition presented shows the specification of the three components A1–A3 and their internal/external couplings. Coupled models group several DEVS into a composite model that can be regarded, due to the *closure property,* as a new DEVS model. The closure property guarantees that the coupling of several class instances results in a model of the same class, allowing hierarchical construction [4].

Because multiple subcomponents can be scheduled for an internal transition at the same time, ambiguity could arise. In our example, if A1 executes its output/internal transition first, producing an output that maps into an external event for A2 (which is also scheduled for an internal transition at the same time), then it is not clear which transition this second component should execute first. There are two alternatives for this:

(a) to execute the external transition first and then the internal transition, with $e = ta(s)$; or
(b) to execute the internal transition first, followed by the external transition, with $e = 0$.

The *select* function provides a simple way to solve this ambiguity. The function defines an ordering over all the components of the coupled model so that only the first model to execute in the case of simultaneous internal events can be chosen. In our example, A1 is executed before A2; thus, we execute the external transition first.

A different definition of coupled models (that we will be using later in this and other chapters) is

$$\text{CM} = < X, Y, \text{D}, \{\text{M}_\text{i}\}, \{\text{I}_\text{i}\}, \{\text{Z}_\text{ij}\}, \text{select} > \tag{2.4}$$

where

X is the set of input events;
Y is the set of output events;
D is an index for the components of the coupled model;
$\forall i \in \text{D}$, $\mathbf{M}_\text{i}$ is a basic DEVS model;
$\mathbf{I}_\text{i}$ is the set of influencees of model i and $\forall j \in I_i$;
$\mathbf{Z}_\text{ij}$ is the i to j translation function, where Z_ij: $\text{Y}_\text{i} \rightarrow \text{X}_\text{j}$

Figure 2.4 shows an example of the execution of a DEVS atomic model (which, due to the closure under coupling property, could also be representing the I/O trajectories of a DEVS coupled model).

Initially, the model is in state s_0, and an internal transition is scheduled for $ta(s_0)$. Time advances (i.e., the elapsed time variable e moves forward in continuous time) and, at time t_0 (before the scheduled time is consumed), the model receives the internal input X_0. Consequently, the external

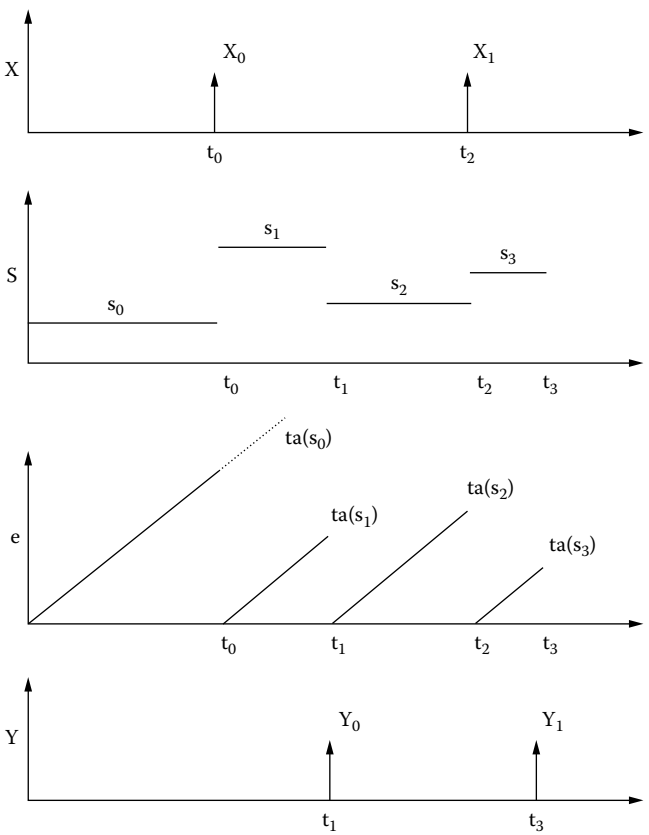


FIGURE 2.4 Sample I/O trajectories.

transition function is triggered, and the model changes to s_1, which has an associated time advance of $ta(s_1)$ (at this point, the elapsed time e is reset to 0). In this case, no inputs are received before the internal event, and when time $t_1 = ta(s_1)$ arrives, we activate the output function (which generates the output Y_0). We then execute the internal transition function (which will make the model change to the state s_2 and schedule the next internal transition at $ta(s_2) = t_2$). At time t_2, we simultaneously receive the external input X_1 (in order for this to happen, the *select* function at the parent coupled model must have triggered an influencer, which executed its output function before our atomic model). Thus, we consume the input event X_1, triggering the external transition function (which makes the model change to s_3 with $ta(s_3) = t_3$). At this time, we trigger both the output function (which generates the output Y_1) and the internal transition function. This cycle is repeated until the end of the simulation.

2.3 A DEVS MODEL EXAMPLE

The Generator, Processor, Transducer (GPT) model presented in this section has been widely used as a "Hello, world!" example for DEVS modeling and simulation [4]. The structure of the model is introduced in Figure 2.5.

The top-level model *GPT* is a simple coupled model that is composed of two basic components: *generator* and *QPT. Generator* is an atomic model that creates jobs to be processed (at random times) and sends them through the *out* output port. *QPT* is a coupled model consisting of two main atomic components: a *processor* that consumes the jobs received (and informs that they are ready through the *out* output port) and a *transducer* in charge of calculating statistics. When a new job arrives through the *arrived* input port, the *transducer* computes the arrival time; when the job finishes, its end time arrives through the *solved* port, and we can use this information to compute metrics. In this case, we have also included a *queue* model, which is used as a buffer for the arriving jobs before they are processed. Based on Figure 2.5, we can define the coupled model for this example as

$$M_{GPT} = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select \rangle \tag{2.5}$$

where

X = ∅;
Y = {(cpuUsage, $\boldsymbol{R}_0^+$); (Throughput, $\boldsymbol{R}_0^+$) };
D = { Generator, QPT };
M_d = { $M_{Generator}$, M_{QPT} } (where $M_{Generator}$ is an atomic model and M_{QPT} a coupled one);
EIC = ∅;
$EOC \subseteq$ { ((QPT, cpuUsage), (Self, cpuUsage)) ; ((QPT, Throughput), (Self, Throughput)) };

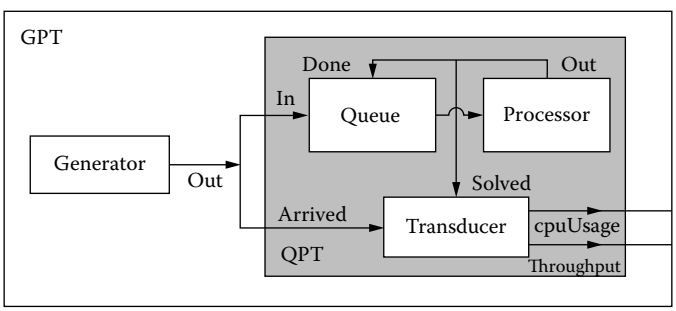


FIGURE 2.5 The GPT model and its internal and external connections.

$IC \subseteq$ { ((Generator, out), (QPT, in)); ((Generator, out), (QPT, arrived)) }; and
select = {QPT, Generator }.

The QPT coupled model can be defined as

$$M_{QPT} = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select> \quad (2.6)$$

where
X = {(in, N); (arrived, N) };
Y = {(cpuUsage, $\boldsymbol{R_0}^+$); (Throughput, $\boldsymbol{R_0}^+$) };
D = { Queue, Processor, Transducer};
M_d = { M_{Queue}, $M_{Processor}$, $M_{Transducer}$ };
EIC = { ((Self, in), (Queue, in)) ; ((Self, arrived), (Transducer, arrived)) };
$EOC \subseteq$ { ((Transducer, cpuUsage), (Self, cpuUsage)); ((Transducer, Throughput), (Self, Throughput)) };
$IC \subseteq$ { ((Queue, out), (Processor, in)); ((Processor, out), (Queue, done)); ((Processor, out), (Transducer, solved)) };
select = {Processor, Queue, Transducer }.

The coupled model definitions show the structure of the whole model, but then we need to define the behavior for each atomic model, which should use the following descriptions:

- **Generator** generates new tasks transmitted through an output port. The output value represents a task identifier (a positive integer uniquely used during the simulation process). The period used to create a new process is generated at random (with probability distributions chosen during the configuration of the experiment).
- **Processor** simulates the tasks' execution delays. A new task ID (a positive integer number) is received through an input port, and the processor remains busy until the processing is finished. Then it sends the process identifier through an output port. The processing time is generated using random numbers with exponential distribution.
- **Queue** is a buffer that stores task IDs (positive integer numbers). When an ID is received through the *done* input port, the buffer must transmit a stored job (if available). The queue uses a nonpreemptive first in, first out (FIFO) policy. A *stop* input port is used to deactivate/reactivate the queue, allowing control flow by higher-level models (this port has not been used in the example introduced in Figure 2.5).
- **Transducer** records metrics and computes statistics. Two measures are considered: throughput (tasks executed per time unit) and CPU usage (average time of tasks waiting in the ready queue). The transducer accepts job IDs on the *arrived* input port and records the time for arrival of the job. Jobs processed must be forwarded to the transducer's *solved* input port so that the transducer can record the time when the job was finished and calculate the elapsed time. This value is subsequently used for calculating throughput and CPU usage.

The functionality of each of these models must be described using the formal specification for DEVS. For instance, the queue model can be formally described as

$$\text{Queue} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \quad (2.7)$$

X = {(in, $\boldsymbol{N}$); (stop, N); (done, $\boldsymbol{N}$) };
S = { state ∈ {active, passive}, preparationTime, timeLeft ∈ $\boldsymbol{R}_0$+, queue ∈ {jobid ∈ $\boldsymbol{N}$}* };
Y = {(out, $\boldsymbol{N}$); };

```
δext (s, e, x) {
    if ( x.port == in ) {                          // A new job has arrived
      add (x.value, s.queue);                      // Add it to the queue
      if ( sizeOf (s.queue) == 1 )                 // The queue was empty
        state = active; ta(state) = preparationTime ;   // The arrived job must be executed.
    }
    if( x.port == done ) {                         // A job has finished
      delete_first (s.queue);                      // Delete it from the queue
      if( !empty(s.queue) )                        // Take the next element in the queue
        state = active; ta(state) = preparationTime ;   // This job must be executed
    }
    if(x.port == stop )                            // Stop the transmision: buffer overflow
      if( state == active && x.value != 0) {       // The queue was active
        timeLeft = preparationTime – e ;           // Record the time left to execute
          state = passive; ta(state) = infinity;   // Deactivate the queue: passivate.
        }
        else                                       // Reactivate the queue
          if( state == passive && x.value == 0 )
            state = active; ta(state) = timeLeft ;      // Use the time left before being stopped
  }

δ ( s ) {
    sendOutput(time, out, first (queue));
}

δint ( s ) {
    passivate();
}
```

The transducer model collects timing information of the jobs arriving in the system and their departure times. Using these events, it computes the number of jobs processed per time unit (*throughput*) and the level of utilization of the CPU (*cpuUsage*). The model can be formally defined as

$$\text{Transducer} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{2.8}$$

$X =$ {(arrived, $\boldsymbol{N}$); (solved, $\boldsymbol{N}$); (done, $\boldsymbol{N}$) };
$S =$ state $\in$ {active, passive}, procCount $\in \boldsymbol{N}$, cpuLoad, frequence $\in \boldsymbol{R}_0+$, unsolvedQ $\in$ {jobid $\in \boldsymbol{N}$}* };
$Y =$ {(throughput, $\boldsymbol{R}_0+$); (cpuUsage, $\boldsymbol{R}_0+$); };

```
δext (s, e, x) {
  cpuLoad += (time – lastChange) * size_of(unsolvedQ) ; // Average load
  if( x.port == arrived ) unsolvedQ[ x.value ] = time;      // Store information about task
  if( x.port == solved ) {                                  // Task ended: erase
    which = find( x.value, unsolved ) ;
    procCount ++ ;
    erase( which ) ;
  }
}
```

λ (s) {
    sendOutput(time, throughput, procCount/time);
    sendOutput(time, cpuUsage, cpuLoad/time);
}

δ_{int} (s) {
    passivate();
}

This model will be used later for varied examples throughout the book.

Exercise 2.1

Modify the *queue* model to implement a LIFO strategy.

Exercise 2.2

Modify the *queue* model to implement a priority-based strategy. Job numbers also represent the priority of the job; thus, every arriving job should be located in the right position in the queue (using the ID number).

Exercise 2.3

Write the model specification for the processor model. Include two different versions: one without preemption and one with preemption (i.e., a newly arriving job will stop the execution of the current job and will start the new one).

Exercise 2.4

Compute the input/output trajectories for the queue model for the following three jobs: (1, 0.3 s), (2, 5.1 s), (3, 10.6 s), where the first number is the job ID and the second the arrival time.

Exercise 2.5

Compute the input/output trajectories for each atomic model and the whole coupled model using the same input trace used in Exercise 2.4.

Exercise 2.6

Use the internal transition function in the queue model to represent a faulty buffer. Every time the size of the queue is a multiple of 13, one element in the queue is deleted.

Exercise 2.7

Modify the mechanism for computing the CPU load in the transducer. In this case, use an accumulator to keep track of the total use of the CPU (i.e., add all the time between arrival/departure of jobs) instead of the average used in the original version.

Exercise 2.8

Add a new model, *ControlFlow*, which will stop or reactivate the queue model according to its internal state. A random number is used to decide when the queue should be stopped. The internal transition function will generate a random number using a normal distribution with average 5 and standard deviation 3. If the number generated is larger than 9, then the output function will generate a "stop" signal for the queue. Then, if the number generated is smaller than 8, the queue will be reactivated. Write a formal specification for this model and modify the corresponding coupled model.

2.4 DEVS WITH SIMULTANEOUS EVENTS (PARALLEL DEVS)

As seen in the previous section, whenever two models are scheduled for state transitions at the same time, a DEVS coupled model will pick the one specified by the *select* function to execute first. This tie-breaking strategy is rigid. Let us suppose that model A2 in Figure 2.3 represents vehicles going into an intersection, and A3 represents vehicles inside the crossing area. According to the *select* function definition, A3 has the highest priority (which tries to free traffic in the crossing area before allowing new vehicles in the intersection). If now we want to be able to represent collisions, we would need to give priority to A2; however, this is not possible in the current specification, which will free space in the crossing first (making it more difficult to represent the collision situation). In addition, *select* introduces serialization in the execution of components when many interconnected atomic models are imminent (which could be executed in parallel in a multiprocessor environment).

Parallel DEVS (or PDEVS) is an extension to DEVS that provides a more flexible way of dealing with these ambiguities [10]. Atomic models provide an additional *confluent* function to specify collision behavior for events that might be scheduled simultaneously and a mechanism for receiving multiple external events at the same time and processing them together. An atomic PDEVS model is defined as

$$M = <X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta> \tag{2.9}$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$	is the set of *input* events, where *IPorts* represents the set of input ports and X_p represents the set of values for the input ports;
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$	is the set of *output* events, where *OPorts* represents the set of input ports and Y_p represents the set of values for the output ports;
S	is the set of sequential states;
δ_{ext}: $Q \times X^b \rightarrow S$	is the external state transition function;
δ_{int}: $S \rightarrow S$	is the internal state transition function;
δ_{con}: $Q \times X^b \rightarrow S$	is the confluent transition function;
λ: $S \rightarrow Y^b$	is the output function;
ta: $S \rightarrow R_0^+ \cup \infty$	is the time advance function, with $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ the set of total states.

PDEVS models use bags (multisets) of events for receiving inputs and collecting outputs (X^b, Y^b) instead of a single event. This allows multiple events to be processed simultaneously. Because external input events received by the component are added to the bag, external transition functions can combine the functionality of a number of external transitions into a single one, and simultaneous events (like the departure of a vehicle and a collision in the intersection) can be treated simultaneously. Also, PDEVS allows a better way to deal with collisions: the model specification includes a confluent transition function (δ_{con}). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The semantics of PDEVS for internal/external transition functions is similar to DEVS. If one or more external events $X^b = \{x_1 \ldots x_n / x_i \in X\}$ occur before $ta(s)$ expires (i.e., while the system is in total state (s, e) with $e < ta(s)$), the new state will be given by the model's external transition function, $\delta_{ext}(s,e,X^b)$. If the external events X^b are received when $e = ta(s)$, the new state of the model will be given by the confluent function (δ_{con}). If multiple components in a coupled model are imminent, all their outputs are first collected and mapped to their influences in parallel. Then the corresponding transition function is executed for every model.

In PDEVS, coupled models are defined as in DEVS, without the need for a *select* function. Formally, a coupled model is defined as

$$CM = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC> \tag{2.10}$$

where definitions for the set of input and output events (X and Y), components (D and M_d), and couplings (*EIC, EOC,* and *IC*) follow the specifications of DEVS coupled models presented earlier in this chapter.

2.5 DYNAMIC STRUCTURE DEVS

The definitions of DEVS presented in the previous sections consider a model with a static structure (i.e., invariant in time). Nevertheless, in many cases it is useful to allow modeling of the dynamic adaptation to dynamic changes in the environment. The only way of doing this with DEVS is by having multiple models defined and a selector model to activate the right one at any time. Dynamic structure systems instead focus on the possibility of changing the system structure dynamically according to the system's real requirements. Dynamic structure DEVS allows addressing some of these issues and supports structural changes on three levels:

1. System level: the structural change happens between coupled models (i.e., a new link between two coupled models is added).
2. Component level: the structural change happens within a coupled model but including two or more atomic models (i.e., a new link is added between two atomic models).
3. Subcomponent level: the structural change only happens within a single atomic model (i.e., the external transition function changes).

There are two popular dynamic structure DEVS definitions, namely, dynamic structure discrete event (DSDE) [11–13] and dynDEVS [14,15]. DSDE divides the models into two groups: *basic* and *network* models. The basic models are atomic structure units (cannot be split); network models are coupled components composed of multiple basic interconnected structure models (which can include structural changes). A *network executive* is a modified basic model that is used to conduct the changes in network models by storing all possible states for structural changes and their corresponding component sets. The two parts are associated together through an index function in the network executive. A DSDE network is defined as

$$DSDEN_N = (X_N, Y_N, \chi, M_\chi) \tag{2.11}$$

where

X_N is the network input value set;
Y_N is the network output value set;
χ is the name of the network executive; and
M_χ is the model of the network executive χ, which is a modified basic model and is defined by

$$M_\chi = (X_\chi, s_{0,\chi}, S_\chi, Y_\chi, \gamma, \Sigma^*, \delta ext_\chi, \lambda_\chi, \delta int_\chi) \tag{2.12}$$

Here,

X_χ, S_χ, Y_χ, δext_χ, λ_χ, and δint_χ are defined as in DEVS;
γ: $S_\chi \rightarrow \Sigma^*$ is the structure function; and
Σ^* is the set of network structures.

If $s_\alpha \in S_\chi$ is a partial state of the network executive, then $\gamma(s_\alpha) = \Sigma_\alpha = (D, \{M_i\}, \{I_i\}, \{Z_i\})_\alpha$ is a network structure (equivalent to a coupled model), where D is the set of component names associated with the executive partial state s_α for all $i \in D$; M_i is the model of the component i for all $i \in D \cup \{\chi, N\}$; I_i is the set of influencees of i for all $i \in D \cup \{\chi\}$; and Z_i is the input function of the component i and Z_N is the network output function.

As we can see, the structure function provides a mapping between a partial state of the network and a new network structure, permitting us to carry out structural changes.

The dynDEVS formalism does not introduce an extra component to conduct dynamic structural changes. Instead, two kinds of dynamic DEVS models are included: dynDEVS (atomic) and dynNDEVS (coupled). The dynDEVS models atomic components are defined as

$$dynDEVS = df <X, Y, minit, M(minit)> \tag{2.13}$$

where

X, Y are the input/output sets;
$minit \in M(minit)$ is the initial model; and
$M(minit)$ is the least set having structure $\{ <S, sinit, \delta_{ext}, \delta_{int}, \rho\alpha, \lambda, ta> \}$.

A dynDEVS model can be interpreted as a set of DEVS models with the same interface plus a transition function that determines which DEVS model succeeds the previous one. It includes an initial state and a dynamic reconfiguration function ($\rho\alpha$), which will be in charge of structural changes. A model's state space, internal and external transition, output, time advance, and model transition functions are subject to change during simulation.

dynNDEVS models are coupled structural components defined as

$$dynNDEVS = df <X, Y, ninit, N(ninit)> \tag{2.14}$$

where

X, Y are the input/output sets;
$ninit \in N(ninit)$ is the start configuration; and
$N(ninit)$ is the least set having the structure $\{<D, \rho_N, \{dynDEVS_i\}, \{I_i\}, \{Z_{ij}\}, Select>\}$.

The dynNDEVS model is similar to a coupled model, but it now includes the dynamic configuration function ρ_N.

Both of the preceding formalisms introduce new structure transition functions to conduct structural changes. In DSDE, the structural changes are carried out by χ (the network executive) and the structure function γ (which maps the network structure state set S_χ and the network structure models' set Σ^*). The centralized network executives make sure that the structure transition is executed sequentially without any conflicts between structural change functions of the models. In dynDEVS, agents associated with the models conduct structural changes. ρ_α and ρ_N are structure transition functions in dynDEVS and dynNDEVS models, respectively, which execute structural changes concurrently and independently.

We will show how to apply these concepts to a model of an automated manufacturing system (AMS) consisting of a flow shop for manufacturing cars. The system consists of dedicated stations that perform tasks on products being assembled and conveyors that transport the products to or from those workstations. The structure of the model is presented in Figure 2.6. As we can see, the flow shop consists of five parts:

1. *Conveyor belts* are used to transport products between the different stations (a conveyor is composed of an engine and sensors).
2. The *control unit* is in charge of controlling the movement of the conveyors according to the production cycle provided by a scheduler.
3. The *scheduler* (SCH) is in charge of the production cycle organization, and it programs the control unit to execute the production cycle on both conveyors.
4. The *display controller* displays the current status of the whole AMS system (a SCADA system).

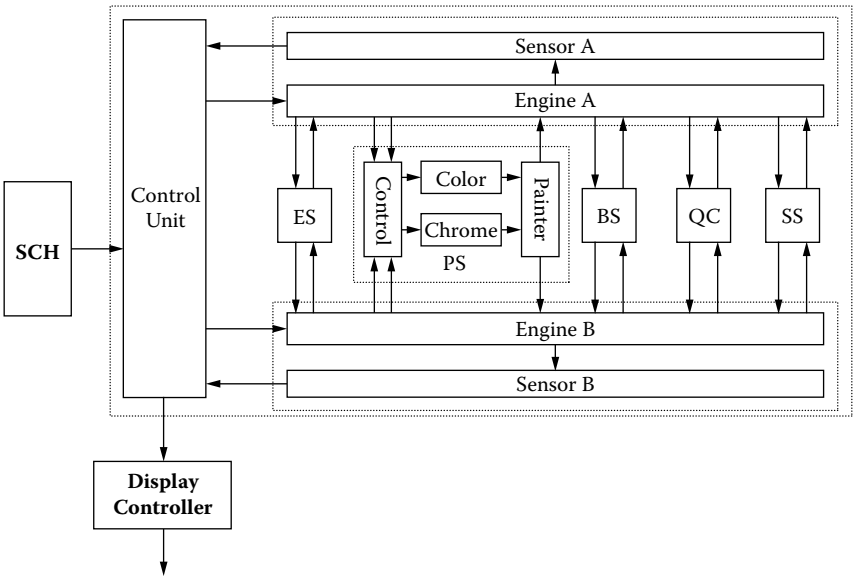


FIGURE 2.6 An automated manufacturing system model.

5. Each *workstation* is in charge of a different task and its quality control. Vehicles being manufactured are delivered to each workstation (in order to be served step by step) by the conveyors. The Engine Station (ES) is in charge of assembling engine parts, the Painting Station (PS) undertakes the painting and special painting tasks, the Baking Station (BS) is in charge of baking (which takes place after painting), QC serves as a Quality Center to evaluate the quality of the vehicles for the whole plant, and the Storage Station (SS) distributes the vehicles to their corresponding warehouses.

During system execution, the structure of the AMS could be affected in order to adapt to the changes of external environment. Two kinds of system adjustments are considered:

- Workstation duty shifts: workstations have different working capacity during day and night.
- Workload in the PS workstation: this station is in charge of color or chrome painting. Vehicles might need color painting or both color and chrome painting. Painting selection is determined by the "control" model residing in the PS workstation.

Figure 2.7 shows the case of dynamic reconfiguration in the ES workstation due to duty shifts. ES and ES′ represent the engine workstation during day and night, respectively, and they can be considered as two structural states of the basic model ς. $Z_{es,0}$ and $Z_{es,1}$ represent the input functions of ES and ES′; $Z_{\varsigma,0}$ and $Z_{\varsigma,1}$ represent the output functions of the structural model ς. χ is the network executive described in DSDE.

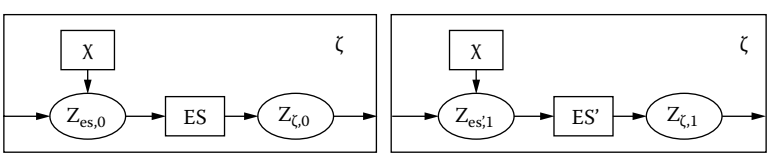


FIGURE 2.7 ES structure layout during daytime and nighttime.

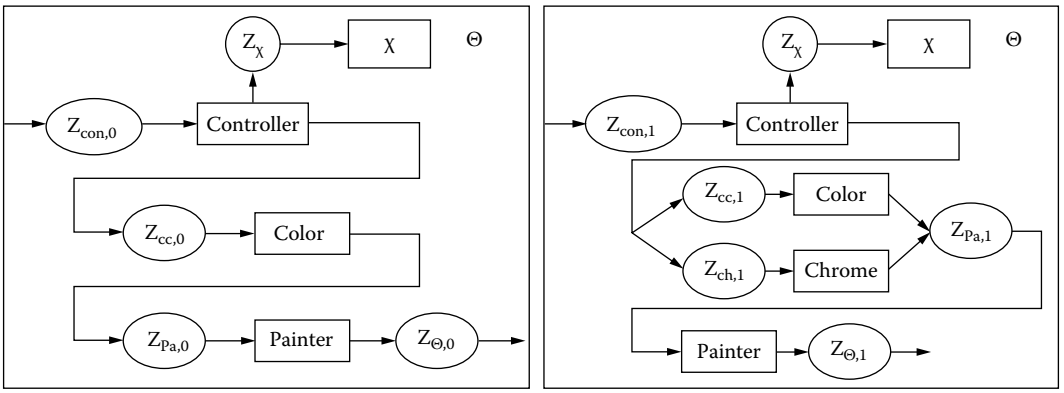


FIGURE 2.8 Painting mode I in PS workstation and painting mode II in PS workstation.

The PS workstation is a coupled model including four atomic models: controller, color, chrome, and painter. The atomic model "chrome" is an optional component. Painting selection is determined by the "Controller." Figure 2.8 shows the two structural states of the network model Θ. $Z_{i,\alpha}$ (i = controller, color, chrome, and painting; $\alpha = 0,1$) is the input function of the atomic models and $Z_{\Theta,\alpha}$ is the output function of the network model Θ. Z_{χ} is the input function of the network executive χ.

Generally, the autos on the conveyor are painted with specific colors. Therefore, only the atomic model "color" is needed in the PS. Assume that the current bulk of autos on the conveyor needs to be painted both color and chrome. The atomic model "chrome" should be added into PS automatically.

Exercise 2.9

Write the dynDEVS specification for the previous example, based on [14,15].

Exercise 2.10

Write a static DEVS specification for all the previous examples.

2.6 QUANTIZED DEVS

As mentioned in Chapter 1, the first existing modeling techniques focused on modeling the continuous behavior of the dynamic systems, using various kinds of differential equation formalisms. The evolution of state variables for dynamic systems is described via state equations modeled using differential equations. Ordinary differential equations (ODEs) are described as

$$\dot{x} = f(x,t) \tag{2.15}$$

with no algebraic constraints for the vector of differential variables. The simplest state-space models are represented by ODEs:

$$\dot{x} = f(x,u,t) \tag{2.16}$$

where $\boldsymbol{x}$ represents the state variables vector and $\boldsymbol{u}$ represents the inputs vector.

Differential algebraic equations (DAEs) are constructed as a set of differential equations with additional algebraic constraints in the form

$$f(\dot{x},x,u,t) = 0 \tag{2.17}$$

where

$x \in \boldsymbol{R}^n$ is a vector of *differential* variables;
$u \in \boldsymbol{R}^m$ is a vector of *algebraic* variables;
$t \in$ R is an *independent* variable; and
$f \in R^{2n+m+1} \rightarrow R^{n+m}$ is the set of DAEs.

As discussed in Chapter 1, continuous systems simulation is mainly solved by approximating the set of differential equations describing the system and finding consistent initial conditions. There is a wide variety of ODE solvers—for example, *forward Euler* (explicit method), *backward Euler* (implicit method), and *Runge–Kutta* [6]. For DAEs, if the equations can be transformed to a set of ODEs, a simulator can numerically approximate the equations using any ODE solver. If the transformation is not possible, a DAE solver can be used (e.g., *DASSL* and *implicit Runge–Kutta*). In DAEs, the simulator might have to differentiate the equations a very large number of times in order to get an ODE in all the state variables (because the ODE's index is equal to zero). Constraints (dependencies among variables that cannot be chosen freely) are usually hidden in these high-index DAEs. Several algorithms for index reduction and finding hidden constraints can be found in the literature, including *Gear and Petzold*, *Bachmann*, and *Pantelides* algorithms for index reduction, etc. [16–18].

Most of the techniques just mentioned have traditionally been simulated by discretizing the time domain and solving the equations over each discrete time interval. However, a few years ago a new approach for continuous systems simulation based on the discrete event paradigm was introduced. Discrete event methods in general and DEVS in particular present several advantages in contrast to the classical discrete time techniques:

- Computational times reduction: for a given accuracy the number of calculations can be decreased.
- Complex model definition in a hierarchical modular fashion: DEVS allows specification of complex systems in a hierarchical way.
- Hybrid systems modeling and simulation: DEVS provides a theory to develop a uniform approach to model and simulate systems with continuous and discrete components.

These techniques are based on a theory of quantized DEVS (QDEVS) [19]. The basic idea is shown in Figure 2.9(a). We discretize the space of the state variables using a fixed value called the *quantum* size. Thus, a state change will be informed only if it crosses the threshold defined by the quantum. As we can see, a continuous curve is now represented by the crossings of an equally spaced set of boundaries, separated by the quantum size, converting the continuous signal into a discrete-event version (in which the signal is piecewise constant). This operation reduces substantially the frequency of message updates while potentially incurring error (like any other numerical method).

The *QSS* (*quantized state systems*) formalism developed by Kofman [20] allows continuous systems simulation based on a combination of QDEVS and hysteresis. This approach constitutes the

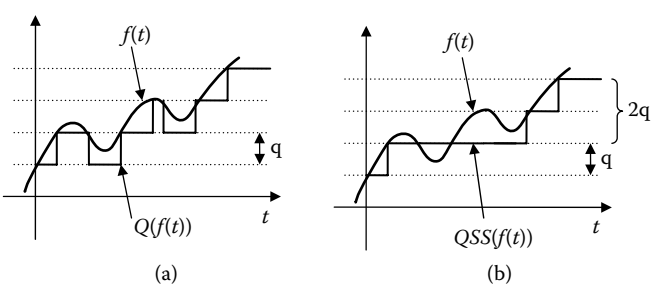


FIGURE 2.9 (a) Signal quantization; (b) quantization function with hysteresis.

first general method for ODE integration using discrete event theory, and it has been proved that ODE systems can be approximated by a legitimate DEVS model. The main difference with QDEVS is that the quantization function is combined with hysteresis (e.g., the quantum size is changed to its double when there are direction changes on the values, as seen Figure 2.9(b)). This means that if a value changes its direction with respect to the last threshold value, the next value will have to change two regions to be transmitted. This eliminates the problem of possibly infinite numbers of transitions performed by a model in a finite interval (a legitimacy problem in DEVS models).

In Kofman [20] it was proven that the original system and the resulting QSS have similar trajectories. Some properties of the original system, like equilibrium points and stability, are preserved on the simulation model. It was also shown that the solution of the simulation model converges to the solution of the original system when the discretization goes to zero, allowing the method to be implemented with an arbitrary small error [6].

Exercise 2.11

Let $y = 3e^x + 1$. Define a simulation algorithm that will "plot" this function starting with $x_0 = -10$ and will use a time step of $h = 0.5$. Run a desk test of the simulation, plotting the results (use any programming language, spreadsheet, or pen-and-pencil solution). Repeat the exercise with $h = 1$.

Exercise 2.12

Invert the function in Exercise 2.11 so that now we can obtain x as a function of y. Define a simulation algorithm that will "plot" this function starting with $y_0 = 2.00013$ and a quantum size of $q = 1$. Run a desk test of the simulator, plotting the results (use any programming language, spreadsheet, or pen-and-pencil solution). Repeat the exercise with $q = 20$.

Exercise 2.13

Write an algorithm for a function that, given the last two values computed, will determine if there was a difference of more than five units between the two values.

Exercise 2.14

Combine the function in Exercise 2.13 with the algorithm introduced in Exercise 2.11 in order to find differences larger than five units.

Exercise 2.15

Define the models of Exercises 2.12 and 2.13 as DEVS atomic models. Combine them as coupled models.

As we can see from these exercises, quantization requires a fundamental shift in thinking about the system as a whole. Instead of determining what value a dependent variable will have (its state) at a given time, we must determine at what time a dependent variable will enter a given state—namely, the state above or below its current state.

2.7 GENERALIZED DEVS (GDEVS)

Another approach recently applied to deal with continuous systems modeling based on discrete-event specifications is the *GDEVS* (*Generalized Discrete Event Specification*) formalism [21]. GDEVS uses polynomials of arbitrary degree to represent the piecewise input–output trajectories of a discrete event model.

GDEVS uses a new definition for the concept of event. The target real-world system is modeled through piecewise *polynomial segments* translated into piecewise constant trajectories. A *coefficient*

event is thus considered as an instantaneous change of at least one of the values of the coefficients defining the piecewise polynomial trajectory of the variable under study. An event is a list of coefficient values defining the polynomial that describes the trajectory of the variable.

A piecewise continuous polynomial segment is one that is defined over a continuous time base $\omega < t_0, t_n > \rightarrow \mathbf{A}$, as follows [21]:

- There is a finite number of elements $\{t_1, \ldots, t_{n-1}\}$, $\forall i \in [1, n-1]$, and t_i is associated with a constant valued n-tuple $(a0_i, \ldots, an_i)$. $\forall\ t \in <t_k, t_l>$, where $t_k, t_l \in \{t_1, \ldots, t_{n-1}\}$; we define $\omega(t) = a0_i + a1_i\, t + \ldots + an_i\, t^n$.
- $\omega_{<t0,tn>} = \omega_{<t0,t1>}/\omega_{<t1,\, t2>}/\ldots/\omega_{<t-1,\, tn>}$, where / represents the left concatenation operator over segments.

For an individual segment $\omega_{<t_i, t_j>}$ of order n, its coefficient value is defined by $(a0, \ldots, an)$, where $a0$ is the value of the segment at time t_l (named the "intercept"), and every ai is the i-gradient.

Figure 2.10 shows the continuous function $f(x) = \sin(2\pi x) + \cos(e^x)$, a polynomial approximation of order 1 (i.e., the function is approximated by $a1x + a2$), and by a events of order one ($a1$) (i.e., by a piecewise constant function, as in DEVS).

For a given piecewise polynomial segment, a coefficient event is defined by an instantaneous change in at least one of the values of the coefficients of the polynomial. For the piecewise polynomial segment $w_{<t0,\, tn>}$, there exists an event at time t_i if the values of the coefficients $(a0_k, \ldots, an_k)$ over $<t_k, t_i$ and those of the coefficients $(a0_i \ldots, an_i)$ over $t_i, t_j>$ satisfy the condition that there exists an l such that $al_k \neq al_i$.

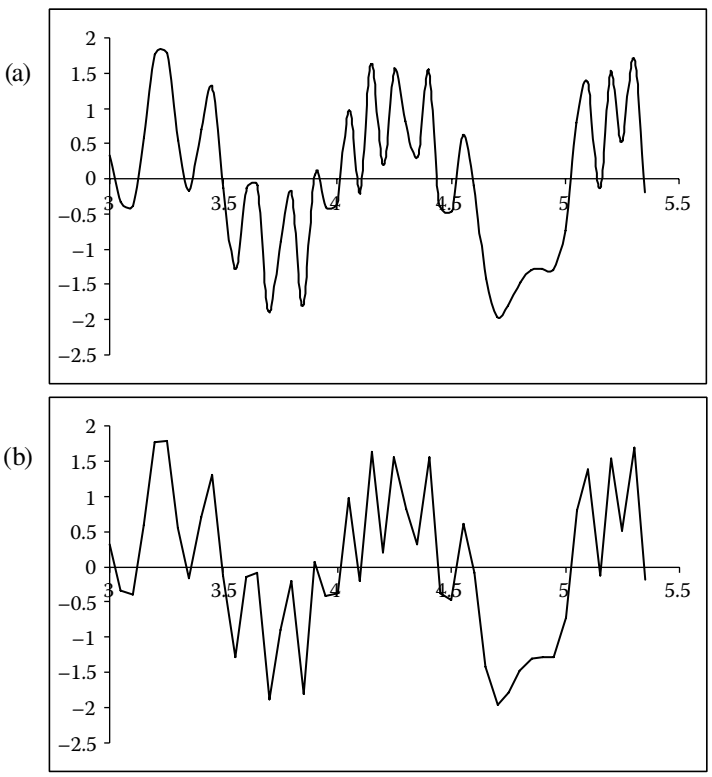


FIGURE 2.10 GDEVS approximation of a continuous signal: (a) continuous segment; (b) linear segment; (c) piecewise segment.

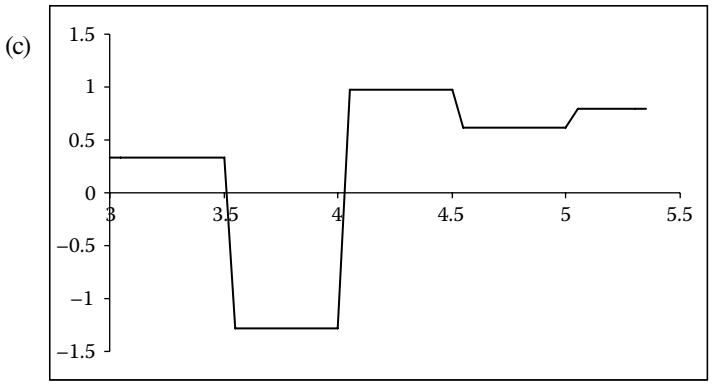


FIGURE 2.10 (continued)

This approach is *solution based* and requires knowing the continuous system response to particular input trajectories and this represents a disadvantage, considering that this information might be available [20].

Exercise 2.16

(a) Write a GDEVS model for the function in Figure 2.10. (b) Write a QDEVS approximation for the same function. (c) Write a QSS approximation for the same function. (d) Simulate the execution of the three previous models.

Exercise 2.17

Repeat Exercises 2.11–2.15 for the function in Figure 2.10.

2.8 SUMMARY

In this chapter, we have introduced the DEVS formalism and different variations. DEVS formal definitions are useful to improve the security and to reduce the development costs of a simulation; a formal conceptual model can be validated, improving the error detection process and reducing testing time. DEVS models are closed under coupling; therefore, a coupled model is equivalent to an atomic one, allowing reuse of previously defined models. Each model can be associated with an experimental framework, allowing the individual testing of components and making integration testing easier. Likewise, the simulation engines are independent from the modeling framework, which allows having a layered view of modeling and simulation (Figure 2.11).

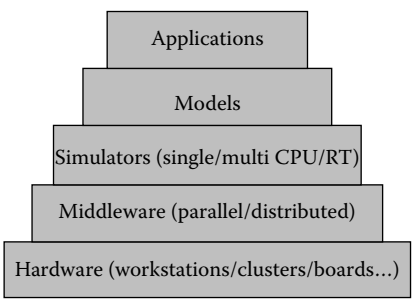


FIGURE 2.11 A layered view of DEVS M&S.

DEVS, as a discrete event paradigm, uses a continuous time base, which allows accurate timing representation. The hierarchical and modular organization allows describing of multiple layers of a given application. This organization makes the definition of submodels easier, which in turn makes the definition of different levels of abstraction easy. The existence of an internal transition function eases the definition of certain properties. Internal state changes can be captured, describing complex internal interactions in a simple and natural way.

Recently, a theory of DEVS quantized models was developed, and it has been verified when applied to predictive quantization of arbitrary ordinary differential equation models. Quantized models reduce substantially the frequency of message updates. As the information interchange is reduced, the models potentially incur error. In this way, DEVS can be used to express hybrid digital/analog systems. GDEVS also enables the definition of hybrid models, which are expressed in a combined discrete event/differential equation formalism approximated by DEVS. In GDEVS, the accuracy of an analog subsystem is preserved using piecewise polynomial segments. The error introduced in this approximation can be controlled by increasing the order of the polynomials that represent analog signals between successive digital events.

REFERENCES

1. Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.
2. Klir, G. J. 1972. *Trends in general systems theory.* New York: Wiley-Interscience.
3. Zadeh, L. A., and C. A. Desoer. 1963. *Linear system theory: The state space approach.* New York: McGraw–Hill.
4. Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation,* 2nd. ed. New York: Academic Press.
5. Zeigler, B. 1984. *Multifaceted modeling and discrete event simulation.* New York: Academic Press.
6. Cellier, F. E., and E. Kofman. 2006. *Continuous system simulation.* New York: Springer Science+ Business Media.
7. Nutaro, J. 2003. Parallel discrete event simulation with application to continuous systems. PhD thesis, University of Arizona, Tucson.
8. Nutaro, J., and H. Sarjoughian. 2004. Design of distributed simulation environments: A unified system-theoretic and logical processes approach. *Simulation* 80:577–589.
9. Kim, T. G., S. M. Cho, and W. B. Lee. 2000. DEVS framework for systems development: Unified specification for logical analysis, performance evaluation and implementation. In *Discrete event modeling & simulation: Enabling future technologies,* ed. H. S. Sarjoughian and F. Cellier. New York: Springer–Verlag.
10. Chow, A. C., and B. Zeigler. 1994. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. *Proceedings of Winter Simulation Conference,* Orlando, FL.
11. Barros, F. J. 1997. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation* 7:501–515.
12. Barros, F. 1998. Abstract simulators for the DSDE formalism. *Proceedings of Winter Simulation Conference,* Washington, D.C., 407–412.
13. Barros, F. J. 1995. Dynamic structure discrete event system specifications: A new formalism for dynamic structure modeling and simulation. *Proceedings of Winter Simulation Conference,* Arlington, VA, 781–785.
14. Uhrmacher, A. M. 2001. Dynamic structure in modeling and simulation: A reflective approach. *ACM Transactions on Modeling and Computer Simulation* 11:206–232.
15. Uhrmacher, A. M., and J. Himmeelspach. 2004. Processing dynamic PDEVS models. *Proceedings of 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04),* Volenlam, the Netherlands.
16. Pantelides, C. C. 1988. The consistent initialization of differential-algebraic systems. *SIAM Journal of Scientific and Statistical Computing* 9:213–231.
17. Fábián, G. D., D. A. van Beek, and J. E. Rooda. 2000. Substitute equations for index reduction and discontinuity handling. *Proceedings of Third IMACS Symposium on Mathematical Modelling,* Vienna, Austria.
18. Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1986. *Numerical recipes.* Cambridge: Cambridge University Press.

19. Zeigler, B. P., DEVS theory of quantization. 1998. Technical report, DARPA contract N6133997K-0007, ECE Dept., University of Arizona, Tucson.
20. Kofman, E. 2003. Quantized-state control. A method for discrete event control of continuous systems. *Latin American Applied Research Journal* 33:339–406.
21. Giambiasi, N., B. Escude, and S. Ghosh. 2000. GDEVS: A generalized discrete event specification for accurate modeling of dynamic systems. *Transactions of the SCS* 17:120–134.

3 The Cell-DEVS Formalism

3.1 INTRODUCTION

Different formalisms have been used to capture the behavior of systems that can be represented as cell spaces (e.g., spatial models in which the space under study is organized as a grid of cells geometrically distributed). Examples of such systems can be found in many fields, from chemistry to engineering and from physics to social sciences [1,2]. "Cellular automata" (CA) is a well-known formalism that describes these types of systems [3–5]. A CA is an infinite regular n-dimensional lattice in which each of the cells can take a finite value. States in the lattice are updated according to a local rule in a simultaneous, synchronous way, and cell states change in discrete time steps (i.e., they are discrete-time, discrete-variable models). The automaton evolves by triggering a local transition function on each cell, which uses the current state of the cell and a finite set of nearby cells (called the neighborhood of the cell). Figure 3.1 shows an example of such a model. In this example the cell space is organized as a two-dimensional grid in which the cells contain a value or are empty, and each cell computes its future state based on this value and the values of the neighborhood (in this example, the 3×3 adjacent cells).

Cellular automata were originally introduced by von Neumann [6] to study self-reproducing systems [3], and they permit us to find emergent behavior of the systems through the definition of simple rules at the micro level [3]. Different parameters define the behavior of the CA: the alphabet chosen to represent the phenomenon, the individual behavior of local computing functions, and the shape of the neighborhood. This set can be *uniform* (i.e., all of the cells in the space use the same local neighbors to evaluate the next state) or *nonuniform* (in which each cell can potentially use different neighborhoods).

Neighbor cells can be in the local immediacy or they can include remote cells. Figure 3.2 shows some of the most widely used neighborhoods. Moore's neighborhood includes the origin and its eight adjacent cells; von Neumann's uses the ones to the up, down, left, right of center (the extended von Neumann uses a rhombus of 5×5 on the center cell). Hexagonal neighborhoods are very popular because they provide higher *isotropy* (i.e., the capacity to represent equivalent behavior in every possible direction), which results in more natural model rules. Triangular meshes can cover areas of more varied topology while having a reduced number of neighbors to compute. Nevertheless, square topologies are one of the most popular due to the ease of mapping and visualization implementation.

A special kind of neighborhood that is useful for varied applications was defined by Margolus [5]. It is useful for reversible models (i.e., those models in which we can go forward or backward in time). The idea is to use a partitioned cell space in which the grid is divided in a finite, disjoint, and uniform set of blocks. While executing, the cellular model applies the rules to a block of cells (using the values included in those particular cells). The blocks are not overlapping, so there is no information interchange between adjacent blocks. Then the grid's partition is changed to a different set.

Figure 3.3 shows the basic idea behind Margolus's neighborhood. In this case, the 2×2 blocks of the even/odd grids are represented with thick or fine lines. As we can see, the grid changes in each step. First, the cell marked in black will use the neighbors within the even grid; in the following step, it will use the neighborhood in the odd grid.

The following sections will give an introduction to formal specifications for CA and an introduction to the Cell-DEVS formalism, which allows us to define cell spaces based on DEVS and CA models.

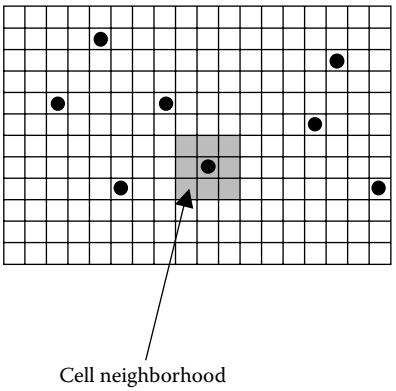


FIGURE 3.1 Sketch of a two-dimensional cellular automaton.

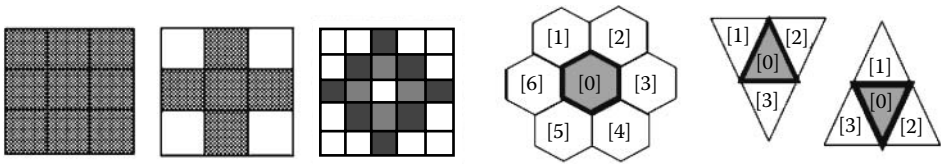


FIGURE 3.2 Widely used neighborhoods: (a) Moore; (b) von Neumann; (c) extended von Neumann; (d) hexagonal topology; (e) triangular topology.

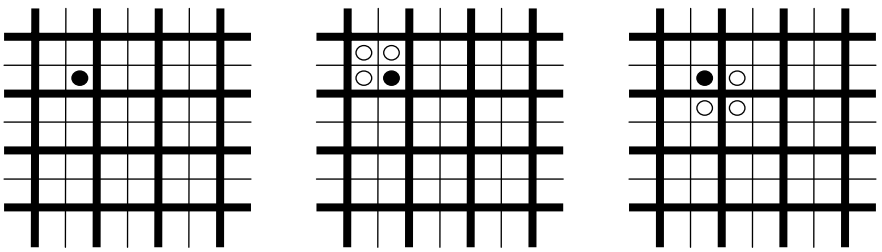


FIGURE 3.3 2 × 2 Margolus neighborhood.

3.2 CELLULAR AUTOMATA

In this section, we introduce a formal definition of CA considering synchronous and asynchronous approaches presented in [7]. From now on, we will use the following notation for our specifications:

- $C_c \in S$ defines the status for the cell c; and
- $c \in \mathbf{Z}^n$, $c = (i_1,\ldots,i_n)$ is the cell's position within an n-dimensional cell space. Here, $\forall\ k \in [1,n]$, $i_k \in \mathbf{Z}$ is the position of the cell in the kth dimension (if we consider conceptual CA, $\forall\ k \in [1,n]$, $i_k \in [-\infty,\infty]$).

Using this notation, a CA can be defined as

$$\text{CA} = < S, n, C, N, T, \tau, q.\mathbf{Z}_0^+ > \tag{3.1}$$

where

$\boldsymbol{S} \subseteq \mathbf{Z} \wedge \#S < \infty$ is the alphabet used to represent the state for each cell;

$\boldsymbol{n} \in \boldsymbol{N}$ is the dimension for the cell space;

$\boldsymbol{C} = \{\ C_c / c \in \boldsymbol{Z}^n \wedge C_c \in S\ \}$ is the state set for the cell space;

$\boldsymbol{N}$ is the neighborhood set; if the neighborhood is homogeneous, $\boldsymbol{N} = \{\ (v_{k,1},\ldots,v_{k,n}) / \forall\ (k \in \boldsymbol{N}, k \in [1,\eta]) \wedge (i \in \boldsymbol{N}, i \in [1,n]), v_{k,i} \in \boldsymbol{Z}\ \}$ (i.e., the neighborhood is an n-dimensional list of elements of the size of the neighborhood). The η value represents the neighborhood's size, and in this case, $\eta \in \boldsymbol{N} \wedge \eta = \#N$. N is usually defined as a set of adjacent cells; that is, each $v_{k,i} \in [-1,1]$. However, for nonhomogeneous neighborhoods, $\boldsymbol{N} = \{N_c / c \in \boldsymbol{Z}^n\}$, with $N_c = \{\ (v_{k,1},\ldots,v_{k,n})_c / \forall\ (k \in \boldsymbol{N}, k \in [1,\eta_c]) \wedge (i \in \boldsymbol{N}, i \in [1,n]), v_{k,i} \in \boldsymbol{Z}\ \}$. Here, $\eta_c \in \boldsymbol{N} \wedge \eta_c = \#N_c$ (i.e., a different neighborhood shape is defined in every cell);

$\boldsymbol{T}$: $C \times q.\boldsymbol{Z}_0^+ \rightarrow C$ is the global transition function;

$\boldsymbol{\tau}$: $C_c \times \mathrm{N} \times q.\boldsymbol{Z}_0^+ \rightarrow C_c$ is the local computation function. If the neighborhood is homogeneous, $C_c[t+q] = \tau(C_{c+v1}[t],\ldots,C_{c+v\eta}[t])$, where $t \in q.\boldsymbol{Z}_0^+$; $\forall\ k \in [1, \eta]$, $vk \in \mathrm{N} \wedge c + vk = (i_1 + v_{k,1},\ldots, i_n + v_{k,n})$. For nonhomogeneous CA:

$\boldsymbol{\tau}$: $C_c \times N_c \times q.\boldsymbol{Z}_0^+ \rightarrow C_c$. Here, $C_c[t + c] = \tau(C_{c+v1}[t],\ldots,C_{c+v\eta c}[t])$, where $t \in q.\boldsymbol{Z}_0^+ \wedge k \in [1, \eta_c]$, $vk \in N_c \wedge c + vk = (i_1 + v_{k,1},\ldots,i_n + v_{k,n})$; and

$q.\boldsymbol{Z}_0^+ = \{\ i / i \in \boldsymbol{N}, i = q.j \wedge j \in N\ \} = \{\ 0, q, 2q, 3q,\ldots\}$ is the (discrete) time base for the CA.

It can be seen that a model is built as an $\boldsymbol{n}$-dimensional cell space ($\boldsymbol{C}$). This state space progresses in discrete time steps: the time base is defined by $q.\boldsymbol{Z}_0^+$ (a set of integer values separated by a time constant). The state for each cell in the space can take a value from a finite alphabet ($\boldsymbol{S}$).

The cell's neighborhood is defined as a list of η n-dimensional neighbors. In the homogeneous case, the neighbors are defined as an n-tuple of positions relative to the origin cell. This definition uses an index (k) that allows us to identify the neighbor number, and a second index (i) indicating the dimension for each neighbor's position. The nonhomogeneous neighborhoods are defined with an array of neighborhood lists. In this case, each cell will have a neighbor's list composed by η_c elements, which are constituted by tuples of indexes relative to the origin cell.

The state space of the automata evolves by executing a global transition function ($\boldsymbol{T}$) that changes the state of the cell space. The behavior of this function responds to the execution results of local transition functions (τ) that execute locally in the neighborhood for the cell ($\boldsymbol{N}$). Conceptually, the computation for these local functions is done synchronously and in parallel for every cell in the space. The semantics of this behavior can be defined by the following rule:

$$\frac{\mathrm{C}_c \in \mathrm{C}\ \forall\ c \in \boldsymbol{Z}^n, \quad t \in q.\boldsymbol{Z}_0^+}{\mathrm{C}[t+q] = \mathrm{T}(\mathrm{C}[t]), \quad \text{with } \mathrm{C}_c[t+q] = \tau(\mathrm{N}_c, \mathrm{C}_c[t])\ \forall\ c \in \boldsymbol{Z}^{\mathrm{n}}; \quad t = t+q} \tag{3.2}$$

This definition considers that as a precondition (rules above the line) the global transition function analyzes all the cell space at the instant t, and then it produces a change in the cell space for the next step (post-condition defined after the line). The period for this step is of q time units. This change can be seen as the individual computation of the local transition function for each cell in the space.

The previous case considered that the index for the cell space can include an infinite number of cells. Because the interest is focused on models that can run in a computer, an executable synchronous cellular automaton can be defined as

$$\mathrm{ECA} = < S, n, \{t_1,\ldots,t_n\}, C, N, B, T, \tau, q.\boldsymbol{Z}_0^+ > \tag{3.3}$$

where all the elements represent the sets in CA, and the following were added:

- $\{t_1,\ldots,t_n\}$ is the number of cells on each of the dimensions.
- **B** is the set of border cells **B** = {∅} if the cell space is "wrapped" (that is, the cells in each border are connected with the cells in the opposite one), or **B** = $\{C_b/C_b \in C\}$. In this case, B has the restriction that $\tau_b \neq \tau_c = \tau \; \forall \; (C_c \notin B \wedge C_b \in B)$.
- τ: $C_c \times N_c \times q.\mathbf{Z}_0^+ \rightarrow C_c$, where $C_c[t + c] = \tau(C_{c/v1}[t],\ldots,C_{c/v\eta}[t])$, with $t \in q.\mathbf{Z}_0^+ \wedge \forall\, k \in [1,\eta_c]$, $vk \in N_c \wedge c/vk = (i_1 + v_{k1} \bmod(\mathbf{t}_1),\ldots, i_n + v_{kn} \bmod(\mathbf{t}_n))$ in the case that B = {∅}, and $C_b[t + c] = \tau_b(C_b[t])$, $\forall$ b ∈ B, with $t \in q.\mathbf{Z}_0^+$. In this case, $\tau \neq \tau_c = \tau \; \forall \; C_c \in B$, and for these, C_c is computed as in (3.2). If the cell space is homogeneous, then $\eta_c = \eta \wedge N_c = N$.

This definition for executable cellular automata differs in certain aspects from that of conceptual CA. The first difference is that the cell space is bounded in each of the dimensions ($\mathbf{t}_1,\ldots,\mathbf{t}_n$). The number of dimensions is also finite, and the cell's indexes are bounded to finite natural numbers. Another constraint is due to the loss of homogeneity in the cell space. This is due to the existence of a finite number of cells. Therefore, it is necessary to include a set of border cells (**B**) with behavior different from that of the others in the cell space. All the cells in the border have different behavior from those in the rest of the automaton.

The use of discrete time poses constraints in the precision and execution performance of these complex models. To achieve the desired accuracy, smaller time slots must be used, thus producing higher needs of processing time. To avoid these problems, asynchronous solutions can be used. Asynchronous CA can be defined as

$$\text{ACA} = < S, n, \{t_1,\ldots,t_n\}, C, N, B, Cn, T, \tau, \boldsymbol{R}_0^+ > \tag{3.3}$$

where all the sets are defined as in the previous cases, except for the time base (that in this case is continuous) and a sorting of cells according to their imminent times (**Cn**). These sets are defined by

- **Cn** = { (c, t)/c ∈ $\boldsymbol{C} \wedge t \in \boldsymbol{R}_0^+$}, where c is the position of a cell in the space, and t is the time of the next event.
- τ: $C_c \times N_c \times \boldsymbol{R}_0^+ \rightarrow C_c$, where $C_c[t_p] = \tau(C_{c/v1}[t],\ldots,C_{c/v\eta c}[t])$, with $t \in \boldsymbol{R}_0^+ \wedge \forall$ k ∈ [1, η], $vk \in N_c \wedge c/vk = (i_1 + v_{k1} \bmod(\mathbf{t}_1),\ldots,i_n + v_{kn} \bmod(\mathbf{t_n}))$ when B ={∅}, and $C_b[t_p] = \tau_b(C_b[t])$, ∀ b ∀ B, with $t \in \boldsymbol{R}_0^+$. In this case, $\tau \neq (\tau_c = \tau) \; \forall \; C_c \in B$. Here, $t_p = \min\{t_i\}_{i=1}^n$, with i, $p \in N/t_i \in \boldsymbol{R}_0^+$, and $b = c_p \vee c = c_p$, with $(t_i, c_i) \in Cn$. If the cell space is homogeneous, then $\eta_c = \eta \wedge N_c = N$.

It can be seen that most of the sets and functions defined are similar to that for the synchronous case. The changes are due to the existence of a continuous time base (that is, the time variables $t \in \boldsymbol{R}_0^+$). To allow the asynchronous definition, the imminent cells list (**Cn**) is included to keep the information related with the next events expected on each of the cells. In this case, the semantics of the global transition function is different from that for the previous case. Here, this function means to execute only a group of nonquiescent cells called the *imminent*. The execution of this function is performed simultaneously in all the imminent cells for a given time.

3.3 CELL-DEVS ATOMIC MODELS

In Wainer and Giambiasi [7] and Wainer [8], the Cell-DEVS formalism was presented. Cell-DEVS a combination of DEVS and CA with explicit timing delays. In Cell-DEVS, each cell is defined as an atomic model, and a procedure to couple cells is defined.

Figure 3.4 depicts informally the basic contents of atomic cells. Upon the occurrence of an external event, the local computing function τ is executed, consuming the inputs in **N.** In order to improve computing time, we activate the influenced cells only when the influencing cell changes, as discussed in Zeigler [9]. Therefore, the result of the local computing function will be transmitted

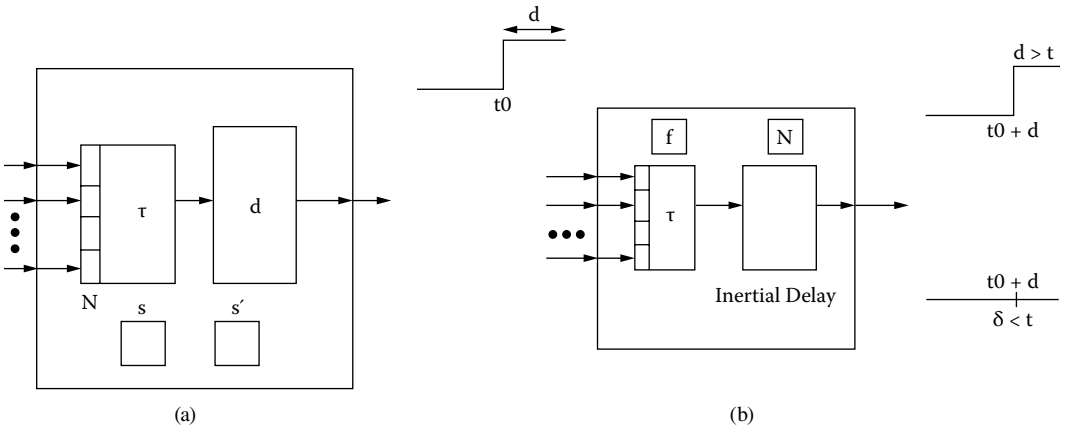


FIGURE 3.4 Informal description of an atomic cell: (a) transport delays; (b) inertial delays.

only when the state changes (i.e., if $s \neq s'$). In this case, the state change is transmitted after a delay of **d** time units. This model can be formally described as

$$\text{TDC} = < X, Y, S, N, type, d, \tau, \delta_{int}, \delta_{ext}, \lambda, D > \tag{3.4}$$

where

X is the set of input external events;
Y is the set of output external events;
S is the state set;
$\boldsymbol{N} \in \boldsymbol{X}^{\eta}$ is the set of input values;
$\boldsymbol{d} \in \boldsymbol{R}_0^+$ is the delay for the cell;
type is the kind of delay (transport/inertial/other);
$\boldsymbol{\tau}$: $N \rightarrow S$ is the local computing function;
$\boldsymbol{\delta_{int}}$: $S \rightarrow S$ is the internal transition function;
$\boldsymbol{\delta_{ext}}$: $Q \times X \rightarrow S$ is the external transition function;
$\boldsymbol{\lambda}$: $S \rightarrow Y$ is the output function; and
$\boldsymbol{ta}$: $S \rightarrow \boldsymbol{R}_0^+ \cup \infty$ is the state's lifetime function.

The **N** set represents the input values received (in general, from the neighbor's cells, although it can receive values from external DEVS or Cell-DEVS models as well). It is represented as a *k*-tuple $(n_1,\ldots,n_k)$, which is used to activate the function τ and compute the next state when a new event is received. If the cell state changes, this result is transmitted after a given delay. The lifetime function $\boldsymbol{ta}(\boldsymbol{s})$ is used to keep track of the elapsed time for a cell state. Finally, $\boldsymbol{\delta_{int}}$, $\boldsymbol{\delta_{ext}}$, and $\boldsymbol{\lambda}$ are used to define the cell's basic behavior as follows.

- A cell will be active while external events are received or internal events are scheduled.
- A cell passivates when there are no further scheduled events to be transmitted.
- When an event arrives (e.g., because a neighbor has changed), the external transition function δ_{ext} is executed, and the function τ is activated.
- If the cell's state does not change, the cell passivates and it remains in a quiescent state. If there is a change, the external transition function schedules an internal transition after a **delay**.

Delays are implemented in a different way according to the kind of delay needed. For transport delays:

- State changes must be informed in the future; therefore, their values and scheduled times are stored in a local queue.
- If the cell is in passive state, it must be activated.
- If the cell is active, the event times stored in the future events queue must be updated to reflect the elapsed time **e**.
- When the delay expires, the value is transmitted by the output function, and the internal transition function removes the first member of the queue.

Let us consider a binary cell defined by using a transport delay of 17 time units. In this case, we have:

$X = Y = \{0,1\}$;
$d = 17$ (transport delay); and
and $\tau(N) = s$.

The cell's behavior for these trajectories is analyzed in Table 3.1. This table shows each transition, its activation time, and the cell's state values. The "a" superscript in the table identifies the execution of internal transition functions, while the remaining lines represent the execution of external transitions.

The present and future states are included in the columns tagged ***s*** and ***s′***, respectively. The fields containing two values separated by a slash represent the variable values before and after execution. The output values will be the state of the ***s*** variable in the lines corresponding to internal transition functions.

Initially, the cell is in passive state. At time 30, an input is received with value 1. When we compute $\tau(1) = 1$, the state has changed, so we reflect the change (*s* changes from 0 to 1), and we schedule an output in 17 time units (thus, we enqueue the value 1 to be transmitted in 17 time units, according to the transport delay). At time 40, we receive another input with value 0, and the cycle is repeated. This time we need to update the elements in the queue to reflect that 10 time units have been elapsed (thus, the next internal event is scheduled 7 time units from now). At this point, we schedule an internal transition (marked with "a"). The output function takes the first element in the queue and transmits it, and the internal transition function deletes the first element in the queue and updates the time on the rest (in order to reflect the elapsed time of 7 time units).

In the case of inertial delays (which represents a delay function with preemptive semantic), the behavior is different: an input must be discarded if its value is not kept for a certain period. If the input flow is steady during that time (called the inertial delay for the cell), the state must change. The

TABLE 3.1
Execution Sequence of a Transport Delay Cellular Model

t	*s*	*s′*	Phase	*ta*(*s*)	*e*	σ Queue
…	0	0	Passive			
30	0/1	1	Active	17	0	(1,17)
40	1/0	0	Active	7	10	(1,7),(0,17)
47[a]	0	0	Active	0/10	17/0	(0,10)
55	0/1	1	Active	2	8	(0,2), (1,17)
57[a]	1	1	Active	0/15	10/0	(1,15)
60	1/0	1/0	Active	12	3	(1,12), (0,17)
72[a]	0	0	Active	0/5	15	(0,5)
77[a]	0	0	Passive	∞	5	

[a] Execution of internal transition functions.

TABLE 3.2
Execution Sequence with Inertial Delays

t	*s*	*s′*	Phase	*ta*(s)	e	*x*	*f*
…	0	0	Passive	∞			
5	0/1	1	Active	5	0	1	1
10[a]	1	1	Passive	0/∞	5		
15	1/0	0	Active	5	0	0	0
19[b]	0/1	1	Active	1/5	4/0	1	0/1
24[a]	1	1	Passive	0/∞			
39	1/0	0	Active	5	0	0	1/0
44[a]	0	0	Passive	0/∞			
45	0	0	Active	5	0	1	0/1
50[a]	1	1	Passive	0/∞			

[a] Execution of internal transition functions.
[b] Behavior of the model under preemptions.

main change for cells with inertial delays is a different semantic for the delay: if the input value for the cell is kept during the inertial delay, the future state will be s′; otherwise, it is preempted.

The behavior for atomic cells with inertial delays can be studied in the following example. The input and output trajectories presented use an inertial delay of 5 time units. In Table 3.2, we can see the execution flow of the transition functions. The last arrived future event can be preempted if there is a new input before the consumption of the inertial delay. This happens only if the new external value is different from the one previously stored. If both values are the same, the new external event that has occurred has the same value as the previous one.

In this case, the cell is initially passive. At time 5, it receives an input producing a state change. Therefore, the feasible future *f* is 1. If this input value is maintained for the next five units, this will be the value of the cell. This time is consumed, and at time 10 an internal transition is executed. The output function transmits the cell's value (1), and it passivates, waiting for the next external event. This will happen at time 15, when the cell schedules an output in five time units (e.g., it should generate an output of 1 at 20). Nevertheless, at time 19, we receive another input, and the local computing function makes the cell to change to 0. Therefore, we preempt the previous state change (which will not be transmitted).

3.4 CELL-DEVS COUPLED MODELS

Once we define the behavior of a single cell, we need to form a cell space. Because most of the examples in the rest of the book will use two-dimensional models, we include the definition of two-dimensional Cell-DEVS coupled models with adjacent neighbors. Further information about formal definitions for *n*-dimensional models with generic neighborhoods can be found in Wainer and Giambiasi [7] and Wainer [8]. A Cell-DEVS coupled model can be represented as

$$\text{GCTD} = < X, Y, Xlist, Ylist, \eta, N, \{m, n\}, C, B, Z, select > \tag{3.5}$$

where

X is the set of input external events;
Y is the set of output external events;
Ylist = { $(k,l)/k \in [0,m], l \in [0,n]$} is the list of output coupling;
Xlist = { $(k,l)/k \in [0,m], l \in [0,n]$} is the list of input coupling;
select = { $(k,l)/(k,l) \in N$ } is the tie-breaking selector function;
$\boldsymbol{\eta} \in \boldsymbol{N}$ is the neighborhood size;

$\boldsymbol{N}$ is the neighborhood set, defined as
$N = \{ (i_p,j_p) / \forall\ p \in \boldsymbol{N}, p \in [1,\eta] \Rightarrow i_p, j_p \in \boldsymbol{Z} \wedge i_p, j_p \in [-1, 1] \}$;
$\boldsymbol{\{m, n\}} \in N$ is the size of the cell space;
$\boldsymbol{C}$ is the cell space set, defined as $C = \{C_{ij} / i \in [1,m], j \in [1,n]\}$, where

$$C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta int_{ij}, \delta ext_{ij}, \tau_{ij}, \lambda_{ij}, ta_{ij} >$$

is a Cell-DEVS atomic component;
$\boldsymbol{B}$ is the border cells set, where

- $B = \{\varnothing\}$ if the cell space is wrapped; or
- $B = \{C_{ij} / \forall\ (i = 1 \vee i = m \vee j = 1 \vee j = n) \wedge C_{ij} \in C\}$, where

$$C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta int_{ij}, \delta ext_{ij}, \tau_{ij}, \lambda_{ij}, ta_{ij} >$$

  is a Cell-DEVS atomic model (i.e., border cells have behavior different from the rest);

$\boldsymbol{Z}$ is the translation function, defined by

- Z: $P_{ij}^{Y_q} \rightarrow P_{kl}^{X_q}$, where $P_{ij}^{Y_q} \in I_{ij}$, $P_{kl}^{X_q} \in I_{kl}$, $q \in [0,\eta]$ and $\forall\ (f,g) \in N$, $k = (i + f) \bmod m$; $l = (j + g) \bmod n$; and
- $P_{kl}^{Y_q} \rightarrow P_{ij}^{X_q}$, where $P_{kl}^{Y_q} \in I_{kl}$, $P_{ij}^{X_q} \in I_{ij}$, $q \in [0,\eta]$ and $\forall\ (f,g) \in N$, $k = (i - f) \bmod m$; $l = (j - g) \bmod n$; and

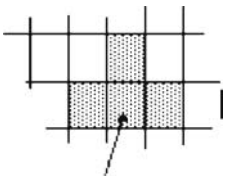
  select is the tie-breaking selector function, with the restriction that $select \subseteq mxn \rightarrow mxn / \forall\ E \neq \{\varnothing\}$, $select(E) \in E$.

First, as in any coupled DEVS model admitting inputs and outputs, sets X and Y are included. Here, the cell space C is a coupled model defined as a fixed size ($m \times n$) array of atomic cell models. Each cell has a set of η neighbor cells, defined by the neighborhood set (N). The set is represented as a list of pairs defining the relative position between the neighbor and the origin cell. The B set defines the cell's space border, and it can be defined in two ways. If $B = \{\varnothing\}$, every cell in the space has the same behavior. Cells in one border are connected with those in the opposite one using the neighborhood relationship. Otherwise, the border cells will have behavior different from that of the rest of the model. They can, for instance, self-generate their state or consume the state of their neighbors.

Finally, the Z function allows defining the coupling between cells in the model. This function translates the outputs of the mth output port in cell C_{ij} into values for the mth input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood, as discussed in Zeigler [9].

The ports' names are generated using the following notation: $P_{ij}^{X_q}$ refers to the qth input port of cell C_{ij}, and $P_{ij}^{Y_q}$ to the qth output port. These ports correspond with the port names denoted as X_q or Y_q for each cell. The number of the cell to be coupled to will be generated by adding the numbers in the neighbor's list to the present cell number. The first output port of a cell will be connected to the first input port of the neighbor, according to the order of the list.

A sketch of this procedure can be seen in Figure 3.5. Figure 3.5(a) shows the neighborhood of cell (i,j) and its representation using the neighbor's list. Figure 3.5(b) shows how the first output port of cell (i,j) is connected with the first input port of the first neighbor in the list, the second port with the second neighbor, etc. On the other hand, for the input ports, the connection is done through the inverse neighborhood list. For each pair (i,j) in the neighborhood, the pair $(-i, -j)$ must be included in this list.

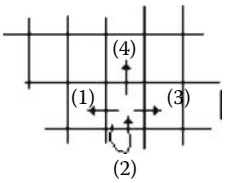


Cell i,j

Neighborhood list:
{ (0,−1), (0,0), (0,1), (−1,0) }

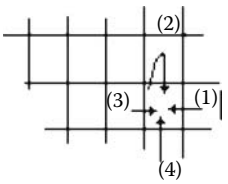
Note: −1: left, up;
1: right, down

(a)



$P_{ij}{}^{Y1} \rightarrow P_{i,j-1}{}^{X1}$ (1)
$P_{ij}{}^{Y2} \rightarrow P_{ij}{}^{X2}$ (2)
$P_{ij}{}^{Y3} \rightarrow P_{i,j+1}{}^{X3}$ (3)
$P_{ij}{}^{Y4} \rightarrow P_{i-1,j}{}^{X4}$ (4)

(b)



Inverse Neighborhood list:
{ (0,1), (0,0), (0,−1), (1,0) }

$P_{ij}{}^{X1} \leftarrow P_{i,j+1}{}^{Y1}$ (1)
$P_{ij}{}^{X2} \leftarrow P_{ij}{}^{Y2}$ (2)
$P_{ij}{}^{X3} \leftarrow P_{i,j-1}{}^{Y3}$ (3)
$P_{ij}{}^{X4} \leftarrow P_{i+1,j}{}^{Y4}$ (4)

(c)

FIGURE 3.5 (a) A cell, its neighborhood, and the neighbor's list; (b) connection of the output ports of cell *i,j* (using the neighborhood list); (c) connection of the input ports of cell *i,j* (using the inverse neighborhood list).

Finally, two extra sets are needed. ***Xlist*** is a list of cells' positions where the model's external events are received. ***Ylist*** is a list of cells' positions whose outputs will be collected to be sent to other models in the hierarchy. The values of these cells will be considered the inputs and outputs of the complete cell space.

The ***select*** function is defined as a list of positions in the neighborhood. The list is ordered according to the selection criteria to be used when more than one cell is active simultaneously.

The definition for DEVS coupled models was changed to allow the definition of cell spaces. If the model considered is a cell space, the coupling uses the internal and external definition of input and output cells. Therefore, DEVS coupled models can be defined as

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select > \tag{3.6}$$

where

$\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{D}$, $\{\boldsymbol{M_i}\}$ and $\boldsymbol{I_i}$ are defined as in Chapter 2, and

$\boldsymbol{Z_{ij}}$ is the i to j translation function, where

Z_{ij}: $Y_i \rightarrow X_j$ if none of the models involved are Cell-DEVS, or

Z_{ij}: $Y(f,g)_i \rightarrow X(k,l)_j$, with $(f,g) \in Ylist_i$ and $(k,l) \in Xlist_j$ if either of the models is a Cell-DEVS.

The Z_{ij} function translates the outputs into inputs between one cell and the models external to the cell space by using the two previously defined lists. To exemplify the external coupling definition, let us consider the models seen in Figure 3.6. The coupling will be done following the Z_{ij} function definition defined in Figure 3.6(c), which was built using the contents of Xlist and Ylist, as can be seen in Figure 3.6(b). The names of the input and output ports are defined by using the contents of the Xlist and Ylist. The port names will be automatically generated by using an identifier (*X* for input, *Y* for output) and a cell position.

As we can see, the present definition for coupled models only allows binary states. The definition can be extended by considering *X* and *Y* (and the corresponding I/O ports) as sets in $\boldsymbol{Z}$ or $\boldsymbol{R}$. The transition functions should compute their results in any of these domains.

Exercise 3.1

Write the formal specification for the coupled model in Figure 3.7, that considering it uses an extended von Neumann's neighborhood.

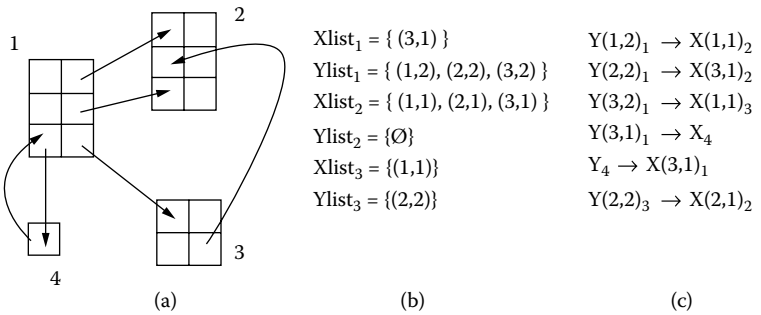


FIGURE 3.6 Example of connection using Z_{ij} function for Cell-DEVS spaces: (a) basic models; (b) X_i and Y_i lists for each model; (c) Z_{ij} coupling.

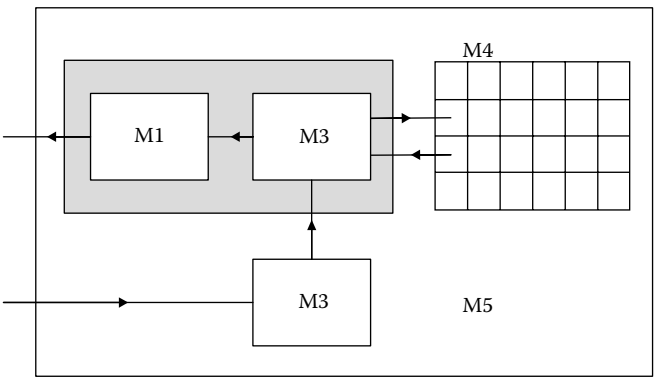


FIGURE 3.7 A model with DEVS and Cell-DEVS components.

3.5 AN APPLICATION EXAMPLE

In this section, we provide a detailed example for application of the formalisms previously presented. Figure 3.8 shows the structure of the model we will introduce here, which has the goal of simulating traffic in a section of urban population.

Model A in Figure 3.8 is a Cell-DEVS model of pollution in the residential neighborhood. The model represents the spreading of particles of smoke on the air, influenced by traffic—in this case, the highway (model C)—and smoke of trucks and CO emissions—in this case, from the factory (model D, a DEVS model). The local computing function models the influence of the wind on the smog: if a particle stays in a cell for some time, it is diffused to the neighbors. If not, the wind removes it (i.e., we can use an inertial delay to model spreading of the particles). The structure of this model can be seen in Figure 3.9.

Model B, whose structure is introduced in Figure 3.9(b), represents the traffic movement in a commercial neighborhood. The streets are one way, and no traffic lights are modeled. Vehicles move forward and do not pass each other (because the streets are one lane). Transport delays are used to model the vehicle's speed.

The Cell-DEVS model C represents a one-way highway that passes between regions A and B. Traffic flow in the highway will be represented using a cellular model of the traffic flow in one-way routes. The model serves to study the traffic flow on the highway and its influence on the rest of the city. Atomic model D represents a factory, with trucks arriving from the highway and other trucks moving onto the highway. The simulation results could be used to schedule the input and output of trucks to the factory, thus improving the flow of products.

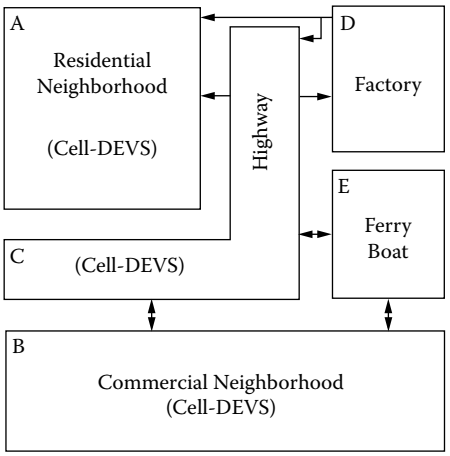


FIGURE 3.8 Coupling of Cell-DEVS and other DEVS models.

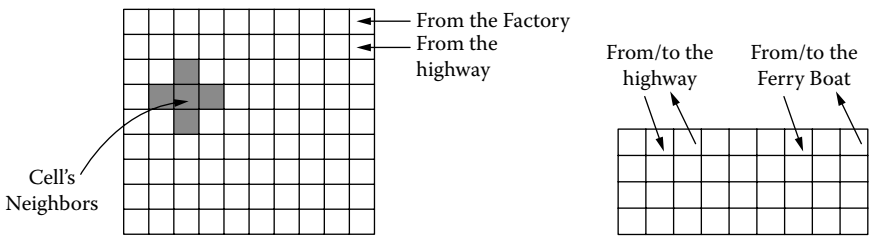


FIGURE 3.9 Structures of model A (pollution in residential neighborhood) and model B (traffic in commercial neighborhood).

Finally, atomic model E (Figure 3.9) represents the entrance to a ferryboat connecting the city with an island. This model could be used to study the traffic flow to the ferryboats (to determine the optimal number of boats to be used, depending on the hour).

The behavior for model C (the highway model) is described as follows:

$$C = < X, Y, S, N, type, d, \tau, \delta int, \delta ext, \lambda, ta > \tag{3.7}$$

$X = Y = S = \{0, 1\}$; (0: empty cell; 1: vehicle in the cell);
$N = \{ (0,0), (1, 0), (–1,0), (0,1), (0, –1), (1, 1), (–1, 1), (1, –1), (–1, –1) \}$;
type = transport;
d = random (average speed = 2); and
δ_{int}, δ_{ext}, λ, and *ta* are defined using Cell-DEVS specifications.

The movement of a car is not explicitly registered in only one rule; rather, the state change takes two movements. The first rule represents the arrival of a new car to the cell (using forward movement or passing stopped vehicles, as seen in Figure 3.11), or a car that cannot move due to a bottleneck situation. Each subexpression represents either of the movements depicted in Figure 3.11.

The corresponding cell is activated when a new input value is received. Then movement rules are computed. If the state changes from 1 to 0, the influencees are activated and the local function computed, leading to new car movement.

The second rule reflects the vehicle abandoning the cell, as shown in Figure 3.12.

τ(N)	N
1	((0,0) = 0 AND (0,−1) = 1) OR // Normal flow ((0,0) = 0 AND (0,−1) = 0 AND (1,−1) = 1 AND (1,0) = 1) OR // Passing on the left ((0,0) = 0 AND (0,−1) = 0 AND (−1,−1) = 1 AND (−1,0) = 1)) OR // Pass on the right ((0,0) = 1 AND Colum n_3 = 1) /* Bottleneck */

0	((0,0) = 1 AND (0,1) = 0) OR // Normal flow
	((0,0) = 1 AND (0,1) = 1 AND (−1,1) = 0 AND (−1,0) = 0) OR // Passing on the left
	((0,0) = 1 AND (0,1) = 1 AND (1,1) = 0 AND (−1,1) = 1 AND (1,0) = 0) // On the right
	OR ((0,0) = 0) // Empty cell not considered in rule 1

FIGURE 3.10 Specification for model C.

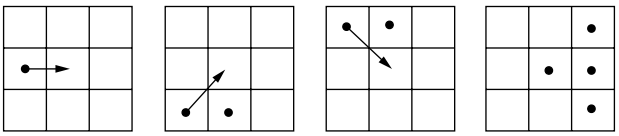


FIGURE 3.11 Valid movements for rule 1 (different expressions).

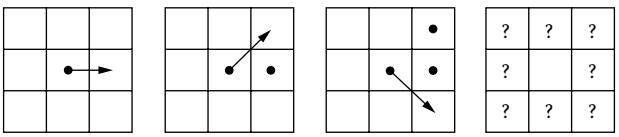


FIGURE 3.12 Valid movements for rule 2 (different expressions).

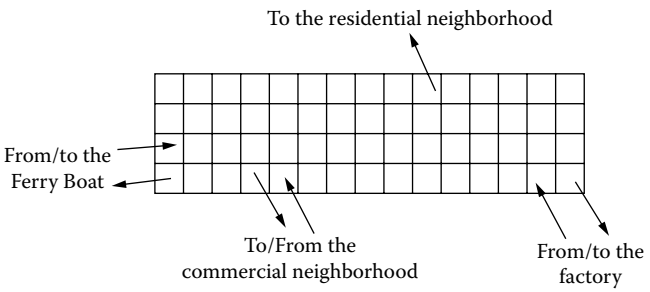


FIGURE 3.13 Cell space C: input/output cells of the model.

The transport delays allow modeling the acceleration delay of the cars. The car movement is delayed prior to the next movement to the following cell, allowing us to model different speeds. The delay could be represented as a random number to model the different speeds of each car.

Figure 3.13 shows the structure of the Cell-DEVS coupled model for this example.

The formal definition for the cell space is

$$C = < X_C,\ Y_C,\ Xlist_C,\ Ylist_C,\ \eta_C,\ N_C,\ \{t1_C,\ t2_C\},\ C_C,\ B_C,\ Z_C,\ select_C> \tag{3.8}$$

where

$\boldsymbol{X_C} = \boldsymbol{Y_C}$ ={0, 1};

$\boldsymbol{Xlist_C}$ = { (4,14), (4,5), (3, 4) };

$\boldsymbol{Ylist_C}$ = { (1,10), (4,4), (4,15), (4,1) }.

$\eta_C = 9$;
N_C = { (–1, –1), (–1,0), (–-1,1), (0, –1), (0,0), (0,1), (1, –1), (1, 0), (1, 1) };
$t1_C = 4$; $t2_C = 15$;
B_C = {∅};
C_C is the cell space set, defined as in the previous example.
Z_C is defined using the coupled model's formal specification as follows:

$$P_{ij}^{Y1} \rightarrow P_{i,j-1}^{X1} \qquad P_{i,j+1}^{Y1} \rightarrow P_{ij}^{X1}$$
$$P_{ij}^{Y2} \rightarrow P_{ii+1,j}^{X2} \qquad P_{i-1,j}^{Y2} \rightarrow P_{ij}^{X2}$$
$$P_{ij}^{Y3} \rightarrow P_{i,j+1}^{X3} \qquad P_{i,j-1}^{Y3} \rightarrow P_{ij}^{X3}$$
$$P_{ij}^{Y4} \rightarrow P_{i-1,j}^{X4} \qquad P_{i+1,j}^{Y4} \rightarrow P_{ij}^{X4}$$
$$P_{ij}^{Y5} \rightarrow P_{ij}^{X5} \qquad P_{ij}^{Y5} \rightarrow P_{ij}^{X5}$$
$$P_{ij}^{Y6} \rightarrow P_{i-1,j-1}^{X6} \qquad P_{i-1,j-1}^{Y6} \rightarrow P_{ij}^{X6}$$
$$P_{ij}^{Y7} \rightarrow P_{i-1,j+1}^{X7} \qquad P_{i-1,j+1}^{Y7} \rightarrow P_{ij}^{X7}$$
$$P_{ij}^{Y8} \rightarrow P_{i+1,j-1}^{X8} \qquad P_{i+1,j-1}^{Y8} \rightarrow P_{ij}^{X8}$$
$$P_{ij}^{Y9} \rightarrow P_{i+1,j+1}^{X9} \qquad P_{i+1,j+1}^{Y9} \rightarrow P_{ij}^{X9}$$

$\mathbf{select_C}$ = { (0,1), (–1,1), (1, 1), (0,0), (–1,0), (1, 0), (–1, –1), (0, –1), (1,–1) }.

Finally, let us consider the formal specification for the complete model, presented in Figure 3.14. This model can be formally defined as

$$M = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select > \tag{3.9}$$

where
$X = Y$ = {∅};
D = { A, B, C, D, E }, and $\forall i \in D$, M_i is one of the basic DEVS models previously defined;
I_i is the set of influencees of model i. In this case,
I_A = {∅};
I_B = { C, E };
I_C = { A, B, D, E };
I_D = { A, C }; and
I_E = { B, C }.

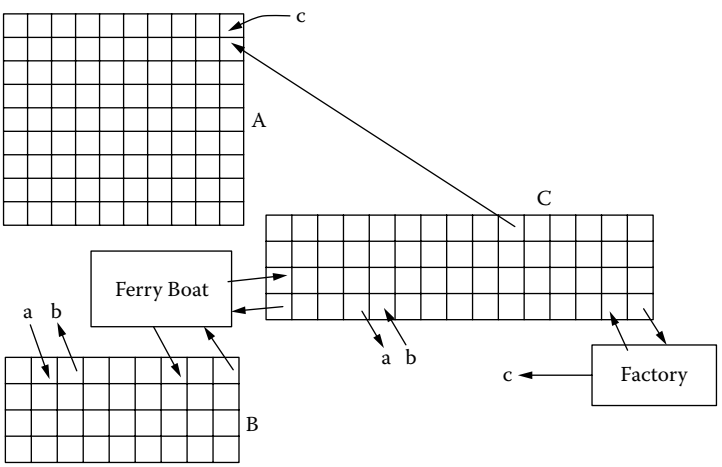


FIGURE 3.14 Model M's structural definition.

The Z_{ij} function is defined as

Z_{BC}: $Y(1,3)_B \rightarrow X(4,5)_C$
Z_{BE}: $Y(1,9)_B \rightarrow IN_f$
Z_{CA}: $Y(1,10)_C \rightarrow X(2,10)_A$
Z_{CB}: $Y(4,4)C \rightarrow X(1,2)_B$
Z_{CD}: $Y(4,15)_C \rightarrow IN_f$
Z_{CE}: $Y(4,1)_C \rightarrow IN_b$
Z_{DA}: $OUT_f \rightarrow X(1,10)_A$
Z_{DC}: $OUT_f \rightarrow X(4,14)_C$
Z_{EB}: $OUT_b \rightarrow X(1,7)_B$
Z_{EC}: $OUT_b \rightarrow X(3,4)_C$

Finally, ***select*** = { C, A, B, D, E}.

Exercise 3.2

Write a formal specification of model A using the previous description. Build a definition for the atomic cell and for the coupled model.

Exercise 3.3

Repeat the previous exercise for model B.

Exercise 3.4

Define model D as a DEVS model representing the flow of trucks at the factory and model E, which shows the flow of vehicles to the ferryboat. In both cases, the models are constructed as queuing servers simulating the arrival and departure of cars. The values could be generated, for instance, by using random numbers based on a Poisson distribution.

3.6 SUMMARY

The Cell-DEVS formalism allows defining cellular models based on the discrete-event system specification. Cell-DEVS allows defining asynchronous cell spaces with explicit timing definition. This approach is still based on the formal specifications of DEVS, but it allows the user to focus on the problem to be solved by using simple rules for modeling (like with CA). Explicit timing delay constructions can be used to define precise timing in each cell.

This approach allows enhancing the modeling experience in different aspects. In terms of performance, only active cells execute their local computing function, and the execution results are spread out after a predefined delay (only if a state change has occurred). The delay function provides a natural mechanism for defining timing information.

The modeling technique permits keeping the ability of CA to describe complex systems using very simple rules, while also permitting us to bridge the gap between a discrete time and a discrete event description like DEVS. The use of DEVS as the basic formal specification mechanism enables us to define interactions with models defined in other formalisms. Individual cells can provide data to those models; integration between them could enable defining of complex hybrid systems and multimodels developed with different techniques and integrated through a DEVS interface. This approach provides "evolvability" of the models through a technique that is easy to understand and to map into other existing techniques, while having the potential of evolving into complex models.

REFERENCES

1. Chandrupatla, T., and A. Belegundu. 1997. *Introduction to finite elements in engineering.* Upper Saddle River, NJ: Prentice Hall.
2. Wolfram, S. 1986. *Theory and applications of cellular automata,* vol. 1. Singapore: World Scientific.
3. Wolfram, S. 2002. *A new kind of science.* Champaign, IL: Wolfram Media.
4. Gutowitz, H. 1995. Cellular automata and the sciences of complexity. Parts I–II. *Complexity* 1:16–22.
5. Toffoli, T., and N. Margolus. 1987. *Cellular automata machines: A new environment for modeling.* Cambridge, MA: MIT Press.
6. von Neumann, J. 1966. *Theory of self-reproducing cellular automata.* Urbana: University of Illinois Press.
7. Wainer, G., and N. Giambiasi. 2002. *N*-dimensional cell-DEVS. *Discrete Events Systems: Theory and Applications* 12:135–157.
8. Wainer, G. 1998. Discrete-event cellular models with explicit delays. PhD thesis, Université d'Aix-Marseille III, France.
9. Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.

Section 2

Building Simulation Models: The CD++ Toolkit

4 Introduction to the CD++ Toolkit

4.1 INTRODUCTION

In this chapter, we introduce the basic features of the CD++ toolkit. CD++ is one of several tools that have been implemented based on DEVS theory and its extensions. The level of interest from the community can be seen in the folowing list, which includes a list of some of the existing DEVS modeling and simulation (M&S) toolkits (this is noncomprehensive because new efforts are ongoing worldwide):

- ADEVS [1] provides a C++ library based on DEVS, which developers can use to build their own models, and supports integration with other simulation environments.
- DEVS-Ada/Tw was the first attempt to combine DEVS and the Time Warp parallel simulation algorithm over a multiprocessor environment. DOHS, the distributed optimistic hierarchical simulation scheme, combines DEVS and Time Warp, implemented in D-DEVSim++. This alternative presents a more general approach for distributed optimistic execution of DEVS models, while addressing some restrictions introduced in DEVS-Ada/TW [2].
- DEVS-C++ [3] is a DEVS-based modeling and simulation environment written in C++, which implements parallel execution and supports large-scale systems.
- DEVS-Scheme [4,5] is a knowledge-based environment for modeling and simulation based on the Scheme functional language (a variation of Lisp).
- DEVS/HLA [6,7] is based on the high-level architecture (HLA) [8]. It was used to demonstrate how an HLA-compliant DEVS environment could improve the performance of large-scale distributed modeling and simulation.
- DEVSJAVA [9] is a DEVS-based modeling and simulation environment written in Java. It provides classes for the users to implement their own DEVS models.
- DEVSim++ [10] is an object-oriented DEVS simulator implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- GALATEA [11] is a simulation platform that offers a language to model multi-agent systems using an object-oriented architecture.
- JAMES [12] implements DEVS theory to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments.
- JDEVS [13] is a DEVS modeling and simulation environment written in Java. It allows general-purpose, component-based, object-oriented, visual simulation of models.
- PyDEVS [14] uses the ATOM3 tool [15] to construct DEVS models and to create the code to be executed. Models are represented as a state graph used to generate Python code and then interpreted by PyDEVS.
- SimBeams [16] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis, and visualization using DEVS.

As we can see, the majority of the existing toolkits support stand-alone simulation, but some (such as DEVS-C++, DEVS/HLA, DEVSCluster, D-DEVSim++, and DEVSJAVA) allow distributed/

parallel execution of DEVS models. The middleware technology used varies from tool to tool, and it includes:

- CORBA (Common Object Request Broker Architecture), an open standard promulgated by the Object Management Group (OMG) [17];
- HLA, a standard specifically designed for distributed simulations [8];
- MPI [18], a message passing interface standard designed for high-performance communication on parallel and distributed environments; and
- Globus (http://www.globus.org), a standard version of grid protocols, created to provide data management, information services, security, and resource management.

Exercise 4.1

Search further information on DEVS simulators and write a comparative study. Include simulation tools not included in this list.

The CD++ tool [19] has been developed following the specifications of DEVS and Cell-DEVS. CD++ information can be found in the tool's Wiki at http://cell-devs.sce.carleton.ca, where the reader will find a complete user manual, installation tools, and the required software application. An open source version of the project can be found at http://sourceforge.net/projects/cdpptoolkit (interested developers are encouraged to participate in the development of the open source version of the simulation tool). Likewise, a repository of models is available for general use. All the software and examples discussed in this book can be found at these two sites, and we encourage the reader to use and modify them for practice. The examples can be found at http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/. From now on, all the examples will refer to the files found in this folder.

The simulation engine tool of CD++ is built as a class hierarchy. Atomic models can be programmed in C++ and incorporated onto a basic class hierarchy. Coupled and Cell-DEVS models are created using a language built in the engine. The following sections will cover the general aspects of the simulation tool, and we will show how to create models using the tools. (As discussed in the preface to this book, the models presented usually are pruned version of the ones found online in order to focus attention on the important aspects, leaving some of the details for the reader interested in using the originals for practice.) Detailed usage instructions can be found in the user manual for the toolkit.

4.2 DEFINING ATOMIC MODELS IN CD++

Figure 4.1 shows an excerpt of the *Atomic* class in CD++ class hierarchy. When implementing a new atomic model, we must define a class derived from *Atomic*, overloading the methods needed. *Atomic* is an abstract class that declares an Application Program Interface (API) to create models, and it defines some service functions the user can use by redefining the base classes. The derived classes can overload the initialization, internal transition, external transition, and output methods. The service functions allow the model to set the current state and its duration. In order to allow parameter configuration at runtime, some of the arguments used by the atomic models can be defined externally (in a coupled model definition file, to be introduced later in this chapter). Then, to define a new atomic model, the user must:

1. Write a class derived from *Atomic* overloading the methods:
   - ***initFunction*** is used to define initial values for the model (the default value for the time advance value is infinite and for the state is passive).
   - ***externalFunction*** implements the external transition function. It is called when an external event arrives in one of the model's input ports.

```
class Atomic : public Model  {
  public:
   virtual ~Atomic() {}  // Destructor

  private:
    State st;

  protected:
   enum State { active, passive } ;

   Atomic( const string &name = "Atomic" ) : Model( name ), st( passive ) {}
    // Constructor

   virtual Model &initFunction() = 0 ;
   virtual Model &externalFunction( const ExternalMessage & ) = 0 ;
   virtual Model &internalFunction( const InternalMessage & ) = 0 ;
   virtual Model &outputFunction( const InternalMessage & ) = 0 ;
   Model &holdIn( const State &, const Time & ) ;
   Model &passivate();
   Model &state( const State &s ) { st = s; return *this; }
   const State &state() const {return st;}
}
```

FIGURE 4.1 Excerpt of the definition of the Atomic class.

- ***internalFunction*** allows defining of the internal transition function. This method is activated when the simulation time is equal to the one scheduled by the time advance function.
- ***outputFunction*** generates outputs; it is called before the internal transition function. *className* is the class name.

2. Modify the **register.cpp** file, adding to the method `MainSimulator::registerNewAtomics()` the new atomic model using the `registerAtomic` method. For instance,

   ```
   SingleModelAdm::Instance().registerAtomic( NewAtomicFunction
     <Queue>(),"myQ");
   ```

   registers a new model named Queue. Here, *Queue* is the name of the class, and *myQ* is the name that will be used to identify the *Queue* model in other files (including the coupled model file that will be explained later). In this way, users can give meaningful names to their models, even if they are used for other purposes. For instance, if this queue is used in the context of a computer network, we could name it "buffer"; if it is needed for a supermarket simulation, we could call it "CustomerLine," making the model easier to read.
3. Recompile the simulator, which will be ready to execute using the new model.

The *Atomic Model* class is provided with a few primitives to interact with the simulator in order to accomplish common operations, including:

- `holdIn(state, time)` is an implementation of the time advance function. The model changes its *state* (i.e., *active*, *passive*) into a new *state*, and it schedules an internal transition to be executed after *time* units.
- `passivate()` allows the model state to change to *passive* with time advance ∞ (this is a macro for `holdIn(passive, Inf)`).
- `sendOutput(time, port, value)` is used to send outputs in the output function.
- `nextChange()` informs the time remaining before the next scheduled state change.
- `lastChange()` records the time of the last change of state.
- `state()` returns the model's state.

Other components of the tool provide some extended service functions. For instance, the method `getParameter(modelName, parameterName)`, which is defined within `MainSimulator::Instance()`, is used to explore the coupled model definition file in order to obtain the `parameterName` value associated with the model `modelName`. For instance, `MainSimulator::Instance().getParameter(description(), "max"))` will search a parameter called *max* in the model file. This parameter should be associated with the model whose name is defined by *description()*. The method `existsParameter` is used to check if the parameter has been defined (with error-checking purposes). For instance, in the previous case, `MainSimulator::Instance().existsParameter(description(), "max"))` will check if the parameter *max* is defined in the coupled model file.

The *getParameter* method returns a string, so we must be careful to convert the value to the desired type. For instance,

```
max = str2Int(MainSimulator::Instance().getParameter
   (description(),"max"));
```

will convert the value of the parameter read into an integer.

The input/output ports carry messages of the type *message*, which are handled by two methods: *port()* and *value()*. In order to ensure correct assignment of the values, static casts should be used in handling message values. For instance,

```
int no = static_cast <int> (msg.value());
```

will convert the message value into an integer number.

Another set of support methods is included in the `distri.h` file, which contains a variety of random number generators with different probability distributions. The following methods are available:

- `Distribution` is the base class used by every type of distribution. The method `*create(const string &distributionName)` is used to define the name of the distribution to be used, and the methods `set()` and `get()` are used to set up the parameters needed to generate the random numbers. Because the *Distribution* method has been built to be generic (and open to include other distributions in the tool), we might need to define how many arguments are needed for a given probabilistic distribution. To do so, we use the method `setVar`. When we invoke `dist->setVar(i, str2float(parameter))`, the method will set the number of arguments needed. If we need to configure the selection of the probability distribution to be chosen externally (by defining it in the coupled model file), we could execute:

  ```
  dist = Distribution::create( MainSimulator::Instance().
     getParameter(description(),"distribution") );
  ```

which searches for the *distribution* keyword in the coupled model file, and it creates an object called *dist* that we can use to generate random numbers with the chosen distribution. For every available distribution, the method `get(unsigned int)` returns the random number generated. The following methods are derived from *Distribution*:

- `ChiDistribution` is used to generate random numbers using a Chi Square (χ^2) probabilistic distribution. The method `&set()` is used to define the degrees of freedom (a positive real number).
- `NormalDistribution` generates random numbers according to the *normal* distribution. Here, `&set()` is used to define the mean and standard deviation.
- `PoissonDistribution` generates random numbers according to the Poisson distribution. Here, `&set()` is used to set the expected number of occurrences during a given interval (a positive real number).
- `ExponentialDistribution` generates random numbers according to the Exponential distribution. Here, `&set()` is used to define the rate parameter.

- The file `mathincl.h` includes a series of mathematical methods and constants. The `time.h` file includes a number of methods to handle simulated time according to the format used in CD++, as follows:
  - `Time( Hours h = 0, Minutes m = 0, Seconds s = 0, MSeconds ms = 0 ): hour(h), min(m), sec(s), msec(ms)` is used to set a variable of type *Time*. For instance, `timeout = Time (0, 1, 0, 0);` initializes the timeout variable to 1 min.
  - `Time( const string &t ) { makeFrom(t); }` converts variable with *Time* format into text.
  - Methods `&hours( const Hours & ), &minutes( const Minutes & ), &seconds( const Seconds & ), &mseconds( const MSeconds & )` are used to set each of the corresponding time values. Methods `const Hours &hours(), const Minutes &minutes(), const Seconds &seconds()`, and const MSeconds &mseconds() are used to query each of the values.
  - `+, -, =, ==, -=, +=, <` can be used to manipulate time variables (with the obvious meaning).
  - `asMsecs()` converts the time into milliseconds.
  - `Zero` and `Inf` are two predefined constants used to define time 0 and ∞. For instance, if we want to generate a time value at random, we can execute

    ```
    Time t( fabs( this->distribution().get() ) ) ;
    ```

    which will get a random number according to the desired distribution (selected in the model file), take the absolute value of it (milliseconds), and save it into a *Time* object named *t*.

4.3 AN EXAMPLE: QUEUE MODEL

As mentioned in Chapter 2, the GPT model is usually employed as a "Hello, World!" application for DEVS simulators. This model is included in every distribution of CD++. (For instance, if we download the Linux version from Sourceforge or from the CD++ Web site, we will find the files *cpu.cpp*, *generat.cpp*, *transduc.cpp*, and *queue.cpp*. If we install CD++Builder, we will find them in *eclipse\plugins\CD++Builder_1.1.0\internal*.)

Our model of a queue will hold any type of user-defined values. The model is based on the one introduced in Chapter 2: it uses three input ports and one output port. Task identifiers are stored in the queue as they are received through the input port *in*. When the queue receives an input in the input port *done*, we know the receiver is ready to receive more work, and the first element in

```
class Queue : public Atomic {
public:
   Queue(); // Default constructor

protected:
   Model &initFunction();
   Model &externalFunction( const ExternalMessage & );
   Model &internalFunction( const InternalMessage & );
   Model &outputFunction( const InternalMessage & );

private:
   const Port &in, &stop, &done;
   Port &out;
   Time preparationTime;

   typedef list<Value> ElementList ;
   ElementList elements ;

   Time timeLeft;
}; // class Queue
```

FIGURE 4.2 *Queue.h*: model definition.

the queue is transmitted through the port *out*. The input port *stop* serves to regulate the flow: if we receive a message on the input port *stop*, we temporarily disable the queue (i.e., it only responds to new events received through the input port *in*). Any input received will be stored, but no output will be sent until the queue is enabled again (by sending another message to the port *stop*). The parameter *preparationTime* is used to model the delay of the queuing device.

To create this model in CD++, we first need to define a class to store the state of the queue. Figure 4.2 lists the queue state class declaration and definition. The example shows the definitions needed to implement the queue model (according to the specifications in Chapter 2), including I/O ports and state variables. The list of values that holds the input data (*ElementList*) is defined using the standard template library (STL [20]), and *timeLeft* is used to store the time remaining if the model is interrupted by a control flow signal.

The constructor, presented in Figure 4.3, creates the input and output ports of the model and sets the default value of the variable *preparationTime* in 10 s.

The *addInputPort* and *addOutputPort* definitions physically create the I/O ports, and they give names to be used in the model's coupled model file (these names are case sensitive). As discussed earlier, *getParameter* queries the coupled model file (described in the following section) and searches for the parameter identified with the *preparation* keyword. The value of the parameter (a string) is converted into the initial preparation time (overriding the default value if needed). The initialization function generates an empty queue (Figure 4.4).

As we can see, the queue is managed using the STL [20], whose services can be used throughout the different models. The default *ta* time is ∞, so the following state change will take place only when an external event arrives (which is why the time advance function is not programmed; if an internal transition is needed prior to the arrival of an external event, the *holdIn* method should be used to schedule one). Figure 4.5 shows the activation of the external transition function upon arrival of messages in the input ports.

```
Queue::Queue() : preparationTime( 0, 0, 10, 0 ), in(this->addInputPort("in")),

           stop(this->addInputPort("stop") ),  done(this->addInputPort("done")),

           out(this->addOutputPort("out") )  {
        this->description( "Queue" ) ;
        string time( Simulator::Instance().getParameter(this->description(),
           "preparation" ) ) ;
        if( time != "" ) preparationTime = time ;
}
```

FIGURE 4.3

```
Model &Queue::initFunction(){
   elements.remove( elements.begin(), elements.end() ) ;
   return *this;
}
```

FIGURE 4.4

```
Model &Queue::externalFunction( const ExternalMessage &msg ){
   if( msg.port() == in )  {
      elements.push_back( msg.value() ) ;
      if( elements.size() == 1 ) this->holdIn( active, preparationTime );
   }

   if( msg.port() == done ) {
      elements.pop_front() ;
      if( !elements.empty() ) this->holdIn( active, preparationTime );
   }

   if( msg.port() == stop )
      if( this->state() == active && msg.value() )    {
         timeLeft = this->nextChange();
         this->passivate();
      }
      else
         if( this->state() == passive && !msg.value() )
            this->holdIn( active, timeLeft );
   return *this;
}
```

FIGURE 4.5

```
Model &Queue::outputFunction( const InternalMessage &msg ){
   this->sendOutput( msg.time(), out, elements.front() ) ;
   return *this ;
}
```

FIGURE 4.6

```
Model &Queue::internalFunction( const InternalMessage & ){
   this->passivate();
   return *this ;
}
```

FIGURE 4.7 *Queue.cpp*: model definition.

An event that arrives in the *in* port represents a new input value, which has to be queued. If it is the only element in the queue, it has to be retransmitted immediately. Hence, we schedule our internal event after *preparationTime* (which represents the delay of the queuing device). An event that arrives in the port *done* indicates that the last element sent has been processed, and therefore it has to be erased from the queue. If there are more elements to be transmitted, the first value in the queue should be prepared. An event that arrives in the port *stop* indicates that the flow should be stopped or restarted. If the queue was in *active* state and the message value is not zero, the queue will pause. Here, the time remaining to process the next state change is calculated (end of preparation time) and then the queue changes its state to *passive* by calling the *passivate* method. If the queue was in *passive* state and the message value is zero, then the queue restarts, and the next state change is scheduled after the remaining processing time.

When the preparation time interval expires, the *outputFunction* shown in Figure 4.6 is invoked, and the first value in the queue is transmitted through the output port *out*. After calling the output function, the internal transition function shown in Figure 4.7 is invoked. Here, there is nothing to do except to wait for the acknowledgment at the *done* port. Thus, we passivate the model.

Exercise 4.2

Modify the Queue model to implement preemption. Every 5 min of simulated time, the queue must be completely emptied (this depends only on the internal state, so the preemption procedure should be implemented in the internal transition function).

Once the new atomic model is created, we need to link it to the CD++ simulator. To do so, we register the model using `MainSimulator::registerNewAtomics`, as explained earlier. Figure 4.8 shows how to modify the *register.cpp* file.

```
void MainSimulator::registerNewAtomics() {
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Queue>() , "Queue" ) ;
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Generator>() , "Generator" ) ;
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<CPU>() , "CPU" ) ;
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Transducer>() ,
  "Transducer" ) ;
}
```

FIGURE 4.8 Contents of *register.cpp*.

EXERCISE 4.3

"Reverse engineer" the source code for the models Queue, Generator, CPU, and Transducer found in CD++, and write a DEVS formal specification for each of them. Compare with the specifications introduced in Chapter 2 and discuss the differences and similarities between the different models' versions.

4.4 COUPLED MODEL DEFINITION

Coupled models in CD++ are defined using a specification language specially defined for this purpose. The language was built following DEVS formal definitions if it is defined in a model configuration file. Optionally, configuration values for the atomic models can be defined in the same configuration files as the coupled model, and these values will be queried by the function `getParameter(modelName, parameterName)`, as discussed in the previous section.

The coupled model at the top level is always defined using the `[top]` clause. As shown in the formal specifications presented in Chapter 2, we must define components, input/output ports, and links between models. The following syntax is used:

- **components:**

  ```
  components: model_name1[@atomicClass1] model_name2[@atomic-
  Class2]…
  ```

  This construction lists the components of the coupled model (this clause is mandatory). A coupled model might have atomic models or other coupled models as components. For atomic components, an instance name and a class name must be specified (this allows a coupled model to use more than one instance of the same atomic class). For coupled models, only the model name must be given. This model name must be defined as another group in the same model configuration file.

- **out:**

  ```
  out : portname1 portname2 ...
  ```

  This construction enumerates the model's output ports. This clause is optional.

- **in:**

  ```
  in : portname1 portname2 ...
  ```

  This clause, which is also optional, enumerates the input ports.

- **link:**

  ```
  link : source_port[@model] destination_port[@model]
  ```

  This clause defines the links between components (internal couplings, i.e., IC), and between components and the coupled model itself (External input/output couplings—EIC/EOC) in DEVS formal specification). If the name of the model is omitted, it is assumed that the port belongs to the coupled model being defined (representing an external coupling; this was represented as *Self* for the EIC/EOC connections in Chapter 2).

Figure 4.9 shows a description of the GPT model (which is a variation of the same model introduced in Chapter 2).

```
[top]
components : transducer@Transducer generator@Generator Consumer
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

[Consumer]
components : queue@Queue processor@Processor
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
```

FIGURE 4.9 Example for the definition of a DEVS coupled model.

The *top* model includes three components: two atomic (*transducer* and *generator*) and a coupled model (*Consumer*, which is composed of *queue* and *processor*). The model has one output port (*out*), and the *Link* clauses show how outputs on one model are connected to inputs in the other. For instance, outputs in the *out* port of the generator are sent to the input port *in* of the *Consumer* and the *arrived* port of the *transducer*. In the consumer model, the outputs at port *out* of the processor are converted as outputs of the coupled model (through the *out* port).

Exercise 4.4

Based on the definition of Figure 4.9, draw a graphical representation of the structure of the coupled model.

Exercise 4.5

Based on the drawing done in Exercise 4.4, write a formal specification of the coupled model. Compare with the one introduced in Chapter 2. Show differences and discuss similarities.

Exercise 4.6

Suppose that we need to create an experimental frame for testing the model. In order to do so, we will create a coupled model as in Figure 4.10, and we will feed input data to each of the input ports to test the model's behavior. Run a simulation given the following input events:

```
00:00:10:00 in 1.5;
00:00:18:00 done 1.5;
00:00:30:00 in 3;
00:00:45:00 done 3;
00:00:50:00 in -7;
00:00:52:00 done -7.
```

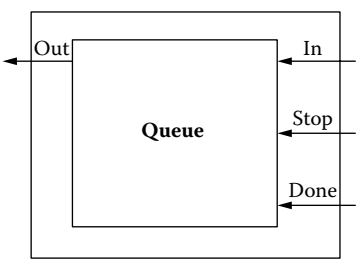


FIGURE 4.10 Testing the queue model.

Discuss the simulation results (details on event files and compilation and simulation logs can be found in CD++ user manuals).

In addition to the coupled model definitions, the model file might include user-defined parameters. In this case, the parameters are specified in a group with the model's name, which is the name used in the *components* clause (and not one used to define the atomic class name). This identifier

is the one used by the *getParameter* method introduced in the previous section. Figure 4.11 shows the syntax for the user-defined values (spaces are *mandatory*).

Figure 4.12 shows the definition of a model with two instances of the atomic class *processor* using different values for the user-defined parameters. In this case, we have one instance of the *processor* using exponential random numbers with a mean of 10 and another instance using a Poisson distribution with a mean of 50.

```
[model_name]
var_name1 : value1
...
var_namen : valuen
```

FIGURE 4.11 User-defined values for atomic models.

EXERCISE 4.7

Modify the GPT initial parameters and rerun the simulation. The model can be found in *./transd.zip* at the URL http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/ (this model is precompiled in CD++; in order to change the parameters, the model configuration file must be changed).

EXERCISE 4.8

Modify the GPT model and include two processors and two queues in tandem. Execute a simulation and analyze the results obtained in the output and log files.

EXERCISE 4.9

Modify the GPT model to execute three processors and their queues in pipeline mode (the output of one should be connected to the inputs of the other). Simulate the model and analyze the results obtained in the output and log files.

4.5 DEFINING CELL-DEVS MODELS

CD++ also includes a specification language to describe Cell-DEVS models. These definitions are based on the formal specifications defined earlier and can be completed by considering a few parameters: size, influences, neighborhood, and borders. These are used to generate the complete cell space. The behavior of the local computing function is defined using a set of rules with the form: POSTCONDITION DELAY {PRECONDITION}. When the *PRECONDITION* is satisfied, the state of the cell changes to the designated *POSTCONDITION*, and the state change will be transmitted after the *DELAY*. If the *PRECONDITION* is *false*, the next rule in the list is evaluated until a rule is satisfied or there are no more rules. In the latter case, an error is raised, indicating that the model specification is incomplete. The existence of two or more rules with same *PRECONDITION* but with different state values or delays can also be detected, avoiding the creation of ambiguous models.

```
[top]
components : Queue@queue Processor1@processor Processor2@processor
...

[processor]
distribution : exponential
mean : 10

[processor2]
distribution : poisson
mean : 50

[queue]
preparation : 0:0:0:0
```

FIGURE 4.12 Example of setting parameters to DEVS atomic models.

In CD++, Cell-DEVS models are a special case of coupled models. Then, when defining a cellular model, all the coupled model parameters are available and, in addition, some extra parameters are needed to define the dimensions of the cell space, the delay type, the default initial values, and the local transition rules. The main parameters to be used in the rest of the book are presented in the following list (a comprehensive list can be found in the user manual):

- **type : [CELL]** indicates the **cell** keyword that must be specified for Cell-DEVS models.
- **width** : an integer value that defines the width of the cell space.
- **height** : an integer value that defines the height of the cellular space model.
- **dim** : $(x_0,\ldots,x_n)$ height and width are provided for backward compatibility with older versions of the tool, and they are used for two-dimensional cellular spaces only. Instead, **dim** can be used for any n-dimensional cell space. All the x_i values must be integers. The vector that defines the dimension of the cellular model must have two or more elements; thus, for one-dimensional cellular models, we must use the form: $(x_0, 1)$. When referencing a cell, all references must satisfy:

$$(y_0,\ldots, y_n),\ 0 \le y_i < x_i,\ i = 0,\ldots, n, \qquad \text{with } y_i \text{ an integer value}$$

- **In** : defines the input ports for the cellular model.
- **Out** : defines the output ports for the cellular model.
- **Link** : defines the components' external coupling (*Xlist* and *Ylist* in the formal specifications of Cell-DEVS). For a coupled Cell-DEVS model, each component is an individual cell reference for receiving or transmitting data. A cell reference is of the form `cellName(x1,...,xn)`. Valid link definitions are of the form:

```
Link : outputPort                    inputPort@cellName(x1,...,xn)
Link : outputPort@cellName(x1,...,xn) inputPort
Link : outputPort@cellName(x1,...,xn) inputPort@cellName(x1,...,xn)
```

- **Border : [WRAPPED | NOWRAPPED]** defines the type of border used (the default is NOWRAPPED). For nonwrapped borders, a reference to a cell outside the cellular space will return the *undefined* value. (The symbol ? represents the *undefined* value.)
- **Neighbors** : cellName $(x_{1,1},\ldots,x_{n,1})$ … cellName $(x_{1,m},\ldots,x_{n,m})$ defines the neighborhood for all the cells of the model. Each cell $(x_{1,i},\ldots,x_{n,i})$ represents an offset from the origin cell $(0,\ldots, 0)$. It is possible to use more than one Neighbors sentence to define the neighborhood.
- **Initialvalue** : [*Real* | **?**] defines the default initial value for each cell in the cell space. There are several ways of defining the initial values for each cell. The parameter **initialvalue** has the least precedence (i.e., if another parameter defines a new value for the cell, then that value will be used).
- **InitialCellsValue** : *filename* defines the name of a file containing a list of initial values for cells in the model. **InitialCellsValue** can be used with any size of cellular model, and it has higher precedence than **InitialRowValue**.
- **InitialMapValue** : *filename* defines the filename for the file that contains a map of values that will be used as the initial state for a cellular model.
- **Initialrowvalue** : integer [*Real* | **?**]* defines the values for the row whose number is first defined.
- **Delay : [TRANSPORT | INERTIAL]** specifies the delay type used for the cells.
- **DefaultDelayTime** : an integer value that defines the default delay (in milliseconds) for those inputs received from external DEVS models and for cells returning *undefined* values. In these cases, we need a default value for the delay function. In the case of using rules with three-valued logic (true/false/undefined) or random functions, we can obtain undefined

states, and we need to define the delays for those cases. If a **portInTransition** is specified, then this parameter will be ignored for that cell.

- **LocalTransition** *transitionFunctionName* defines the name of a group that contains the rules for the default local computing function.
- **PortInTransition** *portName@cellName* (x_1,...,x_n) *TransitionFunctionName* defines an alternate local transition function to be executed when an external event is transmitted to an individual through a specific port. By default, if this parameter is not used, when an external event is received by a cell, its value will be the future value of the cell with a delay as set by the **DefaultDelayTime** clause.
- **Zone** *transitionFunctionName* { range$_1$[..range$_n$] } defines a region of the cellular space that will use a different local computing function. A zone is defined by giving a set of single cells or cell ranges. A single cell is defined as (x_1,...,x_n) and a range as (x_1,...,x_n).. (y_1,...,y_n). All cells and cell ranges *must* be separated by a blank space. As an example, `zone : pothole { (10,10)..(13, 13) (1, 3) }` tells CD++ that the local transition rule *pothole* will be used for the cells in the range (10,10)..(13,13) and the single cell (1,3). The **zone** clause will override the transition defined by the **LocalTransition** clause, and it will use the rules defined within the *pothole* section.

Figure 4.13 illustrates the coupled model file definition for the popular *life* game [21]. In this model, each cell can be occupied by a living entity (value = 1), or it can be empty (a dead cell becomes empty, e.g., value = 0). A living cell remains alive only if it has three or four living neighbors. Otherwise, it dies. A cell becomes alive when there are exactly three living neighbors of an empty cell.

The Cell-DEVS coupled model in the figure is defined by its size (*width* = 20, *height* = 20), its border (*wrapped*, meaning that the cells in one border communicate its results to neighbors in the opposite border), the shape of the neighborhood (Moore's neighborhood), and the type of delay (*transport*). The rules defined by `life-rule` represent the behavior of each cell in the model. In this case, an active cell `((0,0) = 1)` remains active when the number of active neighbors is three

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1-1) life(1,0) life(1,1)
initialvalue : 0
initialvalue : 5    00000001110000000000
initialvalue : 7    00000100100100000000
initialvalue : 8    00000101110100000000
initialvalue : 9    00000100100100000000
initialvalue : 11   00000001110000000000
localtransition : life-rule

[life-rule]
rule : 1 100 { (0,0) = 1 and trueCount = 5 }
rule : 1 100 { (0,0) = 0 and trueCount = 3 }
rule : 0 100 { t }
```

FIGURE 4.13 Example for the definition of a Cell-DEVS life model.

or four (*truecount* indicates the number of active neighbors) using a transport delay of 100 ms. If the cell is inactive `((0,0) = 0)` and the neighborhood has three active cells, the cell becomes active. In every other case, the cell remains inactive (*t* indicates that whenever the rule is evaluated, a *true* value is returned).

Exercise 4.10

Run the Life model defined here (the original model can be found in *./life.zip* at the URL http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/). Change: (a) the initial values; (b) the size of the model; (c) the neighborhood shape (use a von Neumann neighborhood and a modified Moore's neighborhood); and (d) the rules of reproduction. Create a model, simulate it, and analyze the results obtained.

The complete language that defines the cell's behavior can be found in the CD++ user's manual, and we will give varied examples of its use throughout the book. It includes the basic logic operations (*AND*, *OR*, *NOT*, *XOR*, *IMP*, and *EQV*), comparison of real numbers (=, !=, <, >, <=, and >=), and the basic arithmetic operations applicable on them (+, –, *, and /). Different functions are available for real numbers (e.g., trigonometric, roots, power, rounding and truncation, modules, logarithms, absolute values).

Some functions return the number of cells in the neighborhood whose state has a given value. For example, *truecount* returns the quantity of cells whose state value is 1. Also available are the functions *falsecount*, *undefcount*, and *statecount(n)*. The last is the most generic and allows specifying the value (*n*) of the state to count. The remainder of the functions of this type could be defined as being based on *statecount*.

The language provides the use of predefined constants as π (*pi*) and *e*, together with certain constants of frequent use in the domains of physics and the chemistry (gravitational constant, acceleration, light speed, Planck's constant). The constant *INF* represents the infinite value. This constant is returned automatically when the evaluation of a numeric expression produces a numeric overflow. The *Time* function permits obtaining the global time of simulation expressed in milliseconds. Likewise, there are functions for unit conversion. The functions *RadToDeg* and *DegToRad* are used for the conversion of angles expressed in radians to degrees and vice versa, respectively. There are functions for the conversion of polar and rectangular coordinates and temperatures in Celsius, Fahrenheit, or Kelvin degrees.

Different functions are provided to generate pseudorandom numbers using different probability distributions, including uniform, χ^2, β, exponential, F, gamma, normal, binomial, and Poisson. The introduction of random results in the definition of the condition of a rule introduces other problems. For example, in the rule:

```
10 100 { random >= 0.4 }
```

the condition is evaluated to *true* in about 60% of the cases and to *false* on the rest. Therefore, the model could return all the rules evaluated to *false*. In these cases, CD++ assigns the *undefined* value to the cell and uses the default delay time informing this situation and continuing with the simulation. CD++ uses three-valued logic, and expression and function can use the *undefined* value (represented as "?").

Other functions allow obtaining values depending on the evaluation of a certain condition. *IFU*(*c*, *t*, *f*, *u*) evaluates the *c* condition, and, if it is *true*, it returns the *t* value. If it is *false*, it returns *f*, and *u* in the case that is *undefined*. On the other hand, the function *IF*(*c*, *t*, *f*) returns *t* if *c* evaluates to *true*, and *f* otherwise. Other functions allow checking if a number is an integer, if it is even or odd, if it is a prime number, or if it is undefined.

Most existing real systems are studied using models in two or three dimensions. Nevertheless, several theoretical problems can be defined as cellular models with higher dimensions. CD++ supports

defining *n*-dimensional cell spaces. It also supports two boundary conditions for each cell space: they can be *wrapped* (opposite borders are connected) or *nonwrapped* (a fixed boundary is defined).

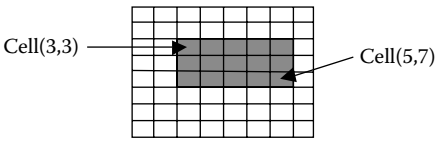


FIGURE 4.14 A zone defined by cell range { (3,3)..(5,7) }

Different *zones* can be defined for the cell space. Each zone is defined by a cell range $\{(x_1, x_2,\ldots,x_n)..(y_1, y_2,\ldots,y_n)\}$. Each zone is associated with a set of rules different from those in the rest of the cell space. This allows having different zones in the same cellular model, with a special behavior for each. Hence, a zone defined by a range of cells is defined by the set of cells (t_1, t_2,…,t_n) of the cell space, such that $t_i \in [\min(x_i, y_i),\max(x_i, y_i)]\ \forall i \in [1, n]$ (Figure 4.14).

When a cell is created, two ports are automatically associated: *NeighborChange* and *Out*. *NeighborChange* defines the input values arriving from the neighbors; *Out* connects the cell with the neighbors and other DEVS models. When an external message arrives at the cell through the *in* port, its value is queued, and it will be used to compute the new cell state. Input ports *in* are created only for those cells connected with external DEVS models. On the other hand, the local computing function τ uses all the inputs, including the values sent by the neighboring cells and the external messages that have arrived through the other input ports (Figure 4.15).

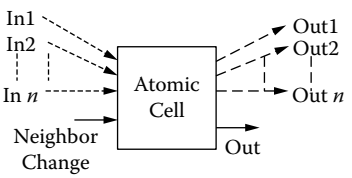


FIGURE 4.15 Structure of an atomic cell.

The *out* port connects each cell with the neighbors and with other DEVS models. The *out* port can also be used to connect a cell with other DEVS models that will receive a new message each time the cell changes its internal state. The rest of the output ports are created dynamically only for the cells that will send state values for other DEVS models, and their names can be defined by the modeler (this will be discussed in later chapters).

As discussed earlier in this chapter, in many cases, rectangular topologies are not enough for defining the behavior of advanced cell spaces. Triangular meshes allow covering of areas with more varied topology, while permitting every cell to have a limited number of nearby neighbors. Hexagonal geometries have higher isotropy—that is, the capacity to represent equivalent behavior in every possible direction (which is not the case for square meshes). This is more natural for building the model's rules. CD++ provides a lattice translator (*Ltrans*) to define the cells' behavior based on these topologies; this translates hexagonal or triangular rules to square CD++ compatible rules. *Ltrans* translates hexagonal or triangular rules to square CD++ compatible rules. *Ltrans* receives a set of rules based on a hexagonal or triangular geometry and translates it into rules based on square geometry to be included in a model to be simulated with CD++. This idea was proposed at http://www.tu-bs.de/institute/WiR/weimar/Zascrptnew/geometry.html, and it is based on using a function that shifts alternate rows in opposite directions, as shown in Figure 4.16. The function maintains the boundary conditions in the square lattice.

Let (*x*,*y*) be the position of a cell, where *x* represents the row and *y* represents the column (remember that the function can be applied only in two-dimensional spaces). The neighborhood relation

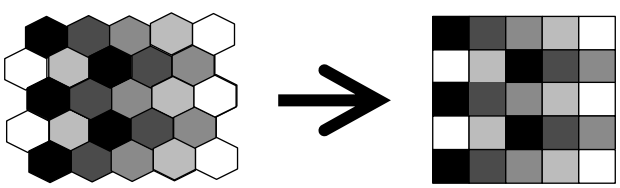


FIGURE 4.16 Shift mapping of the hexagonal lattice to the square lattice.

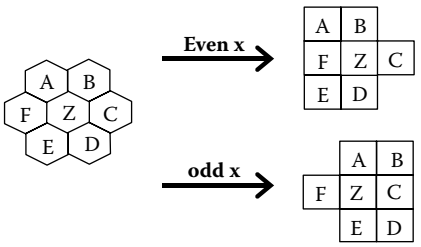


FIGURE 4.17 Neighborhood relation in hexagonal to square mapping function.

is transformed differently, depending on whether the row index x is even or odd, as shown in Figure 4.17.

The mapping of the triangular lattice to the square lattice is similar to the shift mapping for the hexagonal lattice. In the triangular case, every second cell has a different orientation. The mapping function is shown in Figure 4.18. Each row of triangles is mapped to one row of squares, depending on the parity of $x + y$. The nearest neighborhood mapping is shown in Figure 4.19.

The language used to model a cell's behavior in a hexagonal or triangular geometry is the same as that used in CD++; the only difference is the way in which a neighbor is referenced. In CD++, a cell (belonging to a two-dimensional space) is referenced using a pair (x,y), where x (row) and y (col) are the relative positions of the cell. *Ltrans* only supports nearest neighbors, so it was necessary to define nearest neighbors for hexagonal and triangular geometry. For both geometries, each nearest neighbor is referenced using $[n]$, where n is the number assigned to each nearest neighbor (Figure 4.20).

This translated file consists of the set of rules that can be simulated with CD++. Before simulating the model, it must be completed with the other parameters that define a Cell-DEVS model (space dimension, type of border, default delay, etc.). Besides the nearest neighbor translation (done as in Figures 4.17 and 4.19), we need to consider the functions that evaluate the values incoming from the

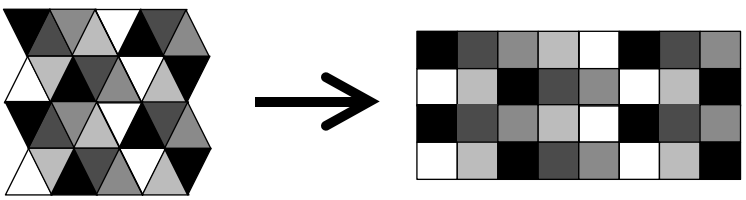


FIGURE 4.18 Visualization mapping of the triangular lattice to the square lattice.

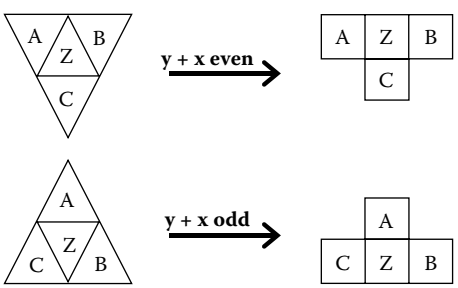


FIGURE 4.19 Nearest neighbors in the triangular mapping.

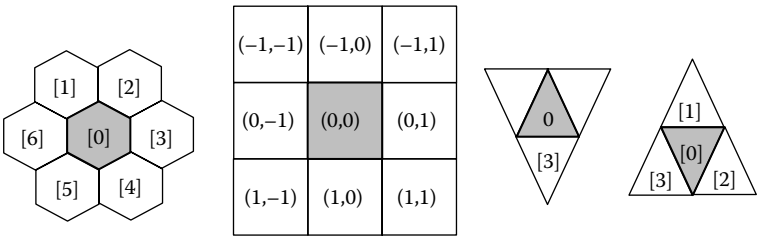


FIGURE 4.20 (a) Nearest neighbors used for hexagonal geometry; (b) nearest neighbors used for square geometry; (c) and (d) nearest neighbors used for triangular geometry.

```
rule : 1 100 { [0] = 1 and truecount = 5 }
```
(a)

```
rule : 1 100 {even(cellpos(0)) and (0,0) = 1 and (truecount-(-1,1)-(1,1)) = 5}
rule : 1 100 {odd(cellpos(0) and (0,0) = 1 and (truecount-(-1,-1)-(1,-1)) = 5)}
```
(b)

FIGURE 4.21 Translating hexagonal rules: (a) hexagonal lattice rules; (b) square lattice rules.

```
rule : 1 100 { [2] = 0 and truecount = 3 }
```
(a)

```
rule : 1 100 { (0,-1) = 0 and odd(cellpos(0)+cellpos(1)) and
                    (truecount-(-1,-1)-(-1,1)- (1,-1)-(1,0)-(1,1) ) = 3}
```
(b)

FIGURE 4.22 Translating triangular rules: (a) triangular lattice rules; (b) square lattice rules.

neighbors. For instance, Figure 4.21 shows the translation of different rules in a hexagonal lattice to the corresponding square notation.

As we can see, when we translate the *truecount* function, we have to delete the extra cells in the new topology. A square lattice includes nine immediate neighbors, but the hexagonal version includes only seven; thus, we delete the values of the corresponding two missing cells according to the translation proposed in Figure 4.17. We need to add two rules: one for the case of an even row and another for the case of odd rows. The translation is similar for triangular meshes, as we can see in Figure 4.22. In this case, we have to consider the position of both rows and columns, and we delete the ones that do not need to be included in the count (in this case, we use only five of the nine near neighbors).

4.6 DEFINING ATOMIC MODELS USING DEVS-GRAPHS

Defining models in C++ allows most users to have great flexibility in defining the model's behavior. Nevertheless, nonexperienced programmers can have difficulties in defining models using this approach. Using a graphical specification enhances the interaction with stakeholders during system specification because graphical notations have the advantage of allowing the modeler to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation [22,23] that allows defining of the behavior of atomic models based on DEVS-graphs [24]. Each DEVS-graph defines the state changes according to internal and external transition functions, and each is translated into an analytical definition.

DEVS-graphs can be formally defined as

$$DEVS\text{-}graph = < X_M, S, Y_M, \delta_{int}, \delta_{ext}, \lambda, D > \quad (4.1)$$

where

X_M = {(p,v)| p ∈ *IPorts*, $v \in X_p$ } is a set of input ports;
Y_M = {(p,v)| p ∈ *OPorts*, $v \in Y_p$ } is a set of output ports;
S = $B \times P(V)$ are states of the model;
B = { b | b ∈ *Bubbles* } is a set of model states;
V = { (v,n) | v ∈ *Variables*, $n \in R_0$ } are intermediate state variables of the model and their values; and
δ_{int}, δ_{ext}, λ, and D have the same meaning as in traditional DEVS models.

Each DEVS model has a unique identifier that will be used subsequently and the model uses a graph-based specification representing state changes for an atomic model. These states are represented by bubbles, including an identifier and the state lifetime. This specification allows defining of the pair (state, duration) associated with internal transition functions. When the lifetime is consumed the model will change its state by executing an internal transition function. For instance, Figure 4.23 shows a state called *start*, whose duration is 15 time units.

DEVS-graphs need an equivalent textual specification that can be used for computation [22]. The models are identified as

```
[modelname]
```

which defines the name of the atomic or coupled model name, which will be used subsequently.

States have identifiers and they are associated with a time advance value, as follows:

```
state : stateId …
stateId : lifetime
initial: statename
```

The `state` construction declares all the state identifiers that will be used for an atomic model, and the *lifetime* of each of the state IDs is then assigned to the corresponding identifier in a separate statement. One of the states must be declared as the *initial* state of the model.

Internal transition functions are represented by dashed arrows connecting a source and a destination state. Each of them can be associated with pairs of ports with values (q, v) corresponding to the output function. The syntax for the output function values is $q!v$. For instance, Figure 4.23 represents an internal transition that will make the model change from state `src` to state `dest`. Before that, the output function will send the value 8 through the port `q1` and 12 through the port `q2`.

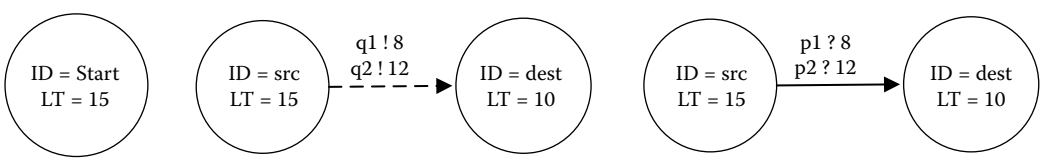


FIGURE 4.23 DEVS-graph notation.

The textual representation for internal transitions uses the following syntax:

```
int : source destination [q!v]* ( { (action;)* } )
```

Here, the keyword **int** is used to define the internal transition; we indicate the origin and destination states and a port list with the corresponding values. The output event is denoted as `q!v` (i.e., sending value v through port q), and there may be multiple outputs sent before each internal transition. For instance, the model on Figure 4.23 can be defined as

```
int : src dest q1!8 q2!12
```

Here, the source and destination represent the initial and final states associated with the execution of the transition function. Because the output function should also execute before the internal transition, an output value can be associated with the internal transition.

External transition functions are represented graphically by a full arrow connecting two states. The notation used to represent input ports and the values expected through them, (p,v), is `p?v` $[t_i..t_f]$. Here, $t_i..t_f$ represent the initial and final expected simulated times for the external transitions. If the triggering external events arrive at a time outside the designated range, an exception will be raised during the simulation, bringing the error to the user's attention. This built-in self-checking mechanism allows users to specify temporal constraints within the model definition, thus improving the safety of the model and facilitating the verification process.

The textual notation for external transitions is as follows:

```
ext : source destination EXPRESSION([p?v]*) { (action;)* } [ti..tf]
```

It describes the origin and destination states, an input port, and a time range counted since the instant of arriving at the start state. The external transition happens only if the required triggering condition is "held," which is specified using a logical expression that involves predefined functions and the input events. In this case, the model will change from state *source* to state *destination*, while also executing one or more actions.

DEVS atomic models interact with others through input and output ports. In DEVS-graphs we represent them as arrowheads attached to a model definition, as we can see in Figure 4.24. In the textual specification, ports are described by including their name and a type, based on the formal specification for DEVS models. They are defined as

```
in : portId:type portId:type …
out : portId:type portId:type …
```

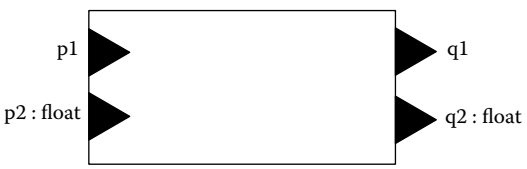


FIGURE 4.24 Graphical and textual notation of input and output ports.

In our example, port `p2` only accepts input data of type *float*. The default data type of a port is an integer (e.g., both input port `p1` and output port `q1` have an integer data type).

Our textual representation also permits defining of temporary variables, which are declared as

```
var : var1 var2 var3 …
```

and they can be optionally initialized as

```
var1 : value1
var2 : value2
```

Each internal and external transition may optionally have a list of actions to manipulate these temporary variables. Actions can be specified as simple mathematical expressions, or they can be implemented in user-defined C++ functions, providing a flexible mechanism for defining complex model behavior. The following table lists some basic arithmetic and Boolean functions provided for defining the actions:

Function	Description
`Add(n1, n2)`	Sum of n1 and n2
`And(n1, n2)`	Return true if both n1 and n2 are true
`Any(port)`	Return true if the port has a valid value
`Between(n1, n2, n3)`	Return true if n1 ≤ n2 ≤ n3
`Compare(n1, n2, n3, n4, n5)`	Return n3, n4, or n5 if n1 is greater than, equal to, or less than n2
`Divide(n1, n2)`	n1/n2
`Equal(n1, n2)`	Return true if n1 = n2; otherwise, return false
`Greater(n1, n2)`	Return true if n1 > n2; otherwise, return false
`Less(n1, n2)`	Return true if n1 < n2; otherwise, return false
`Minus(n1, n2)`	n1 – n2
`Multiply(n1, n2)`	n1 * n2
`Not(n)`	Return the negation of n
`NotEqual(n1, n2)`	Return true if n1 != n2
`Or(n1, n2)`	Return true if n1 or n2 is true
`Pow(n1, n2)`	Return n1 power n2
`Rand(n1, n2)`	Generate a random value between n1 and n2
`Value(n)`	Return the value of n

All these constructions can be combined to define the behavior of atomic models. For instance, Figure 4.25 represents a simple model using all the constructions. In this case, the model will remain in the *start* state for four time units. If we receive an input before that (in particular, between time [1..3]) and the input has a value of 4, the state changes to *process* (for 10 time units). When this time is consumed, the model executes the output function (transmitting the value 1 through port *out*) and changes to the *finish* state for 7 time units. Here, two things can happen: If the time is consumed, the model issues an output (value 6 through the *out* port); if an input of 2 is received on port *in* (between times [2..5]), we return to the *process* state. This model is equivalent to the following specification:

$$\text{Simple\_Proc} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D > \tag{4.2}$$

where

X = { (in, *Z*) };
Y = { (out, *Z*) };
S = { start, process, finish };

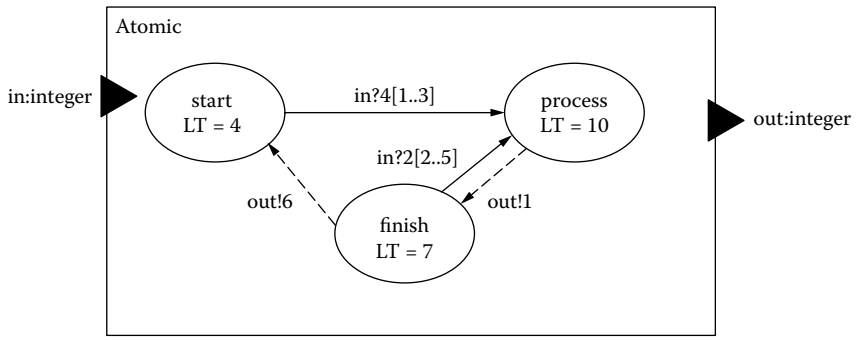


FIGURE 4.25 Definition of an atomic model.

δ_{ext}(s,e,x):

```
case port (in) {
        4:      if (e < 1 or e > 3) error();
                if (state == start)
                state == process;
                ta(state) = 10;
        2:      if (e < 2 or e > 5) error();
                  if (state == finish)
                        state = process;
                        ta(state) = 10;        }
```

λ(s):

```
case (state) {
        finish: send(out, 6);
        process: send(out, 1);       }
```

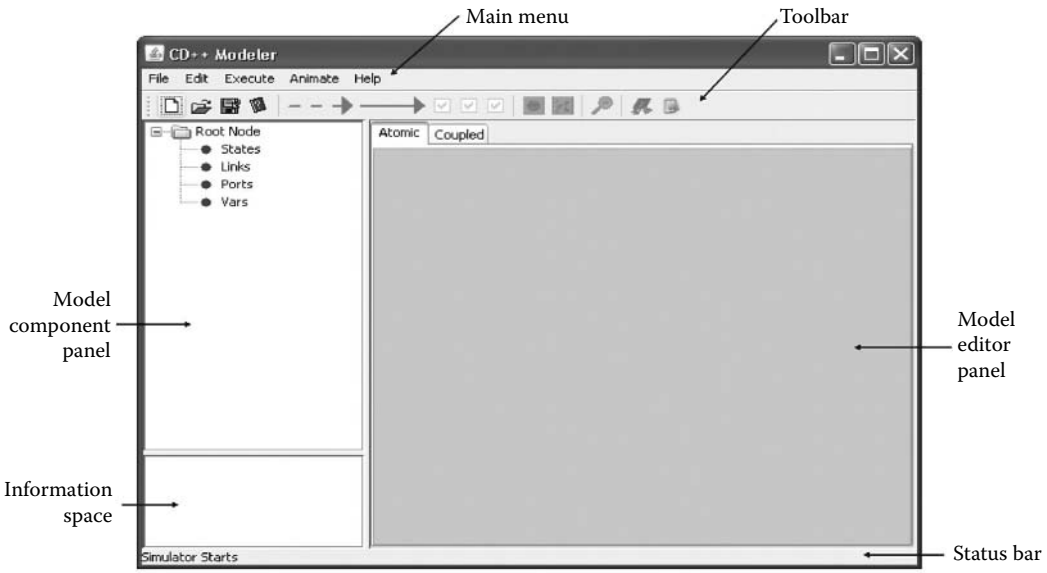
δ_{int}(s):

```
case (state):
        finish: passivate();
        process: state = finish; ta(state) = 7;
```

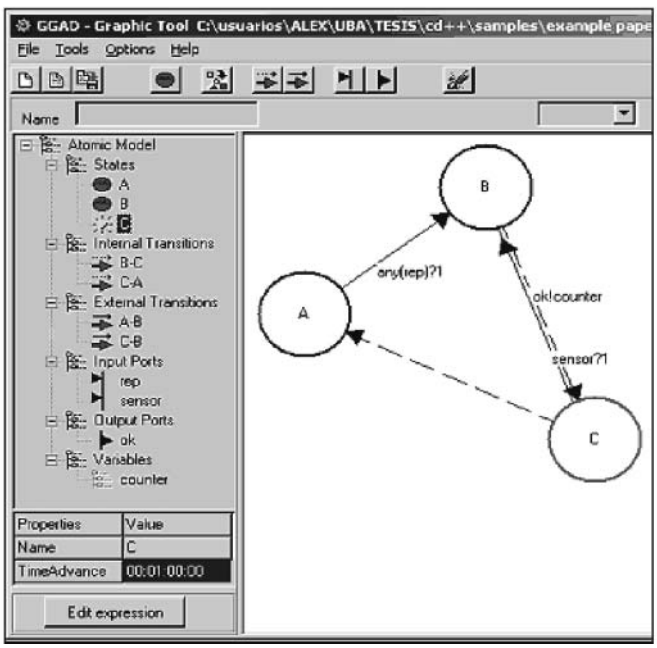
CD++Modeler [25] is a graphical user interface included with the tool that allows the user to define DEVS models using this graphical notation. There are two versions available: one that is Java based (i.e., platform independent) and the GGAD interface (a simpler version running on MS Windows-based environments). The tools are in prototype version, and they can be downloaded for further modifications and enhancements at http://sourceforge.net/projects/cdplusplus/.

CD++Modeler provides a graphical user interface (GUI) that allows users to construct DEVS atomic and coupled models graphically in a simple drag-and-drop fashion. The basic functionality of the two interfaces is similar: atomic and coupled models can be generated graphically (using the notation described in this section for atomic models), input/output ports can be defined, and other services are included—for instance, two-dimensional visualization that presents an intuitive animation of the simulation results to the user. Figure 4.26 shows the general view of both interfaces (detailed use of the interfaces can be found in the user manual).

The main components are a model editor, in which the user can define atomic/coupled models, and a text-based editor that can be used to update the names of the ports, define initial variable values, and activate specialized functions. The GUI has two major panels. On the left-hand side is the model component panel that shows a tree of units defined in the current model (i.e., the states, ports, variables, and transitions of an atomic model or the ingredient components of a coupled model) for quick navigation and access. Once a unit is selected, its attributes are displayed in the information



(a)



(b)

FIGURE 4.26 (a) CD++Modeler; (b) DEVS-graph graphical editor. (From Christen, G. et al. 2004. *Proceedings of MGA, Advanced Simulation Technologies Conference 2004,* Arlington, VA.)

space underneath. On the right-hand side is the model editor panel that provides the workspace where users can choose to build an atomic or a coupled model by selecting the corresponding tabs.

CD++Modeler provides a graphical model editor for defining each type of model (atomic or coupled) using the DEVS-graph notation. The graphical specifications of atomic and coupled models can be later imported as templates (or classes) in construction of other coupled models. In addition, many instances with different properties can be created from an existing model template. A user can

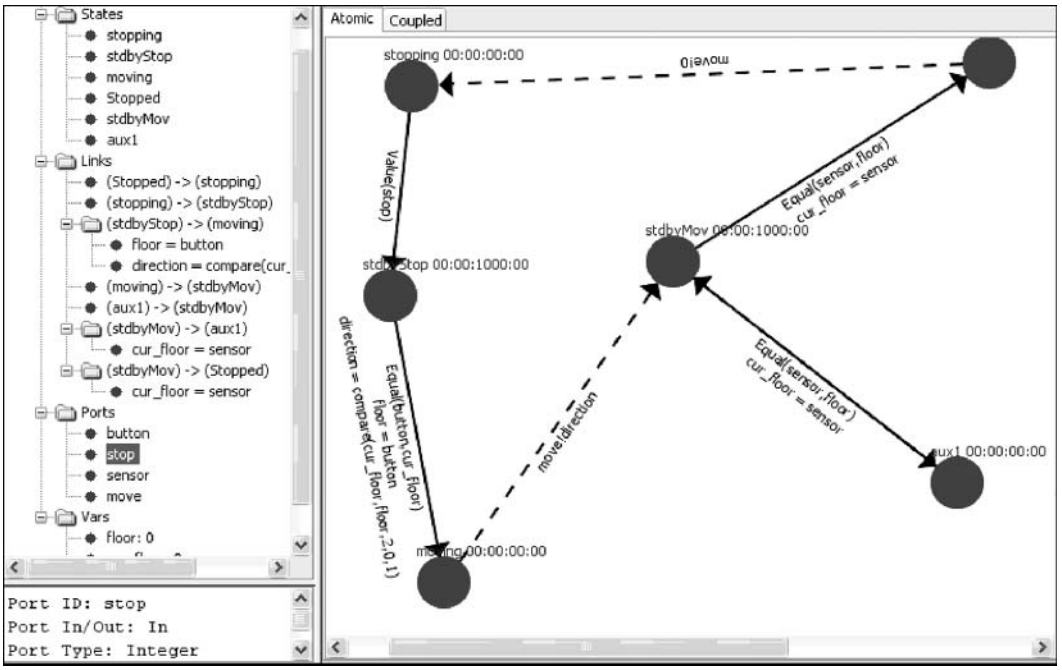


FIGURE 4.27 Definition of an elevator controller in CD++Modeler.

then define the top most coupled model, adding new atomic or coupled components as needed. The GUI allows users to open a component in the corresponding editor to define or modify the model.

Figure 4.26(b) shows a simple atomic model using the notation defined in this section and CD++Modeler. The model includes three states: A, B, and C. In this case, if the model is in state A and it receives an external event through the *rep* input port (shown in the left panel), the function *any* is evaluated. If the result of this evaluation is 1, the model changes to the state B. While in state B, the model waits the time defined by the time advance function to be consumed. It then executes the output function, which will send the value of the intermediate state variable *counter* through the output port *ok*. After that, the internal transition function executes, and the model changes to the state C. (As we can see on the left part of the figure, the time advance function for state C is 1 min, and there is no output associated with the internal transition.)

Figure 4.27 shows a more complex model, representing a simple controller for an elevator. As we can see, we have a graphical pane on the right and a text-based definition for the graph (including states, links, and ports) on the left. The left pane also includes information about intermediate variables, and when we choose any of the elements in the graph, extra information is shown at the bottom of the pane. (In this case, we can see that the *stop* port is an input port, and it receives integer values.)

Figure 4.28 defines the model's specification in the text format that CD++ is able to execute. On the textual specification, we identify six states: `stopping`, `stdbyStop`, `moving`, `Stopped`, `stdbyMov`, and `aux1`. The initial state is `stdbyStop`. We use three auxiliary variables (not part of the state): `floor` (represents the floor where the elevator must move to), `cur _ floor` (stores the current floor), and `direction` (representing whether the elevator must go up, go down, or stay in the same floor).

If an external event is received while in the `stdbyStop` state (which represents the elevator on standby stopped state), we evaluate the function `Equal(button, cur _ floor) ? 0 {floor = button; direction = compare(cur _ floor,floor,2,0,1); }`, which represents the behavior under a button being pressed. In this particular example, we compare the floor of the button just pressed (`button`) with the current floor in which the elevator is on standby

```
[controller]
state: stopping stdbyStop moving Stopped stdbyMov aux1
initial : stdbyStop
in: button stop sensor
out: move
var: floor cur_floor direction
ext: stdbyStop moving Equal(button,cur_floor)?0 {floor = button;direction =
compare(cur_floor,floor,2,0,1);}
ext: stdbyMov aux1 Equal(sensor,floor)?0 {cur_floor = sensor;}
ext: stdbyMov Stopped Equal(sensor,floor)?1 {cur_floor = sensor;}
ext: stopping stdbyStop Value(stop)?1
int: moving stdbyMov move!direction
int: aux1 stdbyMov
int: Stopped stopping move!0

stopping:00:00:00:00
stdbyStop:00:00:1000:00
moving:00:00:00:00
Stopped:00:00:00:00
stdbyMov:00:00:1000:00
aux1:00:00:00:00

floor:0
cur_floor:0
direction:0
```

FIGURE 4.28 Text specification of the elevator controller in CD++Modeler.

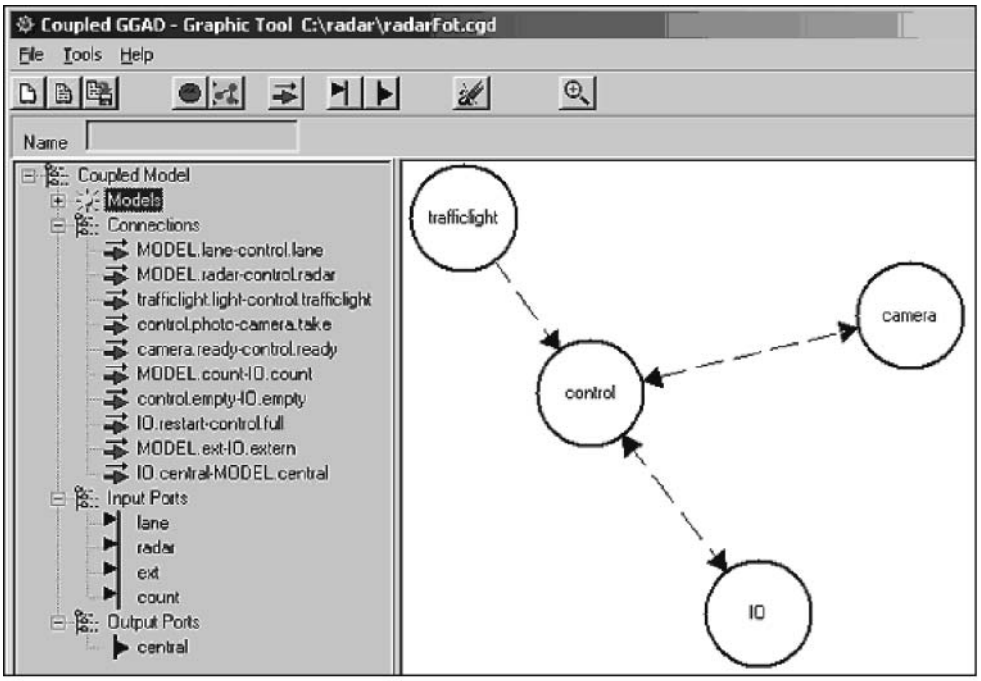
(`cur _ floor`). If they are equal (i.e., the result of the comparison is 0), then the current floor is set to `button` (i.e., the destination `floor` is the one in which the button has been pressed). If not, this means the button was pressed on a higher or lower floor. In order to determine which, we compare the current floor with the destination floor, and we make the direction `up`, `down`, or `none` according to the comparison result. This value is set to the variable `direction`. Finally, we change to the state `moving`. As we can see, this state is associated with an instantaneous internal transition (ta(s) = 0), which will generate an output informing the current direction through the `move` output port and will change the state to `stdbyMov`.

From this state, there are two options. If an external input is received (from the sensors on each floor while the elevator moves), we compare the sensor with the destination floor position. If they are not equal, we change to an auxiliary state that represents that we are still moving. This state will trigger an instantaneous internal transition, which will make the model go again to the `stdbyMov` state. This cycle is repeated on every sensor, until the moment when we arrive at the destination floor. In each of these steps, `cur _ floor = sensor` in order to represent storing the information about the floor sensor sensed. When we arrive at the floor, the external transition function makes the model go to the `stopped` state. At this point, we issue an instantaneous internal transition that will transmit a value of 0 through the `move` output port (which means the elevator motor has stopped), and we switch to the `stopping` state, which is used to represent the delay taken because the motor stops until the elevator actually stops. We will only return to the initial state `stopping` after receiving a signal through the `stop` input port (which will indicate when the elevator has fully stopped).

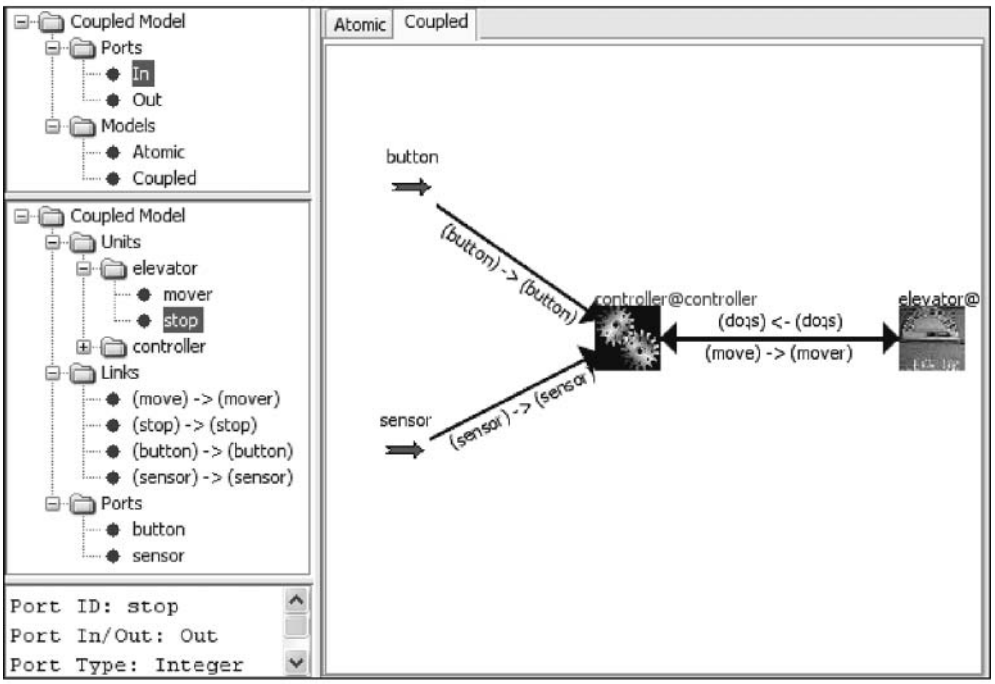
Exercise 4.11

Modify the elevator model found in *./elevatorModeler.zip*, and change (a) the duration of the different transitions and (b) the output functions. Generate a new textual model and analyze the results.

Once the user defines atomic models, they can be integrated into a coupled model using a graphical notation, as described in Figure 4.29. We can see the two versions of the graphical interfaces available.



(a)



(b)

FIGURE 4.29 Coupled models: (a) photographic radar; (b) elevator.

Figure 4.30 defines the coupled model's specification in the text format that CD++ is able to execute. As we can see, we have two components (atomic models defined using the DEVS-graph notation used earlier) connected through the ports *stop* and *move* on each of them. Simultaneously, we can

```
[top]
components : elevator@GGAD controller@GGAD
in : button sensor
Link : move@controller move@elevator
Link : stop@elevator stop@controller
Link : button button@controller
Link : sensor sensor@controller

[elevator]
source : elevator.CDD

[controller]
source : controller.cdd
```

FIGURE 4.30 Structure of the photographic radar model.

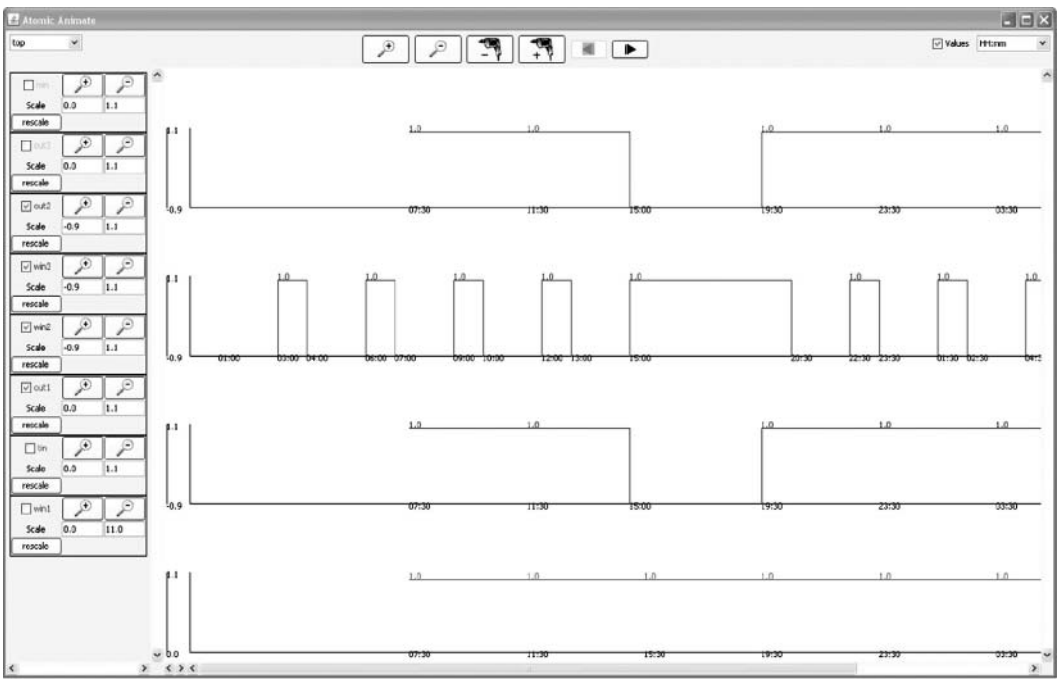


FIGURE 4.31 Animation of the input and output trajectories of a DEVS model in CD++Modeler.

receive external inputs through the *button* port (to simulate buttons being pressed) or *sensor* (to simulate the motor moving through different floors and a sensor checking the arrival at a given floor).

CD++Modeler can be used to visualize the simulation results of DEVS and Cell-DEVS models by parsing the simulation log generated during the simulation. DEVS models can be visualized with the *atomic animation* option, which involves plotting the input and output trajectories of a model component (atomic or coupled). Figure 4.31 shows such a facility.

Figure 4.31 illustrates the animation of an example DEVS coupled model. Users can adjust the horizontal and vertical scaling of the trajectories, change the time format, and focus on specific inputs and outputs by selecting the checkboxes on the control panel. If the model definition file contains multiple components, users can quickly switch the animation between them. This oscillogram-like animation allows users to investigate model behavior at the I/O functional level and establishes a straightforward causal relationship between the inputs and outputs, facilitating the model validation process.

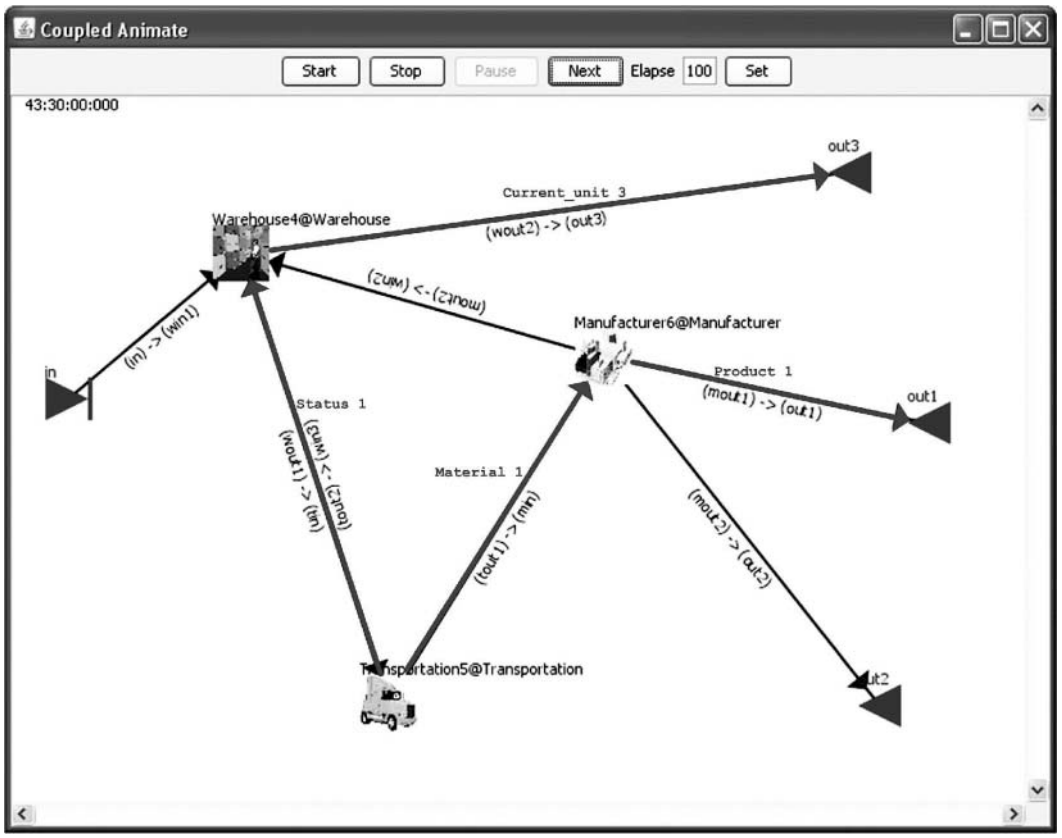


FIGURE 4.32 Animation of the interactions between components of a DEVS coupled model in CD++Modeler.

The *coupled animation* facility shows the interactions between the ingredient components of a DEVS coupled model, shown in Figure 4.32. The animation is overlaid on top of the DEVS graph specification of a coupled model, displaying messages passing along the coupling links with two-dimensional text effects. For example, the animation in Figure 4.32 shows that a message carrying the value of variable *material* is sent from the *Transportation* component to the *Manufacturer* component at simulated time `43:30:00:000`. This animation gives a high-level view of the interior behavior of a coupled model, helping users interpret and reconstruct what is happening in the simulation.

CD++Modeler also provides a GUI to animate cell spaces based on customized coloring schemes. Figure 4.33 illustrates the animation of a two-dimensional Cell-DEVS model that simulates the propagation of forest wildfires. The control panel provides the means to load multiple Cell-DEVS models, change the frame duration, run or pause the animation, and single-step (forward or backward) through the animation sequence.

Although square geometries are widely used to define cell spaces (each cell is represented as a square object), we have discussed that triangular- or hexagonal-shaped cells, may enable more appropriate and natural model definitions. CD++Modeler provides a lattice translator that is able to perform automated mapping between different geometries in visualizing Cell-DEVS models. Figure 4.34 shows the animation of the fire propagation model using triangular- and hexagonal-shaped cells.

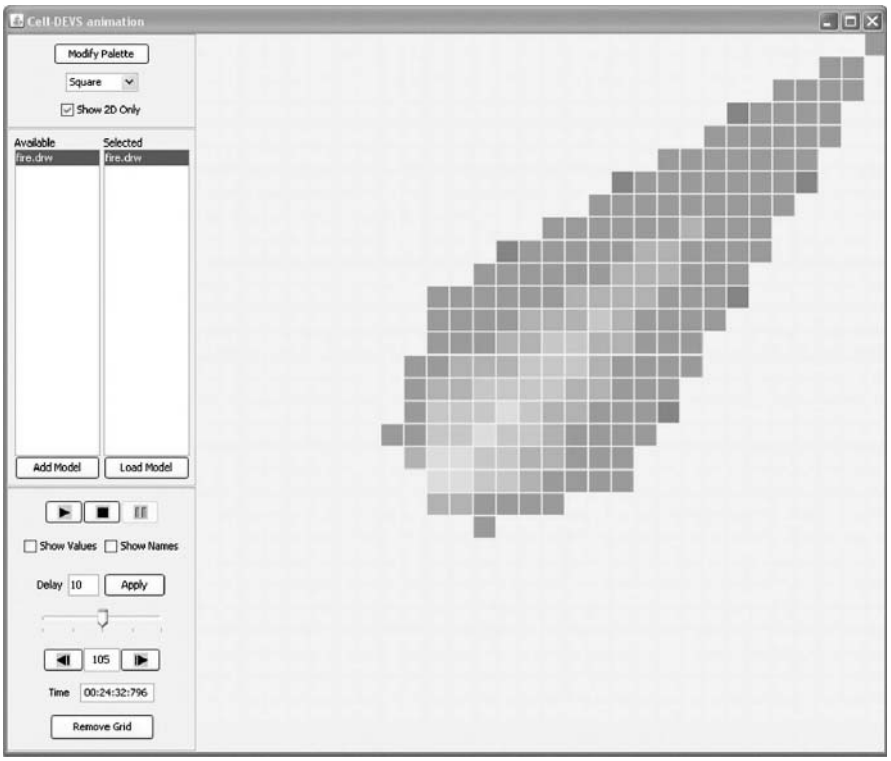


FIGURE 4.33 Animation of two-dimensional Cell-DEVS model using user-specified color palette.

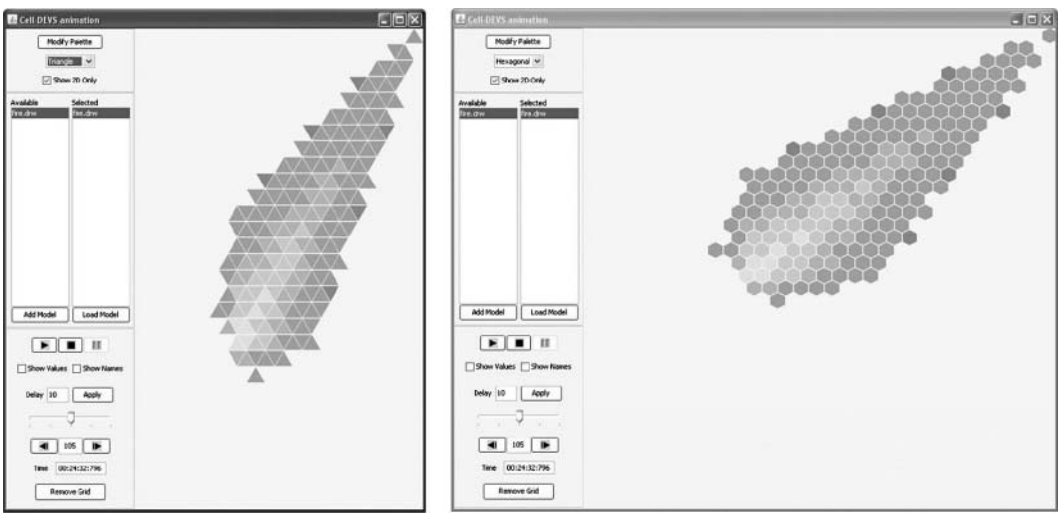


FIGURE 4.34 Animation of two-dimensional Cell-DEVS model using different geometries.

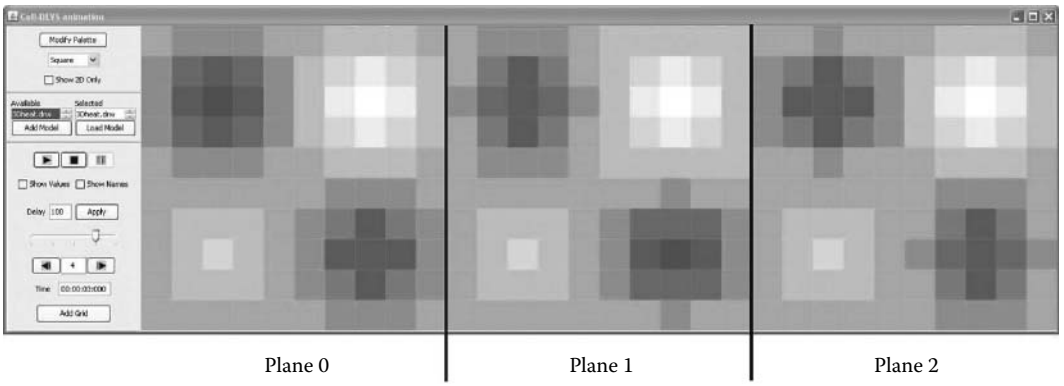


FIGURE 4.35 Animation of three-dimensional Cell-DEVS models as individual planes.

CD++Modeler visualizes high-dimensional cell spaces as a series of individual planes. Figure 4.35 shows the animation of a three-dimensional heat diffusion model ($10 \times 10 \times 3$) as three planes (advanced three-dimensional visualization techniques will be discussed in Chapter 16).

4.7 SUMMARY

In this chapter, we have introduced the basic aspects of the CD++ modeling and simulation toolkit. CD++ follows DEVS and Cell-DEVS specifications. DEVS atomic models can be written using C++ or a state-based notation associated with a graphical user interface.

DEVS coupled models can be defined using a built-in specification language that follows DEVS formal specifications for coupled models. Cell-DEVS models are also defined using a specialized language in which the user can define individual rules for the model and the specific coupled model specifications.

Finally, we showed the basic aspects of CD++ graphical user interfaces.

REFERENCES

1. Nutaro, J. *ADEVS.* URL: http://www.ornl.gov/~1qn/adevs/index.html. Accessed: June 1, 2007.
2. Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1996. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS* 13 (3): 135–154.
3. Zeigler, B., Y. Moon, D. Kim, and D. Kim. 1996. DEVS-C++: A high performance modeling and simulation environment. *Proceedings of 29th Hawaii International Conference on System Sciences,* Honolulu.
4. Zeigler, B. P. 1990. *Object-oriented simulation with hierarchical, modular models: Intelligent agents and endomorphic systems.* Boston: Academic Press.
5. Zeigler, B., and D. Kim. 1995. Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control. Technical report, Department of Electrical and Computer Engineering, University of Arizona.
6. Sarjoughian, H. S., and B. P. Zeigler. 2000. DEVS and HLA: Complementary paradigms for M&S? *Transactions of the SCS* 17:187–197.
7. Zeigler, B. P. 1999. Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions. *Simulation Interoperability Workshop,* Orlando, FL.
8. IEEE Std 1516.1-2000. 2001. IEEE standard for modeling and simulation. High level architecture (HLA)—Federate interface specification. *IEEE Std 1516.1-2000*: i–467.
9. Sarjoughian, H. S., and B. P. Zeigler. 1998. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. *Proceedings of SCS International Conference on Web-Based Modeling and Simulation,* San Diego, CA.
10. Kim, T. G. 1994. *DEVSIM++ user's manual.* CORE Lab, EE Dept, KAIST, Taejon, Korea.

11. Dávila, J., and M. Uzcágegui. 2000. GALATEA: A multi-agent, simulation platform. *Proceedings of International Conference on Modeling, Simulation and Neural Networks,* Mérida, Venezuela.
12. Himmelspach, J., and A. Uhrmacher. 2004. A component-based simulation layer for JAMES. *Proceedings of 18th Workshop on Parallel and Distributed Simulation (PADS),* Kufstein, Austria, 115–122.
13. Filippi, J. B., and P. Bisgambiglia. 2004. JDEVS: An implementation of a DEVS based formal framework. *Environmental Modeling and Software* 19:261–274.
14. Bolduc, J. S., and H. Vangheluwe. 2001. The modeling and simulation package PythonDEVS for classical hierarchical DEVS. Technical report MSDL-TR-2001-01, McGill University.
15. de Lara, J., and H. Vangheluwe. 2002. AToM3: A tool for multi-formalism and meta-modeling. *Proceedings of Fundamental Approaches to Software Engineering, 5th International; Lecture Notes in Computer Science,* 174–188.
16. Praehofer, H., and G. Reisinger. 1995. Object-oriented realization of a parallel discrete event simulator. Technical report, Johannes Kepler University, Department of System Theory and Information Engineering.
17. Henning, M., and S. Vinoski. 1999. *Advanced CORBA programming with C++*. Reading, MA: Addison–Wesley.
18. Dongarra, J. J. 1995. *High performance computing: Technology, methods and applications.* Amsterdam: Elsevier.
19. Wainer, G. 2002. CD++: A toolkit to develop DEVS models. *Software Practice and Experience* 32:261.
20. Plauger, P. J., A. Stepanov, M. Lee, and D. Musser. 2000. *The C++ standard template library.* Upper Saddle River, NJ: Prentice Hall.
21. Gardner, M. 1970. The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American* 23:120–123.
22. Christen, G., A. Dobniewski, and G. Wainer. 2004. Modeling state-based DEVS models CD++. *Proceedings of MGA, Advanced Simulation Technologies Conference 2004,* Arlington, VA.
23. Christen, G., A. Dobniewski, and G. Wainer. 2001. Defining DEVS models with the CD++ toolkit. *Proceedings of European Simulation Symposium,* Marseilles, France.
24. Praehofer, H., and D. Pree. 1993. Visual modeling of DEVS-based multiformalism systems based on higraphs. *Proceedings of WSC '93, the 25th Winter Simulation Conference,* Los Angeles, CA, 595–603.
25. Kidisyuk, K., and G. Wainer. 2007. CD++Modeler: A graphical viewer for DEVS models. Technical report SCE-017, Ottawa, ON, Canada.

5 Modeling Simple DEVS and Cell-DEVS Models in CD++

5.1 INTRODUCTION

In this chapter, we introduce some basic examples of DEVS and Cell-DEVS models and their implementation in CD++, which will provide a basis for better understanding of the varied models introduced in the following chapters. All the examples are simplified versions (for better understanding) of the ones located in the central model repository (http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/), which are ready to be used in CD++.

5.2 BASIC CELL-DEVS MODELS

Our first model is a maze-solving algorithm defined in Nayfeh [1]. The maze is represented as a two-dimensional Cell-DEVS model using value 1 to represent walls and value 0 to define hallways (free cells), using von Neumann's neighborhood. The maze is solved using the following rules for updating the cell's states:

- Wall cells remain unchanged.
- Free cells become wall cells if their neighborhood includes three or more wall cells.
- Free cells remain free if their neighborhood includes fewer than three wall cells.

When this set of rules is processed, the algorithm effectively blocks off every dead-end path in the maze. Every free cell that is accessible from only one direction (i.e., three wall cells around it) must be a dead end and therefore cannot be part of the solution. These cells become new wall cells, and this procedure is repeated until we obtain a steady state, in which the remaining free cells represent the solution(s) to the maze.

The following is the specification for the maze-solving model in Cell-DEVS:

$$\text{Maze} = < X, Y, S, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D > \tag{5.1}$$

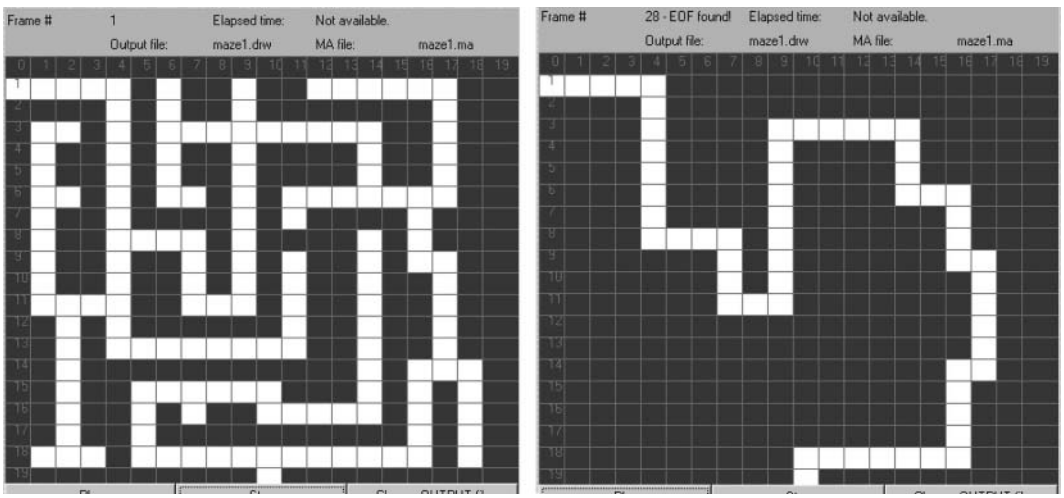
where

$X = Y = \varnothing$;
$S = \{ 0, 1 \}$;
$N = \{ (-1, 0), (0, -1), (0, 1), (1, 0), (0,0) \}$;
d = 100 ms (transport); and
τ: $N \rightarrow S$ is defined by the rules just described—that is:
$S = 1$ if cell (0,0) = 1;
$S = 1$ if cell (0,0) = 0 and number of wall neighbors ≥ 3; and
$S = 0$ if cell (0,0) = 0 and number of wall neighbors < 3.

This specification can be implemented in CD++, as shown in Figure 5.1.

```
[maze]
type : cell
dim : (20, 20)
delay : transport
border : nowrapped
neighbors :             maze(-1,0)
neighbors : maze(0,-1) maze(0,0)  maze(0,1)
neighbors :             maze(1,0)
localtransition : maze-rule

[maze-rule]
rule : 1 100 { (0,0) = 0 and (truecount = 3 or truecount = 4) }
rule : 0 100 { (0,0) = 0 and truecount < 3 }
rule : 1 100 { t }
```

FIGURE 5.1 Defining the Cell-DEVS specification in CD++. (From Lam, K., and G. Wainer. 2003. *Proceedings of 2003 SCS Summer Computer Simulation Conference,* Montreal, Quebec, Canada.)



(a) (b)

FIGURE 5.2 (a) The original maze and (b) the maze after processing in Cell-DEVS.

The model, originally presented in Lam and Wainer [2] and found in *./maze.zip,* is a 20 × 20 Cell-DEVS with transport delays, nonwrapped (for simplicity, we do not show the implementation of border cells), and with von Neumann's neighborhood. The *maze-rule* section defines the rules used for the local computing function. The first rule checks if the cell is empty (free cell); in this case, we count the number of cells with value 1 (*truecount* counts the number of inputs with a value = 1, considering that the total size of the neighborhood is 5). If there are three or four wall cells, we convert the cell into a wall. After 100 ms, this new value will be transmitted to the neighbors. The second rule checks the cases of free cells that remain free. In every other case (*t* means "true," and this precondition is always valid), the cell value becomes 1. Figure 5.2 shows an example of a maze and its solution.

If a maze has no solution, the model will generate a result without any cell in the final path—that is, a solid block of wall cells. Likewise, if the maze has different solutions, the cell space will stop evolving when all the solution paths are revealed. In those cases, further processing would be required for a single solution to be made available.

```
[ExMedia]
type : cell
dim : (9,9)
delay : transport
border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,1) (1,-1) (1,0) (1,1) (0,0)
localtransition : Ex-rules

[Ex-rules]
rule : 0 100 {(0,0)=0 and statecount(2)=0 }
rule : 2 100 {(0,0)=0 and statecount(2)>0 }
rule : 1 100 { (0,0) = 2 }
rule : 0 100 { (0,0) = 1 }
rule : { (0,0) } 100 { t }
```

FIGURE 5.3 Definition of an excitable media model.

Exercise 5.1

Define all the possible preconditions and postconditions for each cell in the maze model explicitly.

Exercise 5.2

Define all the possible preconditions and postconditions for each cell explicitly in the case of Moore's neighborhood. Modify the model to use Moore's neighborhood and analyze the results.

Exercise 5.3

Modify the size of the cell space and the initial values, and run the simulation again. Analyze the results obtained.

Exercise 5.4

Define an algorithm to find the shortest path between a group of solutions obtained as a result of the maze-solving algorithm.

We now show a simple model of excitable media [3], a phenomenon appearing in several real systems (e.g., the nervous tissue of the heart muscle, magnetic fields, forest fires). Figure 5.3 shows a Cell-DEVS representation of such phenomena in CD++ (which can be found in *./Exmedia.zip*).

We first define the Cell-DEVS coupled model and its parameters: size, neighborhood shape, kind of delay, and borders. The *Ex-rules* section represents the local computing function for the model. We use three different state values. For instance, for the heart tissue, these states could represent a cell that is resting, excited, or recovering; in a forest fire, the states could represent a cell without fire, burning, or burned. Here, the first rule defines the case when the cell and its neighbors are not excited (value 0); in this case, the cell must remain resting. The second rule is used when the cell is resting and there are excited neighbors (value 2); in this case, the cell becomes excited. The third and fourth rules make the cell change from the excited to a refractory period and, finally, change to the resting state again. In any other case, the cell does not change (it keeps the original value).

Figure 5.4 shows the results obtained when executing this model. It shows the evolution of the excitable medium using different neighborhoods. Figure 5.4(a) uses all the adjacent neighbors, as defined in Figure 5.3 (Moore's neighborhood). Figure 5.4(b) uses von Neumann's neighborhood.

Exercise 5.5

Increase the size of the cell space and put different initial excited cells. Use different neighborhood shapes and simulate the model. Analyze the results.

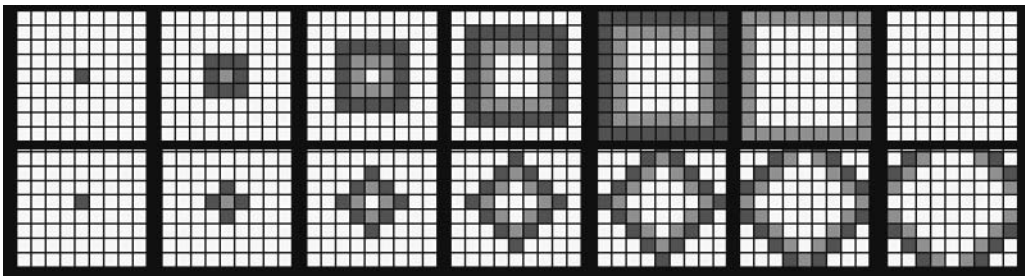


FIGURE 5.4 Results of ExMedia with different neighborhoods: (a) Moore's; (b) von Neumann's.

```
[Tension]
type : cell
dim : (40,40)
delay : transport
border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (1,-1) (1,0) (1,1) (0,0) (0,1)
localtransition : Ten-rules

[Ten-rules]
rule : 0 100 { statecount(0) >= 5 }
rule : 1 100 { t }
```

FIGURE 5.5 Surface tension model specification.

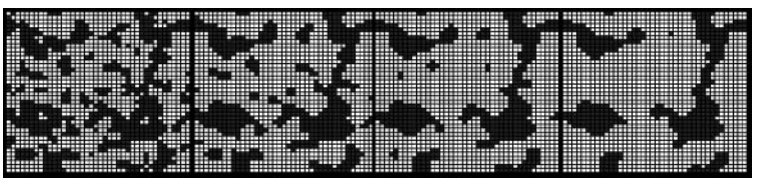


FIGURE 5.6 Execution results of the surface tension model.

Exercise 5.6

Change the rules to provide different delays for each of the rules.

The example in Figure 5.5 represents a surface tension model that employs a majority vote mechanism [4]. The rules simply compute the majority of the cell's values, which are binary, as seen in the figure (this model can be found in *./Tension.zip*).

The Cell-DEVS coupled model parameters include a grid of 40 × 40, Moore's neighborhood, transport delays, and wrapped borders. We have two states: the presence (value 1) or absence (value 0) of particles. In each step, the new state depends on a majority vote between the neighbors. A particle remains in the cell if at most four of the nine cells are occupied; otherwise, it becomes empty. Figure 5.6 shows how particles concentrate in the areas with higher tension. The resulting behavior of the surface is a representation of the majority vote rules defined earlier.

5.3 A MODEL OF A MICROWAVE OVEN

In this section, we show how to create a model of a programmable microwave oven system using DEVS. The microwave oven described here uses a keypad to set the heating time and power level,

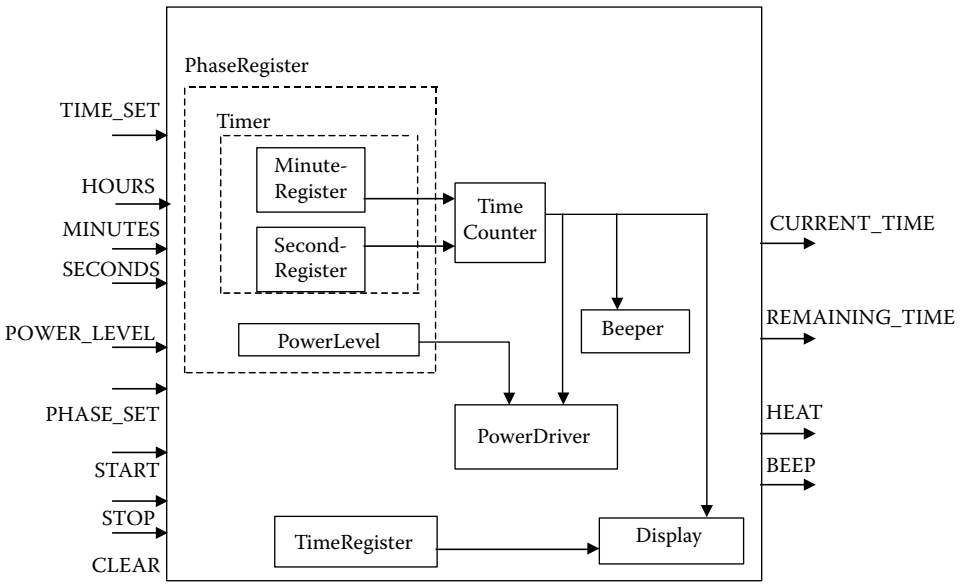


FIGURE 5.7 Structure of a model of a programmable microwave oven.

adjust the time of the day, etc. The heating cycle is started by pushing the START button, and it can be stopped by pushing STOP or CLEAR (in which case the time duration and the power level set for the heating cycle will be reset). The duration of the heating cycle is set by pressing the PHASE_SET button, followed by minutes and seconds input. The number of minutes is set by pressing the MINUTES/SECONDS buttons, which increase one unit every time they are pressed. A display shows the time of the day or the remaining time of a heating cycle. The model, found in *./microwaveoven.zip,* has the structure shown in Figure 5.7.

The model is organized in four levels. The top level uses the following inputs (corresponding to the buttons on the control panel of the oven):

- TIME_SET is used to set or adjust the current time displayed.
- PHASE_SET is used to set the program.
- POWER_LEVEL is used to set the power level (high, medium, or low).
- HOURS is used to set the hour to adjust to the time of day.
- MINUTES is used to set both the number of minutes of the heating cycle and the minutes for the time of day.
- SECONDS is used to set the number of seconds of the heating cycle.
- START starts the cycle.
- STOP stops the cycle.
- CLEAR erases the current set values (heating time, power level, and phase).

The outputs of the model are

- CURRENT_TIME, which represents the current time being displayed;
- REMAINING_TIME, which represents the remaining time for the current heating cycle;
- HEAT, which represents the power level for the current heating cycle (high, medium, or low); and
- BEEP, which represents the sound that the oven makes when it finishes a heating cycle.

The top model is decomposed into the following components:

- *PhaseRegister* is a coupled model used to record the information associated with the current heating cycle, including its duration and power level.
- *PowerDriver* takes the power level information from *PhaseRegister* and controls the heating power level (low, medium, or high).
- *TimeCounter* keeps the remaining time until the end of the cycle.
- *Beeper* controls whether the system needs to make a beeping sound. A beeping sound is made for about 3 s after a heating cycle is finished.
- *TimeRegister* records the current time of day.
- *Display* shows the remaining time of a heating cycle or the current time of day.
- *PhaseRegister* is decomposed into *Timer,* which records the information of the duration of a cycle, and *PowerLevel*, which records the heating level for the cycle.

Figure 5.8 shows the formal specification of the model *TimeRegister* and its implementation in CD++. As previously discussed, this model keeps the current time of the day displayed on the panel when the microwave oven is not heating. It uses three inputs: *time_set*, used to adjust the current time; *hours*, used to adjust the hour of the day (which takes effect only when *time_set* is pressed, increasing the hour by 1 per time pressed); and *minutes*, which is used to adjust the current minute. The model outputs *cur_time*, the current time generated from the model (hours * 100 + minutes; i.e., 10:23 a.m. will be sent as 1023).

As we can see, the external transition function is in charge of reacting to the different buttons pressed. If we press the *hours* button (represented by the corresponding input port), we increase the hours counter (resetting it to 0 after 24 h). If the *minutes* button is pressed, the minute counter is increased. In these two cases, we schedule an instantaneous internal transition. However, if we press the *time_set* button, we have two different behaviors. When the button is pressed, it means the user wants to synchronize the seconds to the beginning of the next minute (i.e., the user must release the button to synchronize the seconds to the minute). Thus, we passivate the model waiting for the next external event (which should be the button being released). When the button is finally released, we need to start counting time (the clock has a precision of 1 min, so we schedule an internal transition function in 1 min).

The output function will get the current time and will send it through the output port *cur_time,* and the internal transition will increment the current time in 1 min ("wrapping" the minutes to hours and hours to days), after which we schedule the next internal transition in 1 min.

Figure 5.9 shows the specifications and implementation of the cooking timer model.

The model (and its CD++ implementation) defines the components of the timer and their interrelationship. We have two registers (one for minutes and one for seconds) interconnected within this coupled model. The model can receive inputs from the seconds/minutes buttons or from the *Time_set* button. The result is the time in seconds and minutes. We have two registers (atomic models) to store information for seconds/minutes. In order to set a new value for these registers, the *time_set* button must be pressed. Thus, this input is linked to the input ports for each of the two registers. Likewise, when we press the seconds/minute buttons, an input must be sent to the corresponding registers; thus, we have a link between them. The two registers are independent (each of them is a register keeping the value for the corresponding time unit), so they are not interconnected (thus, the *IC* set is empty). Finally, the *select* function will pick the seconds updates first. Figure 5.10 shows an execution example for this model. Initially, we change the time of the day to 1:02 by pressing the *minute/hour/time_set* buttons. When the button is released, the time is updated. We then program a cycle, starting with 1 s, 1 min, and a power level of 1. At `45:102`, we program a new cycle of 4:04 min, and we start the cooking cycle. We can see how the remaining time starts to decrease. At `56:003`, we press *stop* (thus, we stop cooking) and then *clear* is pressed (resetting the oven).

```
X={Time_Set    <N>, Hours_in    <N>,
Minutes_in  <N> };
Y={Time_out   <N>};
S={State, hour, minute}

 ext ( s, e, x ) {
case port
 Hours_in:
    if (state==active)
     if (++hour==24)
       hour=0;
       state=active;
       ta(state)=0;

 Minutes_in:
    if (state ==active)
     if (++minute == 60)
       minute=0;
       state=active;
       ta(state)=0;

 Time_Set:
  if (value == true)
    state=active;
    ta(state)=Infinity;
  else
    state=passive;
    ta(state)=1  minute;

}

 int ( s ) {
case state
 active:
  increment minute by one;
  wrap the time appropriately;
  schedule next increment in 1 min.;

 passive:
  state = passive;
  ta(state)=infinity;
 }


  (s) {
 send time(hour*100+minute) to port Time_out;

}
```

```
CurTimeRegister::CurTimeRegister( const string &name )
: Atomic( name )
,time_set( addInputPort( "time_set" ) )
,hours( addInputPort( "hours" ) )
,minutes( addInputPort( "minutes" ) )
,cur_time( addOutputPort( "cur_time" ) ){ }

Model &CurTimeRegister::externalFunction
          ( const ExternalMessage &msg )   {
   if( msg.port() == hours) {
           if( state() == active ) {
                if ( this->hour == 23 )
                   this->hour = 0;
                else
                  ++(this->hour);
                holdIn( active, Time::Zero );
                }
           }

   else if( msg.port() == minutes)   {
           if( state() == active ) {
                if ( this->minute == 59 )
                   this->minute = 0;
                else
                  ++(this->minute);
                holdIn( active, Time::Zero );
                }
        }
   else if( msg.port() == time_set ) {
     if( msg.value() == true )
        // time_set was pressed.
         holdIn( passive , Time::Inf );

     else  // time_set was released.
          holdIn( active , one_minute );
       }

}


Model &CurTimeRegister::internalFunction
      ( const InternalMessage & ) {
   if ( state() == active ) {
      if( this->minute == 59) {
           this->minute = 0;
           this->hour++;
           if (this->hour==24)this->hour=0;
       }
           else
                ++(this->minute);
      holdIn( active, one_minute );
   }
  else
    holdIn( passive, Time::Inf );
}


Model &CurTimeRegister::outputFunction
    ( const InternalMessage &msg ) {
   sendOutput( msg.time(), cur_time,
      this->hour*100+this->minute ) ;
}
```

FIGURE 5.8 Atomic model definition: *TimerRegister* model.

EXERCISE 5.7

Add a new component, called *program*. It uses three inputs: *reheat* (1 min), *boil* (power level = 3, 2 min), and *cook* (power level = 2, 12 min). When the user presses each of these buttons, the rest of the components must be automatically programmed accordingly. Define the DEVS model for *Program*, code it in CD++, integrate it into the microwave coupled model, and run different test scenarios to verify that the desired behavior is obtained.

$$\text{Timer} = < X, Y, \{ \text{MinuteRegister, SecondRegister} \}, EIC, EOC, IC, SELECT > \quad (5.2)$$

where

X = { Minutes_in , Seconds_in, Time_set};
Y = {Time_Minutes, Time_Seconds };
EIC = { (Time_set.Self, Time_set.SecondRegister); (Time_set.Self, Time_set.MinuteRegister); (Seconds_in, Seconds.SecondRegister); (Minutes_in, Minute.MinuteRegister)};
EOC = { (Time_Minute.MinuteRegister, Time_Minute.Self), (Time_Second.MinuteRegister, Time_Second.Self)};
IC = ∅;
SELECT = { SecondRegister, MinuteRegister};

```
components : seccook@SecCookReg mincook@MinCookReg
in : c_time_set c_seconds c_minutes
out : c_time_seconds c_time_minutes
Link : c_time_set time_set@seccook
Link : c_time_set time_set@mincook
Link : c_seconds seconds@seccook
Link : c_minutes minutes@mincook
Link : time_seconds@seccook c_time_seconds
Link : time_minutes@mincook c_time_minutes
```

FIGURE 5.9 Coupled model definition: *TimerRegister* model: (a) formal definition; (b) CD++.

```
30:001 c_minutes       ? #input minutes
35:002 c_time_set      1 #input
                         time_set
36:001 c_minutes       ? #input minutes
38:001 c_hours         ? #input hours
39:002 c_time_set      0 #release
                         time_set
40:001 c_phase_set     1 #input
  phase_set
42:001 c_seconds       ? #input seconds
42:002 c_powerlevel_in ? #input
                         PowerLevel
42:003 c_minutes       ? #input minutes
42:004 c_phase_set     0 #release
                         phase_set
42:008 c_seconds       ? #input seconds
42:008 c_minutes       ? #input minutes
45:100 c_phase_set     1
45:102 c_seconds       ?
45:103 c_seconds       ?
45:104 c_seconds       ?
45:105 c_seconds       ?
45:106 c_powerlevel_in ?
46:102 c_minutes       ?
46:103 c_minutes       ?
46:104 c_powerlevel_in ?
46:105 c_minutes       ?
46:106 c_minutes       ?
48:002 c_start ? #press start
56:003 c_stop  ? #press stop
57:004 c_clear ? #press clear
```

```
...
36:001 c_cur_time 2   #display current time 2
38:001 c_cur_time 102 #display current time 102
42:001 c_rem_time 1   #display remaining time 1
       sec
42:003 c_rem_time 101 #display remaining time
       101
45:102 c_rem_time 1
45:103 c_rem_time 2
45:104 c_rem_time 3
45:105 c_rem_time 4
46:102 c_rem_time 104
46:103 c_rem_time 204
46:105 c_rem_time 304
46:106 c_rem_time 404
48:002 c_rem_time 404
48:002 c_powerlevel_out 3  #output powerlevel 3,
49:002 c_rem_time 403      #start cooking
50:002 c_rem_time 402
51:002 c_rem_time 401
52:002 c_rem_time 400
53:002 c_rem_time 359
54:002 c_rem_time 358
55:002 c_rem_time 357
56:002 c_rem_time 356
56:003 c_rem_time 355
56:003 c_powerlevel_out 0 #stop cooking
57:004 beep_out 1 #clear and beep
57:004 c_powerlevel_out 0
59:004 beep_out 0 #beep ends
```

FIGURE 5.10 Microwave oven execution.

5.4 MARKET DYNAMICS

The model presented in this section (found in *./market.zip* and presented in Liu and Wainer [5]) can be used to study the dynamics of dual markets where consumers choose between two competing products based on their preferences and the influence of other customers. In this example, each cell represents a consumer who periodically renews the license with one of two operating system (OS) providers.

The model is built as a Cell-DEVS with $S = \{0,1,2\}$ ($i = 0$: nonuser; $i = 1, 2$: user of OS_i) and Moore's neighborhood. In order to define $\tau{:}N \rightarrow S$, we use the definitions presented in Oda et al. [6]. The binary variable $X(c_{ij}, n, t)$ (with $n = 1$ or 2) will be 1 if the cell c_{ij} has a user of OS_n at time t. $X(c_{ij}, n, t)$ will be 0 if the cell c_{ij} does not have a user. We assume that $X(c_{ij}, n, t)$ and $X(c_{ij}, 3 - n, t)$ cannot be 1 at the same time for each cell; that is, no consumer can sign contracts with both providers simultaneously. The model represents three factors that influence the user's behavior, and the values of the contributing factors are defined by the following functions:

1. $U(c_{ij}, n, t)$, the **utility** that consumer c_{ij} can obtain by using OS_n in time t.
2. $E(c_{ij}, n, t)$, the **network externality** (i.e., the influence of other consumers):

$$E(c_{ij}, n, t) = (1 - \theta)\,(U_{\max} - U_{\min})\,N(c_{ij}, n, t - 1) \tag{5.3}$$

   where $N(c_{ij}, n, t - 1) = \Sigma_{\text{neighborhood}}\, X(c_{\text{neighborhood}}, n, t - 1)/\eta$ (η is the neighborhood size). The network externality effect for OS_n is based on the number of neighbors using OS_n at the previous time step. $(1 - \theta)\,(U_{\max} - U_{\min})$ is the absolute weight for the effect of network externality.
3. $P(c_{ij}, tn_s)$, the **price** that consumer c_{ij} must pay for the licenses:

$$P(c_{ij}, n, t) = (X(c_{ij}, n, t) + X(c_{ij}, 3 - n, t))\,P_{OSn}\,(c_{ij}, n, t),\ n = 1 \text{ or } 2 \tag{5.4}$$

   where $P_{OSn}(c_{ij}, n, t)$ stands for the price for the corresponding operating system, which is defined as $P_{OSn}\,(c_{ij}, n, t) = Q(n, t) + R(m, n, t)$. ($Q$ is a constant representing the global price for an OS, and R is a variable representing the fluctuation of the local price for this OS.)

In order to maintain a balance between profits and market share, a provider may reduce local price to attract more consumers when it loses market share to its rivals and raise the local price to gain more profits when it has a bigger market share than its competitors. The fluctuating local price is defined as $R(m, n, t) = Rn_{\min} + \mu(Rn_{\max} - Rn_{\min})\,(N(c_{ij}, n, t - 1) - N(c_{ij}, 3 - n, t - 1))$, where $Rn_{\max}$ and $Rn_{\min}$ are the maximum and minimum local price for the OS, and μ is a given constant representing the fluctuation coefficient. The value of $(N(c_{ij}, n, t - 1) - N(c_{ij}, 3 - n, t - 1))$ reflects the influence of market share on the local price. When the provider of OS_n owns a bigger share of the local market, this item has a positive value, which means the local price will rise. When the local market share goes down, the local price will fall accordingly.

At each time step, a cell calculates $V(c_{ij}, n, t) = U(c_{ij}, n, t) + E(c_{ij}, n, t) - P(c_{ij}, n, t)$ for all the three possible states and chooses the state that maximizes $V(c_{ij}, n, t)$ as its next state.

In our example, we did some simplifications. The skill accumulation function $L(c_{ij}, n, t)$ is calculated based only on a cell's *previous* state. In consequence, the local computing rules are defined by

Result: 1	Rule: $V(c_{ij}, 1, t) > V(c_{ij}, 2, t)$ AND $V(c_{ij}, 1, t) > 0$
Result: 2	Rule: $V(c_{ij}, 2, t) > V(c_{ij}, 1, t)$ AND $V(c_{ij}, 2, t) > 0$
Result: 0	Rule: $0 > V(c_{ij}, 1, t)$ AND $0 > V(c_{ij}, 2, t)$

```
[consumerschoice]
type : cell
dim : (30, 30)
delay : transport
border : wrapped
neighbors : (-1,-1)  (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
localtransition : choose_rule

[choose_rule]
rule : {if(stateCount(1)>(stateCount(2) - 2.25), 1,
       if(stateCount(2)>(stateCount(1) + 2.25), 2 , 0))} 100 {(0,0)=0}
rule : {if(stateCount(2)>(stateCount(1) + 6.75), 2 , 1)} 100 {(0,0)=1}
rule : {if(stateCount(1)>(stateCount(2) + 2.25), 1 , 2)} 100 {(0,0)=2}
```

FIGURE 5.11 Definition of local computing rules in CD++: mature market and same-price scenario.

Figure 5.11 shows the local computing rules for the new market and fluctuating price case. The first rule shows the case where the current state is 0, and we depreciate the price in each time step. The second rule shows the case where the state is 1; therefore, the price is incremented before depreciation. Finally, the rule shows a second case of increment before depreciation.

We show the results of a set of experiments with six different settings. The tests are categorized in two groups: mature and new markets. The former group uses an initial cell space where the three possible states for each cell are uniformly distributed (as in a mature market); the latter group starts with only a few cells representing new users. Three pricing strategies are defined: products with the same price, products with different prices, and products with fluctuating prices. The local computing rules are instantiated using the parameter values shown in Figure 5.12.

Figures 5.13–5.18 show the simulation results for these cases. White cells represent nonusers, light gray cells represent OS1, and dark cells represent OS2. The results in Figure 5.13 were obtained using the parameters in the first column of Figure 5.12, where the market is mature and both OSs have the same price. The figure shows that nonusers begin to use one of the two products with approximately equal probability and users using the same products tend to aggregate together to form their own society, which in turn enhances network externality.

In Figure 5.14, we also represent a mature market but with different prices (OS1 is cheaper than OS2, while all other parameters remain unchanged). Consequently, most of the nonusers choose to use OS1. Network externality again results in the aggregation of users.

Figure 5.15 shows a case for a mature market where the company providing OS2 has higher pricing flexibility ($\mu 2 = 1$), and OS1 offers more rigid prices ($\mu 1 = 0.2$). As a result, OS2 gains bigger

Settings	Mature Market			New Market		
	= Price	≠ Price	Fluctuating Price	= Price	≠ Price	Fluctuating Price
Parameters	U_{max}=.8 U_{min}=.4 θ =λ=.5 P_{OS1}=.25 P_{OS2}=.25	U_{max}=.8 U_{min}=.4 θ =λ=.5 P_{OS1}=.25 P_{OS2}=.3	U_{max}=.8,U_{min}=.4 θ=λ=.5 Q1=Q2=.1 $R1_{max}$=$R2_{max}$=.15 $R1_{min}$=$R2_{min}$=.05 μ1=.2, μ2=1	U_{max}=.8 U_{min}=.4 θ =λ=.5 P_{OS1}=.25 P_{OS2}=.25	U_{max}=.8 U_{min}=.4 θ =λ=.5 P_{OS1}=.25 P_{OS2}=.3	U_{max}=.8,U_{min}=.4 θ=λ=.5 Q1=Q2=.0 $R1_{max}$=$R2_{max}$=.3 $R1_{min}$=$R2_{min}$=0 μ1=.2, μ2=1

FIGURE 5.12 Parameter values for experimental frames.

FIGURE 5.13 Mature market and same-price scenario.

FIGURE 5.14 Mature market and different-price scenario.

FIGURE 5.15 Mature market and fluctuating-price scenario.

FIGURE 5.16 New market and same-price scenario.

FIGURE 5.17 New market and different-price scenario.

FIGURE 5.18 New market and fluctuating-price scenario.

market share. If the local market shares for both products are equal, the price fluctuation disappears and network externality becomes the sole force in determining consumers' decisions.

Figure 5.16 shows the development of a new market with two OSs at the same price. The market starts around a few initial users where the network externality takes effect. The number of users of both products rapidly grows at almost the same rate until the market is saturated. The initial users have been the pivots of a new market.

Figure 5.17 shows a new market in which OS1 rapidly monopolizes the whole market by virtue of its lower prices (sensitivity of price is high in a new market).

Finally, Figure 5.18 shows that two types of new users ripple out from the initial ones into alternating cycles. The development of the market exhibits a pattern that cannot be explained by any single factor of the value function.

Exercise 5.8

Modify the parameters in Figure 5.12 and re-run the experiments. Analyze the simulation results.

5.5 A PREDATOR–PREY MODEL

In this model, a predator seeks prey trying to escape [7]. Predators are always on the search; thus, prey must constantly avoid attacks. A predator detects prey by smell because the prey leaves its odor when moving. The predator moves faster than the prey, and when prey is nearby, the predator senses the smell and moves toward it. Each cell in the space represents an area of land (with vegetation, trees, etc.). Dense vegetation does not allow animals to advance, and both prey and predators are trying to abandon the area under study. When a prey is in a cell, it leaves a track of smell, which can be detected for a specific period. The cell's states are summarized in Figure 5.19.

In this model a cell's value finished in 1 represents *toward north* (N), finished in 2 represents *toward east* (E), finished in 3 represents *toward south* (S), and finished in 4 represents *toward west* (W) (and they are combined with values 200, 210, 300, and 310 to represent different states for predator and prey).

The prey follows the areas without vegetation, trying to abandon the area (211-4). If there is no vegetation around, it searches for it (201-4). For example, the cell value 214 represents prey following vegetation to find the exit in an eastern direction. The prey takes 1 s for each movement and 0.5 s to turn. The odor left while moving lasts 4 s, and it vanishes slowly (reducing intensity every 1 s; it is represented by the values 104-1. The prey tries to follow the vegetation; it cannot run over the thick vegetation, a predator, or strong smell (the two highest values). The predator is faster (taking 0.8 s to move and 0.3 s to turn). We use an inertial delay to represent the fact that prey ready to move might be caught by a faster predator. Both move in directions N/S/E/W (although we use Moore's neighborhood in order to avoid collisions).

Figure 5.20 shows the specification of the model using CD++ (found in *./prey.zip*). The cell's value equal to 400 represents thick vegetation, where the prey and predator cannot move. Hence, the first rule in Figure 5.20 tells us that a cell should be evaluated only if the cell's value is equal to 400. This rule has a very long delay (because we do not need to re-evaluate it again after initialization). The second set of rules defines the behavior of a predator following the smell left by prey. A cell with a value of 30*i*/31*i* represents a predator moving to the N/S/E/W.

The next set of rules governs the predator movement when attacking a prey found to the N. These rules work in pairs: one rule is used to move the animal to the new position, and the other is used to update the cell where the animal was. In the movement rules, we use an 800-ms delay (predator speed). As we can see, we first check if the cell has a predator moving N. Then we check to see if the cell to the N has prey (i.e., it has a number between 200 and 250) or traces of smell (i.e., a number larger than 100). In this case, the predator moves N and takes the cell.

The following rules are used by the prey while searching for vegetation. The prey moves in the opposite direction to the smell, leaving a trace of strong smell (104-rules A). When encountering vegetation (rules B), the prey turns (which takes 500 ms) and starts following the path formed by the vegetation.

The last rules represent the smell trace left by prey. Four values represent different dispersion phases. The first line in the *smell* rules governs the smell attenuation by subtracting 1 from the actual cell's value every second. The last rule changes the actual cell's value to zero (no smell in

Cell Value	Description
1..4	An animal moving toward N (1), W (2), S (3), or E (4).
200	Prey looking for vegetation.
210	Prey following the vegetation to leave the area.
300	Predator looking for vegetation.
310	Predator following the forest to find the exit.
400	Thick vegetation (does not allow animal movement).
0	Thin vegetation (allows animal movement).
101..104	Smell.

FIGURE 5.19 Cell state codification.

```
[pred-prey]
type : cell          dim : (20,20)        delay : inertial
neighbors: (-1,-1)(-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1) (1,0) (1,1)
border : nowrapped       localtransition : movement

[movement]
rule : { (0,0) } 100000 { (0,0)=400}

%Predator following smell
rule : 311 300 { (0,0) >300 and (0,0) <350 and (0,0)!=311 and (-1,0) >=100 and (-
       1,0)<250}
rule : 313 300 { (0,0) >300 and (0,0) <350 and (0,0)!=313 and (0,1) >=100 and
       (0,1)<250}
rule : 314 300 { (0,0) >300 and (0,0) <350 and (0,0)!=314 and (1,0) >=100 and
       (1,0)<250}
rule : 312 300 { (0,0) >300 and (0,0) <350 and (0,0)!=312 and (0,-1) >=100 and (0,-
       1)<250}

%Predator eating prey found towards N
rule: 0 800 {(-1,0)>100 and (-1,0)<250 and ((0,0)=311 or (0,0)=301)}
rule: 311 800 {(0,0)>100 and (0,0)<250 and ((1,0)=311 or (1,0)=301)}

% prey searching for vegetation (A)
rule : 104  1000 { (0,0) = 211 and (-1,0) <103 }
rule : 211  1000 { (1,0) = 211 and (0,0)<103 and (0,1)<103}

% vegetation found; turn and follow vegetation (B)
rule : 201  1000 { (1,0) = 211 and (0,0)<103 and (0,1)>=103}
rule : 202  500  { (0,0) = 211 and (-1,0)>=103 }
...

%Rules from smell path
rule: {(0,0)-1} 1000 {(0,0)>101 and (0,0)<105}
rule: 0 1000 {(0,0)<=101}
rule: {(0,0)} 100 {t}
```

FIGURE 5.20 Specification of prey–predator model.

FIGURE 5.21 Execution result of prey–predator model. (From Ameghino, J., and G. Wainer. 2004. *Proceedings of Artificial Intelligence, Simulation and Planning,* Jeju Island, Korea.)

the cell). Because we use inertial delays, any changes in the computation will preempt the previous state change. As we can see, we use different delays for each group of cells, in order to better quantify the timing information for each of the behaviors.

Figure 5.21 shows the execution results of the model. The prey is trying to escape from the predator until it is captured.

Exercise 5.9

Move the expression `((0,0) = 311 or (0,0) = 301)` to the front of the rule and study the difference in time of the execution results (a rule with a false condition in an AND stops being evaluated).

Exercise 5.10

Give different numbers of predators and prey and repeat this exercise.

Exercise 5.11

Change the first movement rule to the end of the model, and study the difference in execution time. (Rules are evaluated in order of appearance; if one is found to be true, the rest are not evaluated.) Propose other changes in the order for rule definitions to improve performance of the simulation.

5.6 HEAT DIFFUSION

In this section, we introduce a multimodel that includes three components: a three-dimensional space reproducing the behavior of a room and two generators (one source of heat and one source of cold). The model, which can be found in *./3d_heatDiffusion.zip*, was introduced in Wainer and Giambiasi [8]. Each cell in the space contains a temperature value in the area corresponding to the cell. The temperature is calculated as the average of the one in the cell and in its near neighbors, depicted in Figure 5.22.

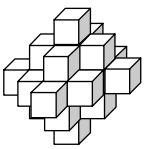


FIGURE 5.22 Neighborhood shape for the heat diffusion model.

A heating device gives high temperatures to cells (2,2,1) and (3,3,0). The heater simulator generates a flow of temperatures between 24 and 80°C with uniform distribution. On the other hand, a source of cold air (e.g., an air conditioner in the summer or an open window in winter) is connected to the cells (1,3,3) and (3,3,2). The corresponding model is used to create values in the range [–10, 15], also with uniform distribution. Both generators create values after x s, where x follows an exponential distribution with mean of 40 s. The coupling scheme of the models is shown in Figure 5.23, and Figure 5.24 shows the complete model definition.

The first lines define the top model and its components. Then we define the coupling scheme between the models: we connect the heat generator to the *inputheat* port in the cell space, and then we connect this port to specific cells (and we use a similar scheme for the cold generator). Then we include the definition of the cellular model (*surface*) that represents the room to be studied, including the grid size, kind of delay, and border. It is composed of 10 × 10 × 4 cells, and the local computing function calculates the present value as an average of the inputs (coming from the neighbors or the external generators).

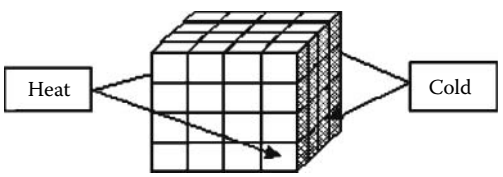


FIGURE 5.23 Coupling scheme of the heat diffusion model.

Because we need specialized behavior on the cells receiving inputs on these new ports, we define different functions to do this (*setHeat, setCold*). The *setHeat* rules define a function that generates a temperature value in the range [24, 80], using a uniform probabilistic distribution. These values are received through the *in* port of the cells (2,2,1) and (3,3,0). Likewise, line *setCold* defines the function that generates temperatures in the range [–45, 10] with uniform distribution. These values will be received through the *in* port of the cells (1,3,3) and (3,3,2). The remaining lines in the file define arguments for the heat and cold generators (which send temperature values with a frequency chosen using a random number generator with exponential distribution, with an average value of 40 s).

Figure 5.25 shows some of the results generated when the model is executed.

In simulated time 00:00:01:000, the heater/cold sources produce changes in the cells where they are connected. Consequently, the state of the neighbors of these cells will change in time 00:00:02:000. In the following stages, the rest of the cells will also change, following the rules of the model. Finally, at 00:00:15:245, the cold source will produce a change in the cells (1,3,3) and (3,3,2), with temperatures of –12.9 and –4.2°C. These values will affect the neighboring cells.

```
[top]
components : surface genHeat@Generator genCold@Generator
link : out@genHeat inputHeat@surface
link : out@genCold  inputCold@surface

[surface]
type : cell
dim : (4, 4, 4)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : (-1,0,-1) (0,-1,-1) (0,0,-1) (0,1,-1) (1,0,-1) (-1,-1,0) (-1,0,0) (-1,1,0)

neighbors : (0,0,0) (0,1,0) (1,-1,0) (1,0,0) (1,1,0) (-1,0,1) (0,-1,1) (0,0,1) (0,1,1)

neighbors : (0,0,-2) (0,0,2) (0,2,0) (0,-2,0) (2,0,0) (-2,0,0) (0,-1,0) (1,0,1)
in : inputHeat inputCold
link : inputHeat in@surface(3,3,0)
link : inputHeat in@surface(2,2,1)
link : inputCold  in@surface(3,3,2)
link : inputCold  in@surface(1,3,3)
localtransition : heat-rule
portInTransition : in@surface(3,3,0)  setHeat
portInTransition : in@surface(2,2,1)  setHeat
portInTransition : in@surface(3,3,2)  setCold
portInTransition : in@surface(1,3,3)  setCold

[heat-rule]
rule : { ( (-1,0,-1) + (0,-1,-1) + (0,0,-1) + (0,1,-1) + (1,0,-1) + (-1,-1,0) +
           (-1,0,0) + (-1,1,0)  + (0,-1,0) + (0,0,0) + (0,1,0) + (1,-1,0) +
           (1,0,0) + (1,1,0) + (-1,0,1) + (0,-1,1) + (0,0,1) + (0,1,1) + (1,0,1)
           + (0,0,-2) + (0,0,2) + (0,2,0) + (0,-2,0) + (2,0,0) + (-2,0,0) ) / 25 } 1000 ( t )


[setHeat]
rule : { uniform(24,80) } 1000 { t }

[setCold]
rule : { uniform(-45,10) } 1000 { t }

[genHeat]
distribution : exponential
mean : 40
initial : 1
increment : 0

[genCold]
distribution : exponential
mean : 40
initial : 1
increment : 0
```

FIGURE 5.24 Model of heat diffusion.

Figure 5.26 shows a graphical representation of the model's execution (this graph shows only one of the planes in the model). In Figure 5.26, we can observe the heat diffusion using different shades. We can see the influence of the heat/cold generators, although the cold generators are not activated as often as the heaters because the used distribution has a different mean, thus implying that the event with cold information appears in during longer periods.

Exercise 5.12

Change the initial conditions for the model and the values for the temperatures generated. Run the simulation and analyze the results.

```
Line : 593 - Time: 00:00:02:000
        0    1    2    3          0    1    2    3          0    1    2    3          0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 25.9 24.0 25.9 23.2|   0| 24.0 24.0 28.0 24.9|   0| 23.0 24.0 23.0 20.3|   0| 21.3 24.0 21.3 22.3|
   1| 21.3 24.0 23.3 25.2|   1| 24.0 26.0 26.0 20.6|   1| 21.3 24.0 23.3 19.3|   1| 21.3 18.6 21.3 21.3|
   2| 25.9 26.0 27.9 25.3|   2| 28.0 26.0 26.0 27.0|   2| 23.0 26.0 25.0 22.3|   2| 21.3 24.0 25.3 22.3|
   3| 25.9 27.9 27.9 24.0|   3| 24.9 26.0 27.0 27.0|   3| 23.0 22.0 25.0 26.9|   3| 24.9 24.0 24.9 19.6|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
Line : 741 - Time: 00:00:02:640
        0    1    2    3          0    1    2    3          0    1    2    3          0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 25.9 24.0 25.9 23.2|   0| 24.0 24.0 28.0 24.9|   0| 23.0 24.0 23.0 20.3|   0| 21.3 24.0 21.3 22.3|
   1| 21.3 24.0 23.3 25.2|   1| 24.0 26.0 26.0 20.6|   1| 21.3 24.0 23.3 19.3|   1| 21.3 18.6 21.3 21.3|
   2| 25.9 26.0 27.9 25.3|   2| 28.0 26.0 41.8 27.0|   2| 23.0 26.0 25.0 22.3|   2| 21.3 24.0 25.3 22.3|
   3| 25.9 27.9 27.9 38.5|   3| 24.9 26.0 27.0 27.0|   3| 23.0 22.0 25.0 26.9|   3| 24.9 24.0 24.9 19.6|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+

          ...               ...               ...
Line : 8860 - Time: 00:00:15:245
        0    1    2    3          0    1    2    3          0    1    2    3          0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 24.1 24.1 24.1 24.1|   0| 24.1 24.1 24.1 24.1|   0| 24.1 24.1 24.1 24.1|   0| 24.1 24.1 24.1 24.1|
   1| 24.1 24.1 24.1 24.1|   1| 24.1 24.1 24.1 24.1|   1| 24.1 24.1 24.1 24.1|   1| 24.1 24.1 24.1-12.9|
   2| 24.1 24.1 24.1 24.1|   2| 24.1 24.1 24.1 24.1|   2| 24.1 24.1 24.1 24.1|   2| 24.1 24.1 24.1 24.1|
   3| 24.1 24.1 24.1 24.1|   3| 24.1 24.1 24.1 24.1|   3| 24.1 24.1 24.1 -4.2|   3| 24.1 24.1 24.1 24.1|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+

Line : 9184 - Time: 00:00:15:640
        0    1    2    3          0    1    2    3          0    1    2    3          0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 24.6 24.6 24.6 24.6|   0| 24.6 24.6 24.6 24.6|   0| 24.6 24.6 24.6 24.6|   0| 24.6 24.6 24.6 24.6|
   1| 24.6 24.6 24.6 24.6|   1| 24.6 24.6 24.6 24.6|   1| 24.6 24.6 24.6 24.6|   1| 24.6 24.6 24.6 24.6|
   2| 24.6 24.6 24.6 24.6|   2| 24.6 24.6 24.6 24.6|   2| 24.6 24.6 24.6 24.6|   2| 24.6 24.6 24.6 24.6|
   3| 24.6 24.6 24.6 24.6|   3| 24.6 24.6 24.6 24.6|   3| 24.6 24.6 24.6 24.6|   3| 24.6 24.6 24.6 24.6|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
          ...               ...               ...
```

FIGURE 5.25 Simulation results of the heat diffusion model.

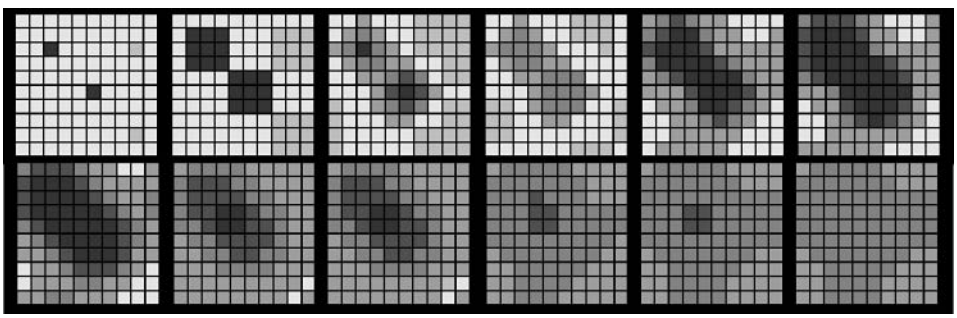


FIGURE 5.26 Execution results of the heat diffusion model.

Exercise 5.13

Change the size of the room and repeat the simulation.

Exercise 5.14

Modify the neighborhood shape and use a $3 \times 3 \times 3$ cube. Repeat the simulation and analyze the results with this change.

5.7 GSM CELLULAR NETWORK AUTHENTICATION SIMULATOR

GSM (Global System for Mobile communications) is a radio-based cellular network used for cell phones, PDAs, and similar devices. It is the most popular standard for these applications because

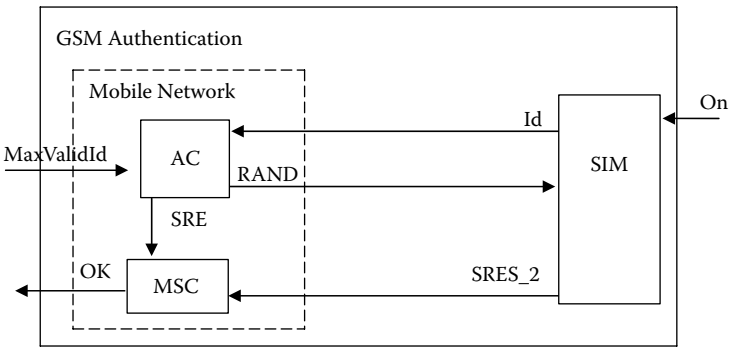


FIGURE 5.27 Structure of GSM authentication model.

it permits roaming in different countries. These networks include a variety of security services implemented between the network and the handsets, including authentication to any given handset. This functionality is performed in a SIM (Subscriber Identity Module) card, which is contained in the handset (hence, from here on we will refer to the SIM card and handset interchangeably). The main function of the SIM is to identify a customer, because the card can be interchanged between phones if the user decides to change devices. Each card is assigned an identifier and a subscriber authentication key (K_i), which is known only to the SIM card and the network (it is never transmitted over the air).

The authentication method modeled here is a *challenge response* method: the handset contacts the network claiming a certain identity (*Id*). To verify the SIM card's identity without giving away the K_i, the network generates a random 128-bit number that is broadcast to all the phones in the area. Both the handset and the network use the *A3* algorithm (found at http://www.gsmworld.com), which takes this number and the K_i and generates a 32-bit signed response (SRES). The handset then transmits the SRES back to the network, which verifies that it is the expected SRES (if not, authentication is rejected).

The GSM authentication model, depicted in Figure 5.27, has two components: the SIM card (SIM) and the mobile network, which is further decomposed into the access control (AC) and the mobile switching center (MSC) models. The GSM authentication model uses two inputs and one output. *MaxValidId* is used to specify the value of the maximum Id recognized by network, and *on* is an integer specifying the *Id* of the mobile phone that is requesting authentication (to simulate the event of a mobile phone being turned on somewhere in the network). The output *Ok* takes a value of 1 for a successful authentication and 0 otherwise. The mobile network is assumed to be very fast compared to the SIM card, and its operation time is assumed negligible (and modeled as an instantaneous event). Meanwhile, the SIM card takes 3 s to send the Id to the network after receiving an *on* input and another 3 s to calculate the value of *SRES_2*.

The DEVS specifications and CD++ implementation of the SIM model are shown in Figure 5.28.

The *on* input of the SIM model is an integer representing the *Id* of the SIM card. When an input is received at this port, its value is sent out the *Id* output port after a preparation time. Immediately after an input is received at the *on* input port, the SIM model is in a state of waiting for a *RAND* input; therefore, if during this time another *on* input is received, it will be discarded. However, when the *RAND* input is received, its value is added to the value of *Id* and, after a delay (*preparationTime*), the result of this addition is sent out the *SRES_2* output port. Any input received during this preparation time is ignored.

```
S = {Sigma, Phase, Authenticating, IdNum,
SRES2_Num, RandNum}
X = {On, RAND}
Y = {Id, SRES_2}

 ext(Authenticating, state, SRES2_Num, IdNum,
                                    RandNum, e, x)
{
     case state
          Passive:
             if (not Authenticating)  {
                  If (x is from On) {
                     IdNum = x.value;
                       state = Active;
              //Wait some time before outputting Id
                       ta(state) = Preparation Time
                       }
              else if (x is from RAND) {
                   RandNum = x.value;
                   SRES2_Num=RandNum+IdNum;
                   state = Active
                  ta(state) = Preparation Time
             }
     }
}

 int(State, Authenticating, IdNum, RandNum,
                                    SRES2_Num) {
     case Authenticating:
          // The Id is sent out and now waiting
          // for authentication from Network
          False: Authenticating = True
                   passivate ():
        // The SRES_2 is sent out and now
        // setting back to initial state.
          True: Authenticating = False
               passivate ():
}

 (Authenticating) = Send SRES2_Num to port
    SRES_2
 (!Authenticating) = Send IdNum to port Id
```

```
SIM::SIM( const string &name ) : Atomic(
name )
, On( addInputPort( "On" ) )
, Rand( addInputPort( "Rand" ) )
, Id( addOutputPort( "Id" ) )
, SRES_2( addOutputPort( "SRES_2" ) ) { ... }

Model &SIM::externalFunction( const
                ExternalMessage &msg ) {


if (this->state() == passive) {
  if (msg.port() == On && !Authenticating)
{


      IdNum = static_cast < int >
      (msg.value());


       holdIn( active, preparationTime );
  }

  else if (msg.port()==Rand
  &&Authenticating ){
       RandNum = static_cast <int>
       msg.value();
       SRES2_Num = RandNum + IdNum;

       holdIn( active, preparationTime );
       }
   }
       return *this;
}
Model &SIM::internalFunction( const
                  InternalMessage & )  {

       Authenticating = !Authenticating;

       passivate();
       return *this ;
}









Model &SIM::outputFunction( const
InternalMessage &msg )  {
  if (!Authenticating)
      sendOutput( msg.time(), Id, IdNum) ;
  else
   sendOutput( msg.time(), SRES_2, SRES2_Num) ;
       return *this ;
}
```

FIGURE 5.28 Defining the SIM model.

The model can be found in *./GSM_Authentication_Sim.zip*, and the coupled model (Figure 5.29) for the whole system is defined.

Figure 5.30 shows the simulation results for the SIM model for the given inputs.

GSM=<X,Y,{Mobile_Network,SIM},EIC,EOC,IC,SELECT>
X = {MaxValidId, On}
Y = {Ok}
EIC = { (MaxValidId.Self, MaxValidId.Mobile_Network),
(On.Self, On.SIM); }
EOC = { (OK.Mobile_Network, OK.Self) }
IC = { (Id.SIM, ID.Mobile_Network);
(SRES_2. SIM, SRES_2.Mobile_Network);
(RAND.Mobile_Network, RAND.SIM); }
SELECT: ({Mobile_Network, SIM}) = SIM;

```
[top]
components : SIMCard@SIMCard
MobileNetwork
in : MaxValidId
in : On
out : Ok
Link : MaxValidId
MaxValidId@MobileNetwork
Link : On On@SIMCard

Link : Ok@MobileNetwork Ok
Link : Id@SIMCard Id@MobileNetwork
Link : SRES_2@SIMCard
SRES_2@MobileNetwork
Link : RAND@MobileNetwork
RAND@SIMCard

[SIMCard]
preparation : 00:00:03:00
distribution : normal
mean : 50
deviation : 25
```

FIGURE 5.29 Defining the network coupled model.

```
00:00:10:00 On 8                    00:00:13:00 id 8
00:00:30:00 Rand 15                 00:00:33:00 sres_2 23
00:00:45:00 On 12                   00:00:48:00 id 12
00:01:00:00 On 55
00:01:05:00 Rand 25                 00:01:08:00 sres_2 37
00:01:25:00 On 46                   00:01:28:00 id 46
00:01:30:00 Rand 5                  00:01:33:00 sres_2 51
00:01:55:00 Rand 17
00:01:55:00 On 5
00:01:56:00 On 10                   00:01:58:00 id 5
00:02:00:00 Rand 21
00:02:01:00 Rand 50                 00:02:03:00 sres_2 26
```

FIGURE 5.30 Simulation results for the SIM model.

For each *on* input (except for the ones in bold), we obtain an output with an *Id* output with the same value (delayed 3). The others are discarded. Likewise, for each *Rand* input (except for the ones in bold), the output contains an *SRES_2* value with the sum of the *Rand* input and the previous (nondiscarded) *on* value. Again, the *SRES_2* outputs appear 3 s later than the *Rand* input (*Rand* inputs in bold are discarded).

The GSM authentication simulator coupled model is made up of the combination of the SIM atomic model and the mobile network coupled model. The GSM authentication simulator has two inputs: *on* and *MaxValidId*. The behavior of the inputs is as described in the earlier sections—namely, receiving an *on* input causes the SIM to start the authentication process by sending its *Id* (the value received at the *on* input port) to the mobile network. Receiving a *MaxValidId* input sets the maximum *Id* value that will be considered an authentic *Id* by the mobile network. The only output of the GSM authentication simulator is the *Ok* output, which indicates, for each received *on* input, whether that particular mobile phone was successfully authenticated or not. The *Ok* output should appear 6 s after the time the *on* input was received due to the *preparationTime* delays in the SIM atomic model. For each *on* input with a value less than or equal to the last *MaxValidId* input, the output file should contain an *Ok* output of 1. Otherwise, it should contain an *Ok* output of 0. The outputs are 6 s after the inputs. This global behavior can be seen in the simulation results for the coupled model shown in Figure 5.31.

00:00:10:00 MaxValidId 27	
00:00:15:00 On 25	00:00:21:000 ok 1
00:00:25:00 On 45	00:00:31:000 ok 0
00:00:35:00 On 15	00:00:41:000 ok 1
00:00:40:00 MaxValidId 5	00:00:51:000 ok 0
00:00:45:00 On 10	

FIGURE 5.31 Simulation results for the coupled model.

00:00:10:00 MaxValidId 27	
00:00:15:00 On 25	00:00:21:000 ok 1
00:00:25:00 On 45	00:00:31:000 ok 0
00:00:35:00 On 15	00:00:41:000 ok 1
00:00:40:00 MaxValidId 5	
00:00:45:00 On 10	
00:00:46:00 On 1	
00:00:47:00 On 1	
00:00:48:00 On 1	
00:00:49:00 On 1	
00:00:50:00 On 1	
00:00:51:00 On 1	00:00:51:000 ok 0
00:00:52:00 On 1	00:00:58:000 ok 1

FIGURE 5.32 Simulation results: multiple consecutive inputs.

The test in Figure 5.32 was designed to reproduce the one in Figure 5.31, with the addition of a continuous input (every second for 7 s) at the *on* port at the end. The goal is to examine the behavior of the simulator when new inputs are received while it is busy with a previous input.

As expected, the last consecutive inputs resulted in only one output corresponding to the last input (all other inputs from time 46 to 51 are ignored). The reason for this is that, in our model, the SIM card takes 3 s to prepare to send out the *Id* output to the network and 3 s to prepare to send out the *SRES_2* output to the network (the network time itself is negligible). During these 6 s, the SIM card ignores any inputs and only the input received after 7 s is not discarded.

Figure 5.33 shows the changes in each of the relevant state variables for the preceding inputs using a CD++ modeler graphical interface.

The first event received is on *MaxValidId* (a value of 27 at time 10). This value is kept until time 40, where it becomes 5 (as in the specified input). Also following the inputs, the *on* state variable gets the appropriate values at the appropriate times. In response to the *on* inputs, an output is seen at the *Id* state variable 3 s after the *on* input is received. At the same time (3 s after input *on*), a random number is generated and put in state variable *Rand*. Notice that, for the consecutive inputs at the end, no *Id* or *Rand* values are generated because those inputs are ignored. Now, another 3 s after each of the *Rand* values, an *SRES_2* value is generated. This value is sent to the MSC, which immediately produces an output, and it can be seen in the graph of *Ok*.

5.8 SUMMARY

In this chapter, we have introduced the use of CD++ to model a few simple applications. Initially, we presented a few Cell-DEVS models based on traditional cellular automata applications and showed how to define them using the toolkit. We also presented a basic model to represent a microwave oven as a DEVS model, trying to mimic the behavior of the oven controller and display. We also introduced a model of a duopolistic market represented as a cellular model, in which we can analyze the behavior of such a market under varied conditions. We presented a model of prey running away from a predator and a three-dimensional heat diffusion model. Finally, we introduced a simple networking application based on authentication for GSM cellular phones. These multiple applications show the basic ideas on how to create DEVS and Cell-DEVS models in CD++. In the following chapters, we will focus on more advanced models in different fields of application.

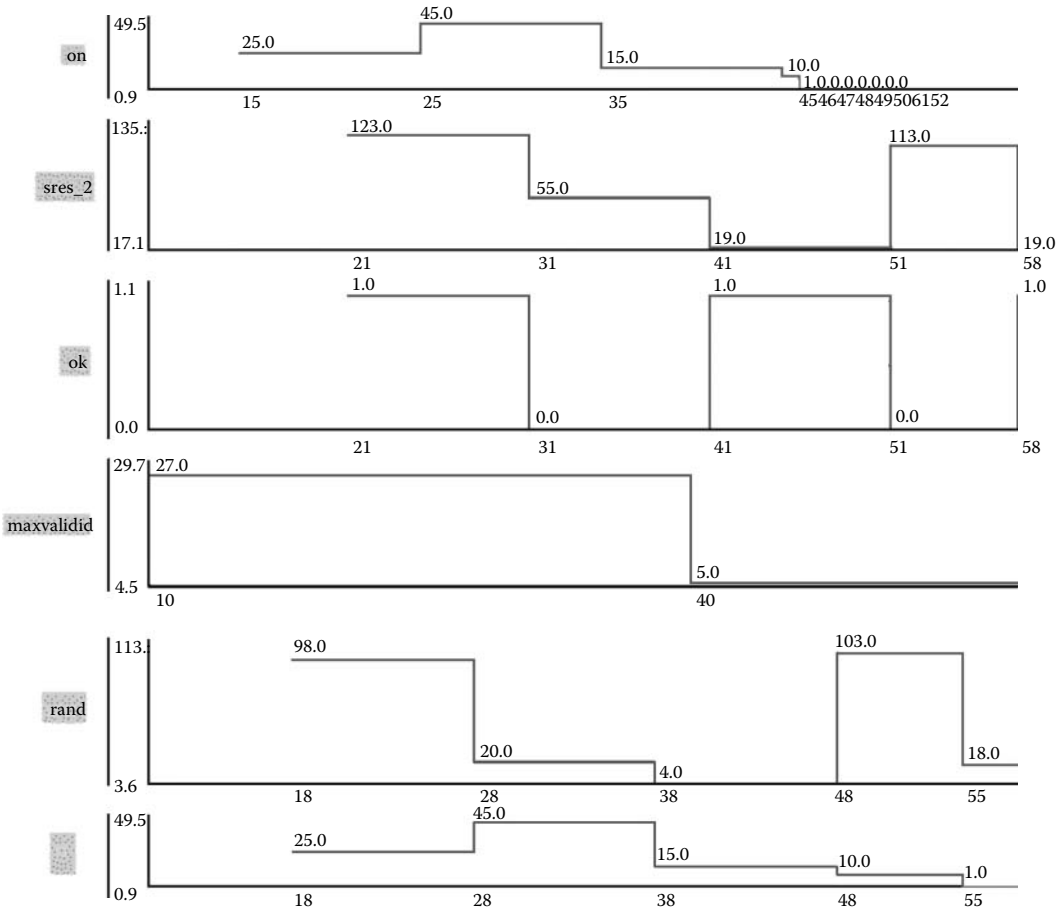


FIGURE 5.33 Graphical display from the model.

REFERENCES

1. Nayfeh, B. 1993. Cellular automata for solving mazes. *Doctor Dobb's Journal,* February 1993.
2. Lam, K., and G. Wainer. 2003. Modeling of maze-solving problems using cell-DEVS. *Proceedings of 2003 SCS Summer Computer Simulation Conference,* Montreal, QC, Canada.
3. Toffoli, T. 1994. Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular automata). *Proceedings of International Conference on Cellular Automata for Research and Industry,* Rende, Italy.
4. Toffoli, T., and N. Margolus. 1987. *Cellular automata machines: A new environment for modeling.* Cambridge, MA: MIT Press.
5. Liu, Q., and G. Wainer. 2005. Simulating market dynamics with CD++. *Proceedings of International Conference on Computational Science,* Atlanta, GA, 368–372.
6. Oda, S. H., K. Iyori, M. Ken, and K. Ueda. 1999. The application of cellular automata to the consumer's problem. *Proceedings of Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL'98,* Canberra, Australia, 454–461.
7. Ameghino, J., and G. Wainer. 2004. Application of the cell-DEVS formalism for modeling cell spaces. *Proceedings of Artificial Intelligence, Simulation and Planning,* Jeju Island, Korea.
8. Wainer, G., and N. Giambiasi. 2001. Timed cell-DEVS: Modelling and simulation of cell spaces. In *Discrete event modeling & simulation: Enabling future technologies,* ed. H. S. Sarjoughian and F. E. Cellier. New York: Springer–Verlag.

6 Discrete-Event Applications with DEVS

6.1 INTRODUCTION

In this chapter we introduce a few generic applications of discrete-event systems using DEVS models, focusing on their implementation using the CD++ toolkit. We include a model of a simple automated teller machine (ATM), a model of a water reservoir for a city, and a traffic light model with radar to detect vehicles moving at high speed.

6.2 A MODEL OF AN ATM

We want to model the behavior of a simple machine that is a simplified version of a fully featured ATM (the machine we model here is only capable of dispensing money). The operations of entering a PIN and withdrawing available funds are generated at random (assuming that 90% of the time a PIN entered is correct and that 80% of the time the amount requested is available to the customer). The goal of such a model is to study an estimate for the duration of a whole withdrawal operation. The model has the structure shown in Figure 6.1.

As we can see, the machine is composed of three submodels:

- **Card reader** is responsible for accepting a card into the machine and reading the customer's information stored on it. It also returns the card at the end or after a failed transaction.
- **Cash dispenser** is in charge of providing the required (and approved) funds to a customer.
- **Authorization** is a coupled model that receives a PIN number, validates it, gets the amount requested, checks the available funds, and gives the approved funds to the customer. It is composed of the following:
  - **User interface** acts as the interface between the customer and the ATM, permitting the customer to enter the PIN and amount of money to withdraw.
  - **PIN verifier** verifies the PIN entered by the customer (which is simulated as a random function in which we use a uniform probabilistic distribution to decide if the PIN entered is correct or not). If valid, it returns OK (so if the **PIN verifier** gets connected to the user interface, this model will know about this fact and it can then request the amount to be withdrawn). If the PIN is not valid, the model informs about this fact (so, for instance, the **user interface** model can ask the user to re-enter the PIN).
  - **Balance verifier** checks to see if the customer has enough funds to cover the required amount (which is also simulated as a random function in which we use a uniform probabilistic distribution to decide if the customer has enough funds or not). If not, it returns a request for a new amount.

We first need to define each of the atomic models in the hierarchy. As an example, we will show the details of the PIN verifier model (the remaining atomic models, which are built using similar ideas, can be found in *./atm.zip*). As previously discussed, the PIN verification will be simulated by generating a random number with uniform distribution; in 90% of cases it would be considered to match the PIN stored, as seen in Figure 6.2.

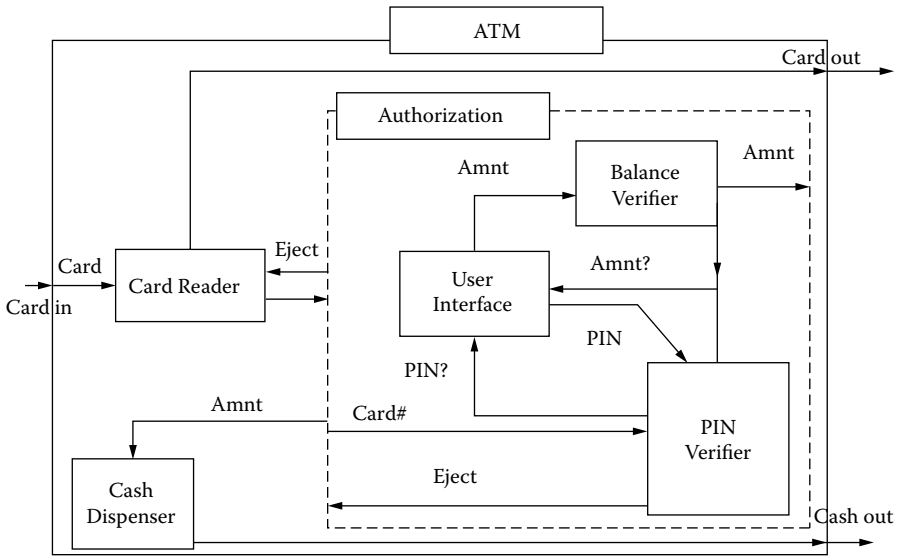


FIGURE 6.1 Structure of the ATM model.

Initially, we define the model's interface (one input port to receive a PIN and three output ports: one to request the card to be ejected, one to request a new PIN to be entered, and another to inform that the PIN is correct and now the amount to be withdrawn needs to be requested). The external transition function stores the PIN number and then generates a random number with uniform distribution; 90% of the time the PIN is considered to be OK (in this case, we reset the number of attempts). Otherwise, we reject it and increase the number of trials. We then wait a random time period (representing the delay of the operation), after which an internal transition function is triggered. If the PIN was correct, the output function will issue a request through the *get_amnt* port, asking for the amount to be withdrawn. If the PIN was incorrect, we request the user to try again, but after three attempts the card is rejected. We finally execute the internal transition function, which passivates the model after resetting the counters (if needed).

Once all the atomic models have been defined, we can create a coupled model of the whole system (based on Figure 6.1). The following is a formal specification for such a model:

$$\text{ATM} = \langle X, Y, D, \text{EIC}, \text{EOC}, \text{IC}, \text{SELECT} \rangle \tag{6.1}$$

X = {Card}; Y = {Card_out, Cash_out}
D = {Card_reader, Cash_Dispenser, Authorization}
EIC = {(Self.in, Card_Reader.Card_in)}
EOC = {(Card_Reader.Card_out,Self.Card_out), (Cash_Dispenser.out,Self.Cash_out)}
IC = { (Card_Reader.CardNo_out, Authorization.in), (Authorization.Amnt_out, Cash_Dispenser.in), (Authorization.Eject,Card_Reader.Eject) }
SELECT :{Card_Reader, Cash_Dispenser} = Cash_Dispenser;
{ Authorization, Cash_Dispenser} = Cash_Dispenser

Using CD++ we can represent this formal specification in the graphical notation shown in Figure 6.3. When we export this graphical model, we obtain the text specification shown in Figure 6.4, which will be the one we can use to execute the model.

Figure 6.1, Figure 6.3, and Figure 6.4 represent different ways of representing the coupled model (and they are equivalent, as all of them were built based on DEVS formal specifications for coupled models, i.e., the ATM coupled model defined earlier). The following figure shows the simulation

PIN Verifier = <X, Y, S, δ_{int}, δ_{ext}, λ, ta>
X = {PIN | PIN is 3-digit positive Integer }
Y = {Get_Amnt , Get_PIN, Eject}
S = { State, PIN, PIN_OK, No_of_trials}

δ_{ext} (S,e,X) {
  Case State
    Passive:
      Pin = X;
      RandNo = Generate random no.;
      If (RandNo <= 0.9)
        PIN_OK = TRUE;
        No_of_trials = 0;
      Else
        PIN_OK = FALSE;
        No_of_trials++;

      state = busy;
      ta(busy) = Random;
}

λ (S) {
        If PIN_OK
          Send (GetAmnt_port, 1);
        Else
          If No_of_trials < 3
            Send (GetPIN_port, 1)
          Else
            Send (Eject_port, 1);
}

δ_{int} (PIN_in) {
  if (PIN_OK = 0 and No_of_trials >=3)
      No_of_trials = 0;
    state = passive;
    ta(state) =infinity;
}

```
PINverifier::PINverifier( const string
&name ): Atomic( name ),
pin_in( addInputPort( "pin_in" ) ),
get_pin( addOutputPort( "get_pin" ) ),
get_amnt( addOutputPort("get_amnt") ),
eject( addOutputPort( "eject" ) ) {}

Model &PINverifier::externalFunction (
const ExternalMessage &msg ) {
 if ( this->state() == passive) {
    pin = (int) msg.value();
    randnumber = rand() / RAND_MAX;
    if ( randnumber <= 0.9) {
        PIN_OK = 1;
        no_of_trials = 0;    }
    else
        {
        PIN_OK = 0;
        no_of_trials++;
        }
   }

   holdIn(active,Time(static_cast<float>
     ( fabs( distribution().get() ) )));
}

Model &PINverifier::outputFunction( const
InternalMessage &msg ) {
if (PIN_OK)
  sendOutput(msg.time(), get_amnt, 1);
 else
  if ( no_of_trials < 3)
    sendOutput( msg.time(), get_pin, 1 );
  else
    sendOutput( msg.time(), eject, 1 );
  }
}


Model &PINverifier::internalFunction(
const InternalMessage & ) {
  if ( PIN_OK == 0 && no_of_trials >= 3)
            no_of_trials = 0;
  passivate();
}
```

FIGURE 6.2 Definition of the pin verification atomic model.

results for this model, in which we consider a user arriving at time 10:000. Figure 6.4 shows a detailed log with the activities at the top level. When the simulation starts, we receive an input (X) through the *in* port in the *top* model, which is routed to the *card_in* port of the *cardreader* model (following the coupled model specification in Figure 6.4). The *cardreader* then schedules an internal transition in `01:987` (D), which is triggered at `11:987` (*). At this time, we execute the output function (Y), which will return a card number (697 in port *cardno_out*) and the internal transition function that will passivate the model ("…" represents infinity). The *top* model will then get the card number and, according to the coupling scheme, it will forward it to `in@auth`. Following the hierarchical coupling scheme, this message will be converted into an input to the *userface* model. This model will verify the card and will generate an output/internal transition at `13:579`, when a PIN will be generated (549). This information is then sent to the authorization submodel through `pin _ in@pinver`. With this information, we activate the pin verification submodel (defined in

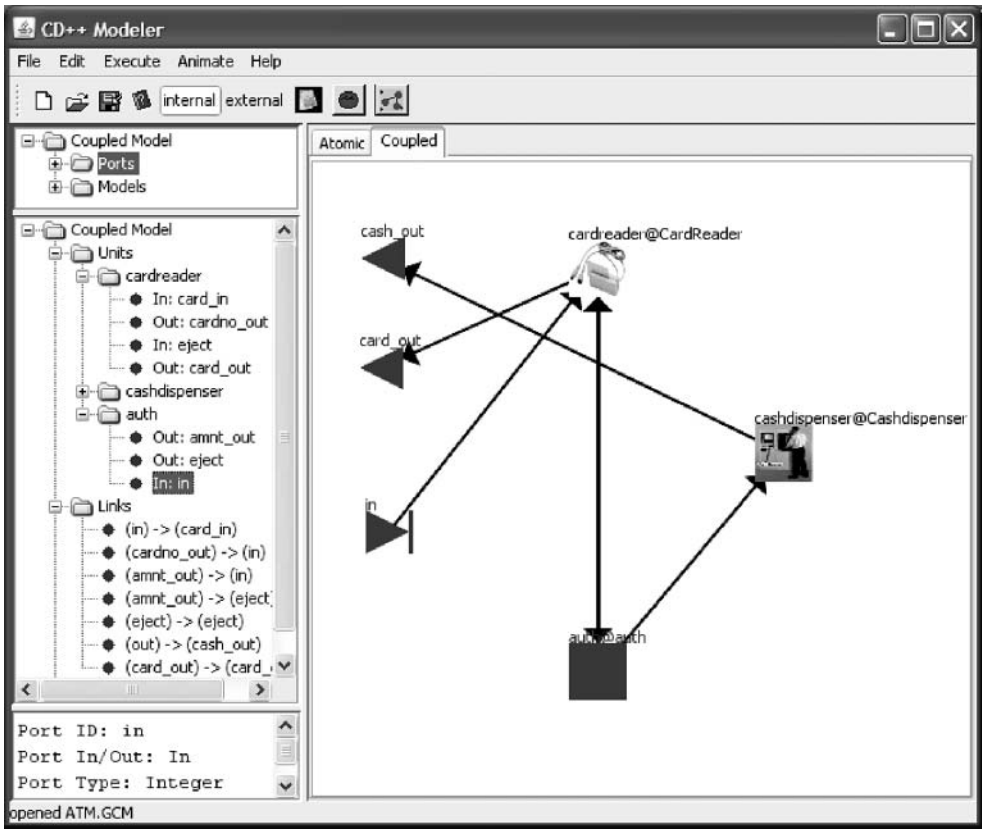


FIGURE 6.3 Structure of the ATM model in CD++. (From Kidisyuk, K., and G. Wainer. 2007. *Proceedings of DEVS Symposium 2007,* Norfolk, VA, and Cidre, J. I. 2006. A Web-based interface for the CD++Modeler toolkit. MCS thesis, Universidad de Buenos Aires, Argentina.)

```
[top]
components : cardreader@CardReader cashdispenser@Cashdispenser auth
out : cash_out card_out
in : in
Link : in card_in@cardreader
Link : cardno_out@cardreader in@auth
Link : amnt_out@auth in@cashdispenser
Link : amnt_out@auth eject@cardreader
Link : eject@auth eject@cardreader
Link : out@cashdispenser cash_out
Link : card_out@cardreader card_out

[auth]
components : balancver@Balanceverifier userface@UserInterface pinver@PINverifier
out : amnt_out eject
in : in
Link : in cardno@userface
Link : get_pin@pinver get_pin@userface
Link : pin_out@userface pin_in@pinver
Link : get_amnt@pinver get_amnt@userface
Link : amnt_out@userface amnt_in@balancver
Link : get_amnt_out@balancver get_amnt@userface
Link : amnt_out@balancver amnt_out
```

FIGURE 6.4 Structure of the ATM model in CD++ (text definition).

Figure 6.2). If we follow the code for this atomic model, we see that the number 549 received through the *pin_in* port is saved in the *pin* variable. Then we decide if it is accepted or rejected at random, and we generate an internal transition after `01:915 s`. At this point, we activate the model's output function (which transmits the number 1 through the port *get_amt*, as we can see in the model's definition), and the internal transition passivates the model.

Exercise 6.1

Explain in detail the rest of the execution of the ATM simulation presented in Figure 6.6.

Exercise 6.2

Modify the PIN verification mechanism. In this case, instead of doing it at random, define a database of PINs (in a file or in an array in memory) and use it to check if the PIN is correct.

Exercise 6.3

Modify the model in Exercise 6.2 to change the mechanism for balance verification. Create a database (in a file or in an array) where each account is associated with the available balance. Modify the cash dispenser model to update the database according to the amount of money given back to the customer.

Exercise 6.4

Modify the coupled model to be flat (i.e., all the models are at the same level).

Exercise 6.5

Modify the model in Exercise 6.3 to include two new submodels: one to check and display the current balance of the account and another to allow customers to deposit funds.

Figure 6.6 shows the simulation results of the model using CD++Modeler. The first figure shows a detailed execution of the authorization coupled model. As we can see, the input/output trajectories follow those presented earlier in Figure 6.5.

The model shows only the input/output ports of the authorization submodel. Initially, we receive a card number (697) and a PIN (549), as explained earlier. Because the PIN is OK, the *get_amnt* port returns True (1), and we request the amount to be withdrawn (981), which is delivered through the *amnt_out* port shortly thereafter. We see that a new card/PIN is entered afterwards (at 00:32/00:37), and the user requests different sums that are rejected, until at 1:11 the user's request is granted.

6.3 A WATER RESERVOIR CONTROLLER FOR A CITY

In this model we intend to model the control system in charge of a reservoir of water for a small city. The structure of the model is presented in Figure 6.7.

The system consists of two separate entities: the water wells and the pump station. This station consists of a number of electrical pumps (two or more) connected to the water reservoir (basically, a large water tank holding a reserve of water for the city). The water comes from the wells into the reservoir and then is pumped into the city water system. An operator can start/stop one or several of the electrical pumps, trying to keep the flow on the outgoing pipe at a certain rate, according to water consumption in the city at that moment and the water available in the reservoir and the wells. Likewise, the pumps' overflow is returned to the wells.

Figure 6.8 shows the definition of the water pump model in CD++, which can be found in *./city_water.zip*. Initially, we define the model I/O ports and initialize a few variables (level, power, start/stop, etc.). We query the nominal flow and the alarm information from the model's definition file. The external transition function reacts to different inputs:

```
X / 10:000 / Root / in / 1 to top
X / 10:000 / top / card_in / 1 to cardreader
D / 10:000 / cardreader / 01:987 to top
* / 11:987 / top to cardreader
Y / 11:987 / cardreader / cardno_out / 697 to top
D / 11:987 / cardreader / ... to top
X / 11:987 / top / in / 697 to auth
X / 11:987 / auth / cardno / 697 to userface
D / 11:987 / userface / 01:592 to auth
D / 11:987 / auth / 01:592 to top
* / 13:579 / top to auth
* / 13:579 / auth to userface
Y / 13:579 / userface / pin_out / 549 to auth
D / 13:579 / userface / ... to auth
X / 13:579 / auth / pin_in / 549 to pinver
D / 13:579 / pinver / 01:915 to auth
* / 15:494 / auth to pinver
Y / 15:494 / pinver / get_amnt / 1 to auth
D / 15:494 / pinver / ... to auth
X / 15:494 / auth / get_amnt / 1 to userface
D / 15:494 / userface / 04:070 to auth
* / 19:564 / auth to userface
Y / 19:564 / userface / amnt_out / 981 to auth
D / 19:564 / userface / ... to auth
X / 19:564 / auth / amnt_in / 981 to balancver
Y / 21:929 / balancver / amnt_out / 981 to auth
D / 21:929 / balancver / ... to auth
Y / 21:929 / auth / amnt_out / 981 to top
D / 21:929 / auth / ... to top
X / 21:929 / top / eject / 981 to cardreader
X / 21:929 / top / in / 981 to cashdispenser(07)
D / 21:929 / cardreader / 02:160 to top
D / 21:929 / cashdispenser(07) / 02:849 to top
* / 24:089 / top to cardreader
Y / 24:089 / cardreader / card_out / 1 to top
D / 24:089 / cardreader / ... to top
Y / 24:089 / top / card_out / 1 to Root
D / 24:089 / top / 00:689 to Root
* / 24:778 / Root to top
* / 24:778 / top to cashdispenser(07)
Y / 24:778 / cashdispenser(07) / out / 981 to top
D / 24:778 / cashdispenser(07) / ... to top
Y / 24:778 / top / cash_out / 981 to Root
D / 24:778 / top / ... to Root
```

FIGURE 6.5 ATM simulation results.

- Power can be turned on and off.
- The current level of the reservoir is received and stored.
- If a low-level value is available, the pumps are stopped (and the model passivates).
- The inputs arriving at the *start_in* port are used to restart a stopped pump. We use the *start* command, and, if the current level is higher than the alarm level, we record the state change and change the flow based on the nominal flow rate.
- When we receive a *stop_in* input, we decrease the flow according to the nominal flow defined by the user. When the internal transition function is triggered, we output the current flow/state, and we schedule a new internal transition function after *active_time*.

Figure 6.9 shows the definition of the atomic model using CD++Modeler. The model has four states: *Idle, vol_supp, vol_pump,* and *level_comp.* Initially, the model is idle and in a passive state (99:99:99:999 represents infinity). At this point, we can receive a water supply from the wells (*q_in* input port) or excess from the pump (*qp_in* input port) and change state accordingly. Ten seconds

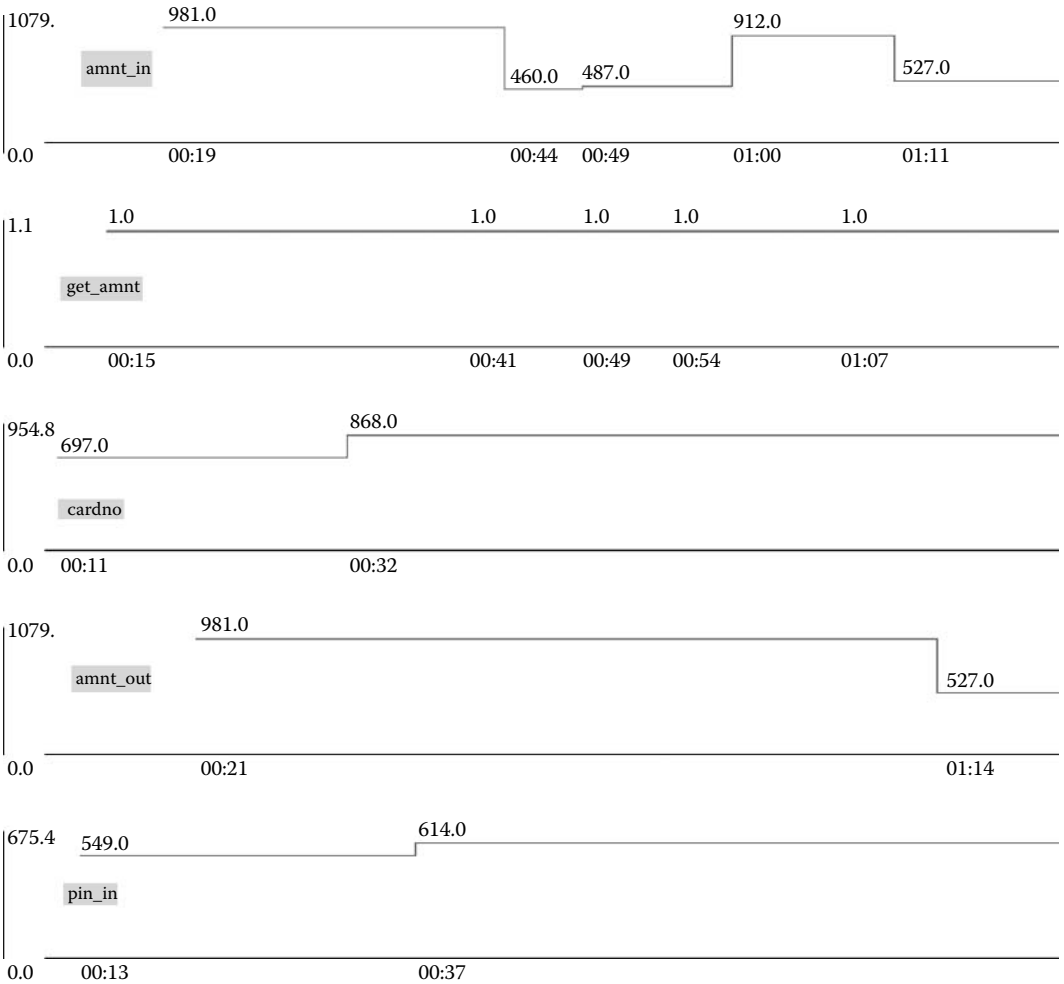


FIGURE 6.6 ATM simulation results: authorization coupled model

after any of these events, we trigger an internal transition function. In both cases, we change to the *level_comp* state, but two different computations are executed by the output functions: if we get an overflow from the pump, we compute the overflow. Otherwise, we need to consider both pump's and well's inputs in the computation.

Figure 6.10 shows the execution of the model. The external events turn on the pump submodel and then initiate the wells. At this point, we start seeing activity. At 00:02:33:001, the wells generate a flow of 475 L/s (the maximum is 500 L/s). At the same time, we can see the second pump working. After that, we can see the level of the reservoir going up and the second pump being activated. At 00:04:03:001, we can see that when *flow_out* is below the limit, the pump is switched off, and the level of the reservoir reduces (but is refilled by the output from *pump1*). Figure 6.11 shows the execution of such a model using CD++Modeler.

We can see that the initial level of the reservoir goes down slowly, until we turn on the pumps. At this point, the level goes up and down according to the flow level and the status of the controller.

Exercise 6.6

Change the model to include three pumps and analyze the simulation results.

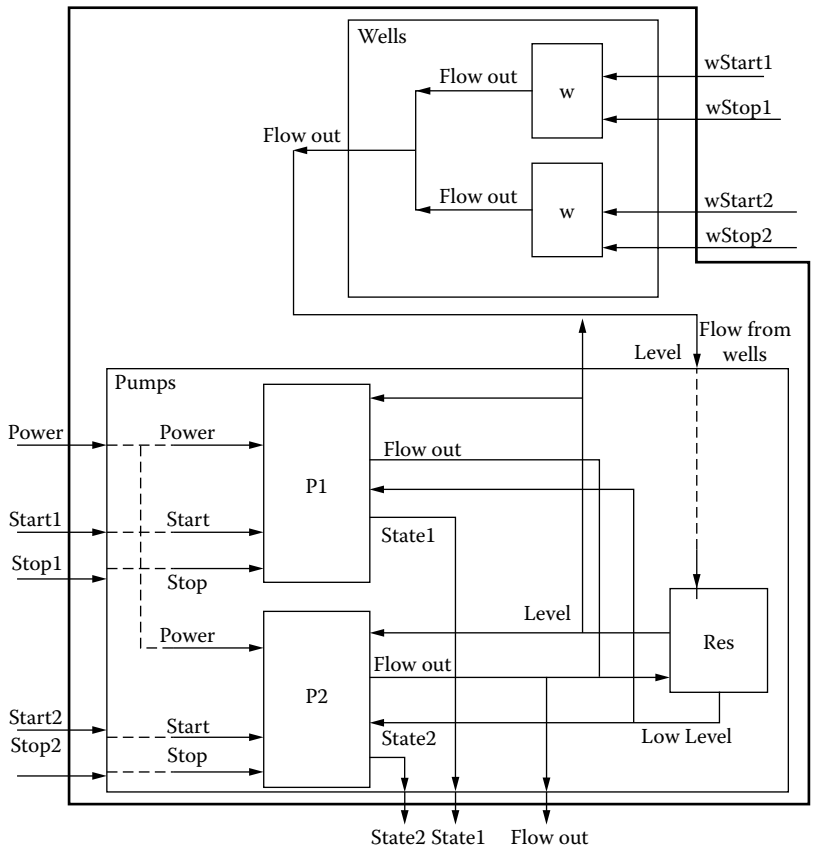


FIGURE 6.7 A water reservoir for a city.

EXERCISE 6.7

Change the external parameters to increase the flow of water and the limits to detect alarms and execute the simulation again. Analyze the results.

6.4 RADAR-BASED TRAFFIC LIGHT

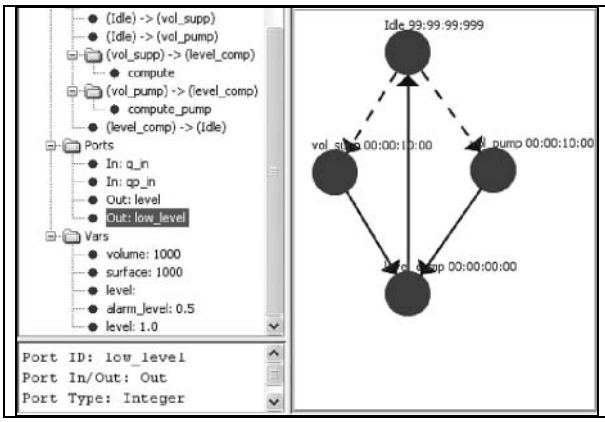
The model presented in this section (found in *./RadarModeler.zip* and presented in references 4 and 5) defines the behavior of a traffic light with radar and a digital camera that takes pictures of vehicles speeding or crossing the intersection when the light is red. We presented the coupled model structure for this application in Figure 4.29(a) in Chapter 4. The model is organized in four components. *Control* is an atomic model that controls the activities of the system; according to the current value of the traffic light, it will trigger an output directed to the *Camera* atomic model to take a photo. The *TrafficLight* cycles among red, yellow, and green lights, while the *IO* model is in charge of communication tasks.

The top-level coupled model uses the following I/O ports:

- *Lane* (in) detects the presence of a car over the pedestrian zone at the corner of the street.
- *Radar* (in) represents a sensor detecting a car over the maximum speed. This sensor will be activated by the traffic light atomic model when the lane sensor detects a car in the pedestrian zone and the traffic light is red.

```
Pump::Pump( const string &name ): Atomic( name )
, start_in( addInputPort( "start_in" ) ), stop_in( addInputPort( "stop_in" ) )
, level_in( addInputPort( "level_in" ) ), low_level_in( addInputPort( "low_level_in" ) )
, power_in( addInputPort( "power_in" ) ), state_out( addOutputPort( "state_out" ) )
, flow_out( addOutputPort( "flow_out" ) ), wflow_out( addOutputPort( "wflow_out" ) )
, preparationTime(0,0,10,0), activeTime(0,0,10,0), level(2.3), state(0), power(1)
, start_command(0), stop_command(0){
 string nom_debit( MainSimulator::Instance().getParameter( description(), "nominal_flow" ) ) ;

  if (nom_debit != "") nominal_flow = str2float( nom_debit ) ;
 string al_niv( MainSimulator::Instance().getParameter( description(), "alarm_level" )  ) ;

  if (al_niv != "")alarm_level = str2float( al_niv );
}

Model &Pump::externalFunction( const ExternalMessage &msg ) {
  if ( msg.port() == power_in )
        this->power = static_cast< short >( msg.value()) ;

  if ( msg.port() == level_in )
        this->level = static_cast< float >( msg.value()) ;

  if ( msg.port() == low_level_in )
        this->low_level = static_cast< short >( msg.value()) ;

  if (this->low_level) {    // stop pumps
        this->state =this->flow = 0.0;
        this->passivate() ;
  }
  else
        this->holdIn( active, this->preparationTime ) ;

  if ( msg.port() == start_in ) {
        this->start_command = static_cast< short >( msg.value()) ;

        if ( this->level > this->alarm_level )
                this->state = this->start_command & this->power;

        this->start_command = 0;      // reset start command

        this->flow += this->nominal_flow * this->state;

        this->holdIn( active, this->preparationTime ) ;
  }

  if ( msg.port() == stop_in )        {
        this->stop_command = static_cast< short >( msg.value()) ;

        if (this->state)
                this->flow -= this->nominal_flow;

        this->state =  this->stop_command = 0;    // reset stop command

        this->holdIn( active, this->preparationTime ) ;
  }
}

Model &Pump::internalFunction( const InternalMessage & ) {
  if (this->state)
        this->holdIn( active, this->activeTime ) ;   // Pumping
  else
        this->passivate() ;
}

Model &Pump::outputFunction( const InternalMessage &msg )  {
  this->sendOutput( msg.time(), this->flow_out, this->flow ) ;


  this->sendOutput( msg.time(), this->state_out, this->state ) ;
}
```

FIGURE 6.8 Definition of water pump model in CD++.




```
[reservoir]
in: q_in qp_in
out: level low_level
state: Idle vol_supp vol_pump level_comp
initial : Idle
var: vol surface level alarm_level level
ext: Idle vol_supp q_in?250
ext: Idle vol_pump qp_in?200
int: vol_supp level_comp level!q_in +
            qp_in level!1 {compute;}
int: vol_pump level_comp {compute_pump;}
int: level_comp Idle
Idle:99:99:99:999
vol_supp:00:00:10:00
vol_pump:00:00:10:00
level_comp:00:00:00:00
vol:1000
surface:1000
alarm_level:0.5
level:1.0
```

FIGURE 6.9 A model of the reservoir: graphical representation.

```
00:00:00:50 power_in 1
00:00:02:00 wstart_in1 1
00:00:30:00 start_in1 1
00:01:00:00 start_in2 1
00:02:02:00 wstart_in2 1
00:04:00:00 stop_in2 1
```

```
00:02:33:001 flow_out 475
00:02:33:001 state_out2 1
00:02:34:001 level 0.9771
00:02:43:001 level 0.98085
00:02:53:001 flow_out 475
00:02:53:001 state_out1 1
. . .
00:04:03:001 level 0.97285
00:04:13:001 flow_out 250
00:04:13:001 state_out1 1
00:04:13:001 flow_out 250
00:04:13:001 state_out2 0
00:04:14:001 level 0.9716
00:04:23:001 level 0.97535
00:04:33:001 flow_out 250
...
```

FIGURE 6.10 Executing the model of the reservoir.

- *Ext* (in) issues output commands (defined to be transmitted to the camera and traffic light).
- *Count* (in) gives information about the available memory (in number of pictures left).
- *Central* (out) alerts Central when there is not enough memory left.

Based on the graphical model introduced in Figure 4.29(a) of Chapter 4, CD++Modeler generates the text specification of the coupled model depicted in Figure 6.12. This specification represents the coupled model as in the original figure, and the atomic models are created as *cdd* files. Each of these files contains the description of the atomic model using DEVS-graphs (presented in Chapter 2). For instance, Figure 6.13 represents the model's behavior for the control atomic model using such notation.

This atomic model starts in the *active* state. If the *radar* port receives a message with high speed (or we detect over the lane when the traffic light is red), then the model switches to the *speed* state. In this state, we schedule an internal transition function instructing the camera to take a picture (*photo!1*); this changes the model to the *click* state (which could result in taking a picture and becoming *active* again. In this state, we first check to see if there is memory available; if not, we switch to the *void* state and the traffic light detector becomes *inactive* (in this state, we lose any infractions). In this process, the number of remaining shots is zero, so we send a message through the *empty* port and wait for the memory to be emptied on the port *full*. When we receive more

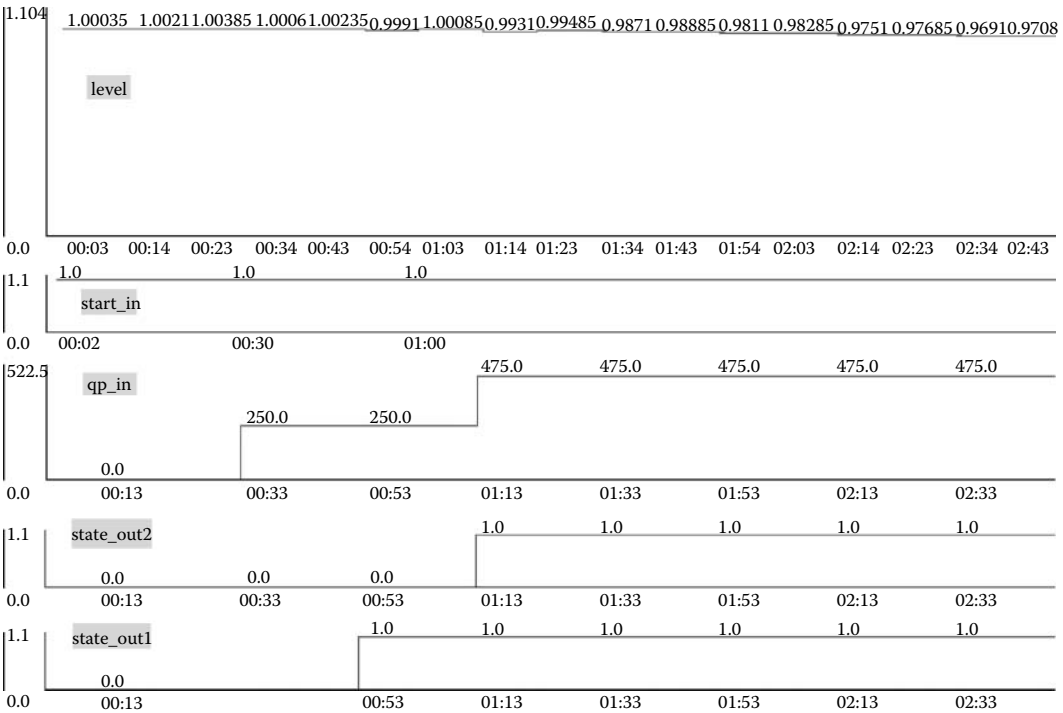


FIGURE 6.11 Executing the model of the reservoir (graphical).

```
[top]
components : trafficlight@GGad control@GGad camera@GGad IO@GGad
out : central
in : lane radar ext count
Link : lane lane@control
Link : radar radar@control
Link : light@trafficlight trafficlight@control
Link : photo@control take@camera
Link : ready@camera ready@control
Link : count count@IO
Link : empty@control empty@IO
Link : restart@IO full@control
Link : ext extern@IO
Link : central@IO central

[TrafficLight] source : trafficlight.cdd
[control]      source : control.cdd
[camera]       source : camera.cdd
[IO]           source : IO.cdd
```

FIGURE 6.12 Text specification of the radar coupled model.

memory (*Any(full)?1*), the control model becomes *active* and restarts the cycle. A second condition to take a picture is when we detect a vehicle in the lane with a red traffic light (*And(Equal(Lane, 1), Equal(light, 0))*). In this case, we switch to the *redlight* state and give the order to take a picture (changing to the *click* state). Every time the traffic light cycles, the model updates the value of the variable *light*. The textual representation is the equivalent of the graphical model (Figure 6.14).

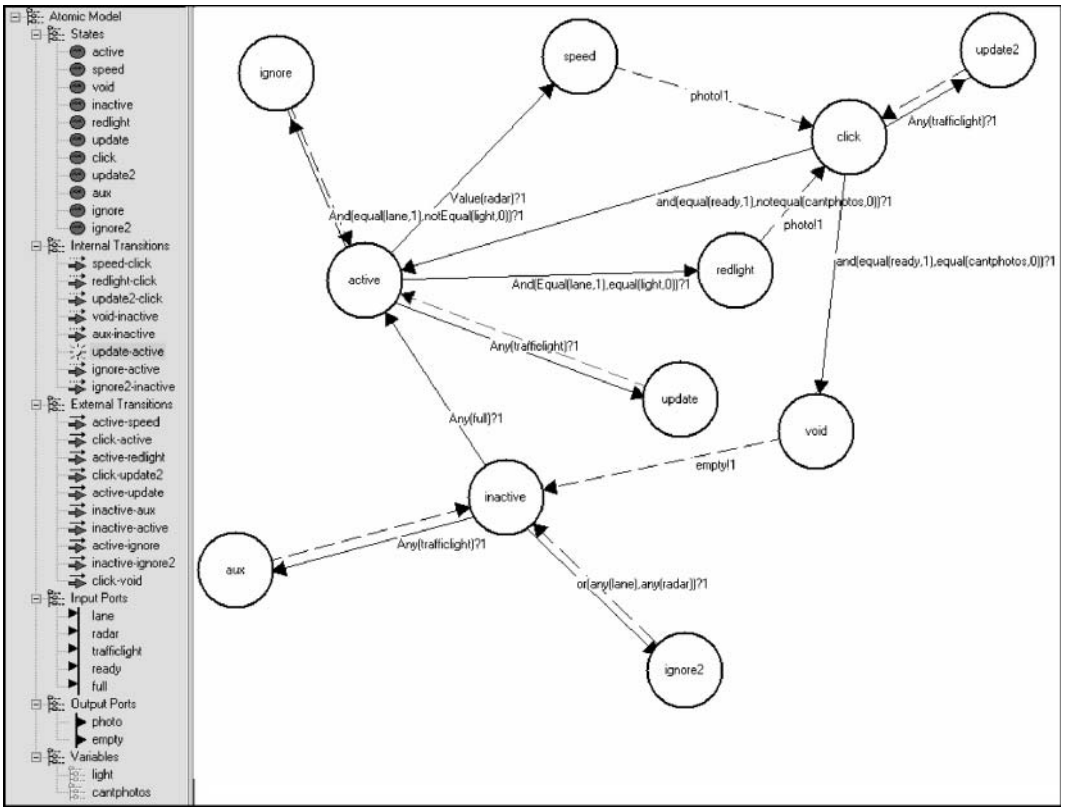


FIGURE 6.13 DEVS graph for the control model. (From Christen, G., A. Dobniewski, and G. Wainer. 2004. *Proceedings of MGA, Advanced Simulation Technologies Conference 2004,* Arlington, VA.)

Exercise 6.8

Change the control model so that now it also detects vehicles running through a yellow or red light.

The remaining submodels were also defined using this notation (for a detailed description, see references 4–6):

- **Camera** is in *StdBy* state until it receives the shot instruction to take a value on the port *take*. The time spent storing the picture is represented by the time advance function associated with the state *prepare*. When the picture is stored, a message is emitted in the port *ready.*
- **Traffic light** is a simple timed cycle between each of the three states of the traffic lights. Each change of lights is announced on the port *light*. There are no input ports.
- **IO** takes the responsibility of communicating with the traffic central, in order to inform the lack of memory available for new pictures and downloading them. If a signal is received on port *empty*, it sends the information through the port *central* to communicate the lack of memory available for new pictures.

After defining each submodel, we can execute the simulation in order to analyze the behavior of this system. We log the following information:

```
[control]
in: lane radar TrafficLight ready full
out: photo empty
var: light quantity_of_pictures
state: active speed void inactive redlight update click update2 aux ignore ignore2
initial: active
int: speed click photo!1 {quantity_of_pictures = minus(quantity_of_pictures,1);}
int: redlight click photo!1 {quantity_of_pictures = minus(quantity_of_pictures,1);}
int: update2 click
int: void inactive empty!1
int: aux inactive
int: update active
int: ignore active
int: ignore2 inactive
ext: active speed Value(radar)?1
ext: click active and(equal(ready,1),notequal(quantity_of_pictures,0))?1
ext: active redlight And(Equal(lane,1),equal(light,0))?1
ext: click update2 Any(TrafficLight)?1 {light = TrafficLight;}
ext: active update Any(TrafficLight)?1 {light = TrafficLight;}
ext: inactive aux Any(TrafficLight)?1 {light = TrafficLight;}
ext: inactive active Any(full)?1 {quantity_of_pictures = full;}
ext: active ignore And(equal(lane,1),notEqual(light,0))?1
ext: inactive ignore2 or(any(lane),any(radar))?1
ext: click void and(equal(ready,1),equal(quantity_of_pictures,0))?1
active:infinite
speed:00:00:00:00
void:00:00:00:00
inactive:infinite
redlight:00:00:00:00
update:00:00:00:00
click:infinite
update2:00:00:00:00
aux:00:00:00:00
ignore:00:00:00:00
ignore2:00:00:00:00
light:0
quantity_of_pictures:2
```

FIGURE 6.14 Text specification of the control atomic model.

- Input Message (type : ?) is recorded every time a message is received on an input port: {port receiving the message, value received}
- Output Message (type : O) is recorded each time the output function is executed: {output port, value}
- Internal Transition Function (type : I): {init, final, {(var-before,val),(var-after, val) ,...} }.
- External Transition Function (type : E): init state, final state, {(var1, value), (var2, value), ... }

The following shows the simulation results of this model:

```
Time          Port    Value
00:00:02:00   lane    1
00:00:04:00   radar   1
00:00:12:00   lane    1
00:00:15:00   lane    1
00:00:19:00   lane    1
00:00:24:00   radar   1
00:00:26:00   ext     1
00:00:26:00   count   10
. . .
```

The following log files show the execution of the four atomic models that compose the coupled system. By analyzing these log files, we can study the models' behaviors and their interactions.

Control

```
C 00:00:00:000 : active , (quantity_of_pictures=2) (light=0)
? 00:00:02:000 : lane , 1
E 00:00:02:000 : active ,redl(quantity_of_pictures=2) (light=0)
O 00:00:02:000 : photo , 1
I 00:00:02:000 : redl,click (quantity_of_pictures=1),(light=0)
? 00:00:04:000 : radar , 1
? 00:00:04:000 : ready , 1
E 00:00:04:000 : click, active(quantity_of_pictures=1),(light=0)
? 00:00:05:000 : trafficl, 2
E 00:00:05:000 : active,update(quantity_of_pictures=1),(light=2)
I 00:00:05:000 : update,active(quantity_of_pictures=1),(light=2)
? 00:00:06:000 : trafficl , 0
E 00:00:06:000 : active, update(quantity_of_pictures=1)(light=0)
I 00:00:06:000 : update,active(quantity_of_pictures=1),(light=0)
...
? 00:00:19:000 : lane , 1
E 00:00:19:000 : active,redl(quantity_of_pictures=1),(light=0)
O 00:00:19:000 : photo , 1
I 00:00:19:000 : redl , click (quantity_of_pictures=0),(light=0)
? 00:00:21:000 : ready , 1
E 00:00:21:000 : click , void (quantity_of_pictures=0),(light=0)
O 00:00:21:000 : empty , 1
I 00:00:21:000 : void,inact(quantity_of_pictures=0),(light=0)
? 00:00:22:000 : trafficl , 1
E 00:00:22:000 : inact,aux(quantity_of_pictures=0),(light=1)
I 00:00:22:000 : aux,inact(quantity_of_pictures=0),(light=1)
? 00:00:24:000 : radar , 1
E 00:00:24:000 : inact,ignore2(quantity_of_pictures=0) (light=1)
I 00:00:24:000 : ignore2,inact(quantity_of_pictures=0) (light=1)
? 00:00:26:000 : full , 10
E 00:00:26:000 : inact,active(quantity_of_pictures=10) (light=1)
...
```

Camera

```
C 00:00:00:000 : stdby ,
? 00:00:02:000 : take , 1
E 00:00:02:000 : stdby , run
I 00:00:02:000 : run , prepare
O 00:00:04:000 : ready , 1
I 00:00:04:000 : prepare , stdby
? 00:00:19:000 : take , 1
E 00:00:19:000 : stdby , run
I 00:00:19:000 : run , prepare
O 00:00:21:000 : ready , 1
```

```
I 00:00:21:000 : prepare , stdby
...
```

IO

```
C 00:00:00:000 : receive , (quantity_of_pictures=0)
? 00:00:21:000 : empty , 1
E 00:00:21:000 : receive , send (quantity_of_pictures=0)
O 00:00:21:000 : central , 1
I 00:00:21:000 : send , receive (quantity_of_pictures=0)
? 00:00:26:000 : count , 10
? 00:00:26:000 : extern , 1
E 00:00:26:000 : receive , resume (quantity_of_pictures=10)
...
```

TrafficLight

```
C 00:00:00:000 : green ,
O 00:00:05:000 : light , 2
I 00:00:05:000 : green , yellow
O 00:00:06:000 : light , 0
I 00:00:06:000 : yellow , red
O 00:00:11:000 : light , 1
I 00:00:11:000 : red , green
O 00:00:16:000 : light , 2
I 00:00:16:000 : green , yellow
O 00:00:17:000 : light , 0
I 00:00:17:000 : yellow , red
O 00:00:22:000 : light , 1
I 00:00:22:000 : red , green
...
```

The control model starts in state *active*. There are two photos left (*quantity_of_pictures* = 2), and the traffic light is red (*light* = 0; the light is encoded as red = 0, green = 1, and yellow = 2). The camera starts in state *stdby,* which means it is ready to take a shot.

A message is received after 2 s on input port *lane*, with value 1. This event fires the external transition function. A suitable transition is found by the simulator from state *active* to state *redlight.* This state represents the fact that a car has crossed the lane when the light was red. An instantaneous (i.e., ta = 0) internal transition is fired to request the camera to take a photo. The message is generated by the output function through the port *photo.* This port is connected at the coupled model level with the atomic model of the camera. When it receives a message on port *take* (from the *control* model), it simulates the process of taking the photo and also of advancing to the next memory position to store a new photo. After evaluating the output function, an internal transition is fired from state *redlight* to *click,* decreasing the count of remaining photos. A while later a message from the radar gives an alert of a car exceeding the speed limit at the same time that the camera ends the previous request. Because the camera was not ready, the car must be ignored. After that, the control returns to *active,* so it can process new infractions. The traffic light sends its first message to indicate a light change. The control updates its internal variable *TrafficLight* to match the status received. At 00:00:19:000 we have a real infraction: a car over the lane while the light is red. The

control model commands the *camera* model to get a shot of the car. Then the model goes to state *click* to wait for the camera to complete the take.

At 00:00:21:000 the camera sends its *ready* message, and the controller decreases the count of photos and discovers that no more are available. The controller changes to the *void* state, sending a message on the port *empty.* This request is received by the model *IO,* which on time 00:00:26 warns on port *full* that the pictures have been stored and there is enough capacity to store 10 new photos. In that period, the controller was *inactive,* processing only messages incoming from the traffic lights.

The *TrafficLight* model just cycles over the three lights. Note that the yellow light has a 1-s time advance, while the other two have a 5-s time advance.

The *IO* model is activated when the control model wants to request memory from the central. The output port *central* is connected at the top coupled model level to the exterior of the system. *IO* receives a message from *count* with space for 10 new photos, so it passes this information control model through output port *restart,* which is connected to input port *full.*

Exercise 6.9

Create a new model that will generate speeding and lane cars at random and connect it to the top-level coupled model. Execute the simulation and analyze the results.

Exercise 6.10

Change the frequency of the traffic light and repeat the previous exercises.

6.5 SUMMARY

In this chapter we have given detailed explanations on how to use CD++ to build DEVS models. We first explained the definition of a model of an ATM machine, focusing on the definition of the atomic models in C++ and showing the definition of the coupled model using both text and graphical interfaces. We showed how models execute, showing both graphical and text logs. We then presented a model of a reservoir for a city, including a state-based model for the reservoir, and the definition of the pumps in C++. Finally, we introduced a complete model based on DEVS-graphs notation, representing a controller for a traffic light with control for speeding.

These examples show the basic ideas needed to build a DEVS model, and they will be used in later chapters to analyze more advanced models with applications in varied fields.

REFERENCES

1. Kidisyuk, K., and G. Wainer. 2007. CD++Builder: A toolkit to develop DEVS models. *Proceedings of DEVS Symposium 2007,* Norfolk, VA.
2. Cidre, J. I. 2006. A Web-based interface for the CD++Modeler toolkit. MCS thesis, Universidad de Buenos Aires, Argentina.
3. Kidisyuk, K., and G. Wainer. 2007. CD++Modeler: A graphical viewer for DEVS models. Technical report SCE-017, Ottawa, Carleton University, Canada, 2007 (in *Poster Sessions, SpringSim 2008,* Ottawa, Canada).
4. Christen, G., A. Dobniewski, and G. Wainer. 2001. Defining DEVS models with the CD++ toolkit. *Proceedings of European Simulation Symposium,* Marseilles, France.
5. Christen, G., A. Dobniewski, and G. Wainer. 2004. Modeling state-based DEVS models CD++. *Proceedings of MGA, Advanced Simulation Technologies Conference 2004,* Arlington, VA.
6. Christen, G., and A. Dobniewski. 2003. Extending the CD++ toolkit to define DEVS graphs. MSc thesis, Computer Science Dept., Universidad de Buenos Aires, Argentina.

7 Defining Varied Modeling Techniques Using DEVS

7.1 INTRODUCTION

DEVS has been shown to be a general formalism such that several other existing ones can be expressed as DEVS models. As discussed in numerous references (for instance, Vangheluwe [1]), varied formalisms—like FSM, cellular automata, Petri nets, Bond Graphs, event scheduling, and state charts (among others)—have been transformed into DEVS models. As discussed in Chapter 1, DEVS allows representation of all the systems whose input/output behavior can be described by sequences of events. The generality of DEVS is derived from the fact that it permits modeling systems with a set of infinite possible states and where the new state after an event arrival may depend on the (continuous) time elapsed in the previous state. Consequently, a modeler can express different properties in the formalism of choice and use DEVS hierarchical coupling as the integration mechanism. In this chapter we will introduce the implementation of different modeling techniques as DEVS using CD++ to show how this can be implemented.

7.2 FINITE STATE MACHINES

Finite state machines (FSMs), introduced in Chapter 1, are popular for modeling systems in a variety of areas such as software design and digital logic design. In this section we describe an implementation of FSM Moore machines using CD++ (thus, FSM means a Moore FSM from here on). The construction of a library of atomic models to represent FSM has been straightforward since it was shown in Zeigler [2] and Zeigler and Vahie [3] that FSMs can be embedded in DEVS because any discrete event behavior can be expressed as a DEVS model.

Every FSM consists of a finite number of states linked through the FSM transition functions. In order to define this behavior, we created a generic *state* atomic model; then the FSM is defined as a coupled model connecting those *state* atomic models. This model, found in *./fsm2.zip* and presented in Zheng and Wainer [4], is called *MooreState*, and it is defined as shown in Figure 7.1.

The *State* model is a unique global *stateCode* (assigned to each state in an FSM) and a given *stateValue* (an optional parameter whose value is transmitted through the *stateOut* port). The *eventIn* port receives numbers encoded that represent external events. The *transitionOut* port informs the next transition to be executed.

Every time a state receives an event, we first check to see if it is a legal event (i.e., if it is listed in the *events* array). If we receive the event *transitionIn,* we save the code of the event. In both cases, we schedule an instantaneous internal transition. The output function will determine if the input was an event, and it uses the *stateOut* port to transmit the current *stateValue* or the *transitionOut* port to send out the next transition (represented by the *tempNextState* signal), which will transmit to all the *transitionIn* ports in the FSM (announcing which state is active in the next step). A state becomes active if the encoded number received from the *transitionIn* port is the same as its *stateCode*. Therefore, only one state is active at a time in one FSM.

FSM models could be easily created as coupled DEVS models by connecting a number of instances of *MooreState*. There are a few basic rules in creating an FSM by connecting the states:

```
MooreState::MooreState( const string &name ): Atomic( name )
, transitionIn( addInputPort( "transitionIn" ) ), eventIn( addInputPort( "eventIn" ) )
, stateOut( addOutputPort( "stateOut" ) ), transitionOut( addOutputPort( "transitionOut" ) )
{
  isEvent = false;

  string code(MainSimulator::Instance().getParameter( description(), "StateCode" ) );
  stateCode = str2Int(code);

  string value(MainSimulator::Instance().getParameter( description(), "StateValue" ) );
  if (value != "" )    {
        stateValue = str2Int(value);
        hasStateValue = true;
  }
  else
  {     stateValue = stateCode;
        hasStateValue = false;
  }

 string num(MainSimulator::Instance().getParameter( description(), "NumberOfTransitions" ) ) ;

  if (num != "" )
        numOfTransitions = str2Int(num);
  else
        numOfTransitions = 1;
  events = new int[numOfTransitions];
}

Model &MooreState::externalFunction( const ExternalMessage &msg ) {
   if (msg.port() == eventIn ) {
        tempEvent = static_cast <int> (msg.value());
        bool done = false;

        for (int i = 1; i <=numOfTransitions && !done; i++)   {
                int transCode = events[i-1];

                if ( tempEvent == transCode/100 )     {
                        done = true;
                        tempNextState = transCode % 100;
                }
          }

         if (done) isEvent = true;
         holdIn(active, Time::Zero);
      }
  else
      if (msg.port() == transitionIn)  {
         tempEvent = static_cast <int> (msg.value());
         if (tempEvent == stateCode)  {
                isEvent = false;
                holdIn(passive, Time::Zero);
        }
     }
}

Model &MooreState::internalFunction( const InternalMessage & ){
  if (isEvent)
                passivate();
        else
                holdIn(active, Time::Inf);
}

Model &MooreState::outputFunction( const InternalMessage &msg ) {
  if (isEvent)
        sendOutput(msg.time(), transitionOut, tempNextState);
  else
        if (hasStateValue) sendOutput( msg.time(), stateOut, stateValue ) ;
  }
}
```

FIGURE 7.1 Moore's machine state.

- All the *transitionOut* and *transitionIn* ports should be connected together inside the FSM. That is, in the view of an FSM, these ports do not communicate with the external environment, and they are not visible to the outside world.
- All the *eventIn* ports of states should be connected to an input port of an FSM.
- All the *errorOut* ports of states are connected together as an output port of an FSM.
- Each *stateOut* port of state could either be connected to an individual output port of an FSM or connected together as one output port of an FSM.
- All the states in an FSM are encoded as integer numbers (*stateCode*) during simulation. The state with the *stateCode* 0 is the initial state in the FSM.
- All the legal events for this FSM are also encoded as integer numbers during simulation.

Exercise 7.1

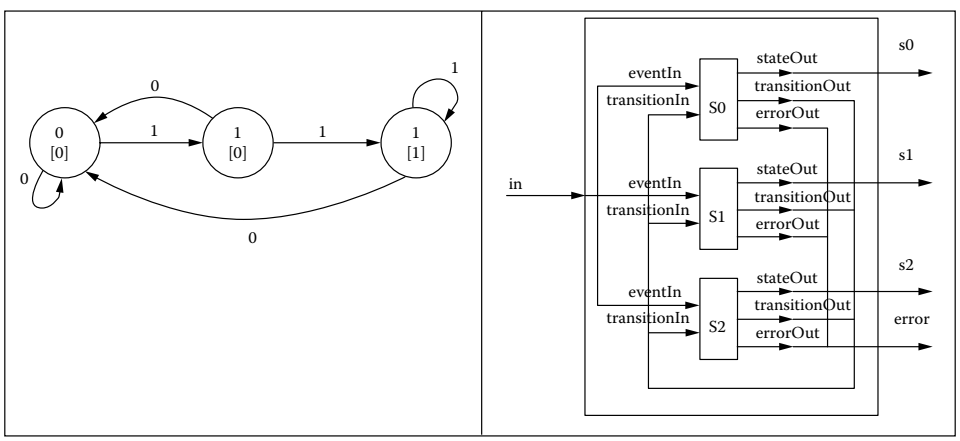
Create a new *atomic* model. In this case, the model can be in one of two states: active or passive. Each state uses a unique identifier. When a model passivates an input, it becomes active. After a fixed delay, it generates an output (indicating the current identifier). The FSM is created by linking different instances of the *State* model and giving initial identifiers to each of the states.

Exercise 7.2

Use CD++Modeler and create a coupled model with a graphical representation to visualize states/transitions for the exercises in this chapter. In this graphical model, each state transition will be observed as an animation in the coupled model definition.

Figure 7.2 shows a simple Moore machine that sends an output of 1 whenever its input string has at least two 1s in sequence. Otherwise, value 0 is sent out. Three states are defined in this FSM. The state machine should send out 0 at state *0* or *1* and 1 at state *2*. Figure 7.2(b) shows a sketch of the definition of this FSM as a DEVS coupled model. Each of the states is represented using the *MooreState* atomic model. *S0, S1,* and *S2* represent the three states in the state machine. The *in* port receives the external events, and ports *s1, s2,* and *s3* generate outputs. The port *error* is used to check whether there are illegal events received.

Using this information, we can define this FSM as shown in Figure 7.3. We can see the definition of the three states (each of them including initialization parameters). The *StateCode* defines the unique ID for the state. For instance, *s0* has a state code of 0 and a *StateValue* of 0 (which is the



(a) (b)

FIGURE 7.2 (a) A simple FSM; (b) coupled model definition of the FSM.

```
components : s0@moorestate s1@moorestate s2@moorestate
out : s0 s1 s2
in : in
Link : in eventIn@s0
Link : in eventIn@s1
Link : in eventIn@s2
Link : transitionOut@s0 transitionIn@s0
Link : transitionOut@s0 transitionIn@s1
Link : transitionOut@s1 transitionIn@s0
Link : transitionOut@s1 transitionIn@s2
Link : transitionOut@s2 transitionIn@s2
Link : transitionOut@s2 transitionIn@s0
Link : stateOut@s0 s0
Link : stateOut@s1 s1
Link : stateOut@s2 s2

[s0]
StateCode : 0
StateValue : 0
NumberOfTransitions : 2
Transition01 : 0->0
Transition02 : 1->1

[s1]
StateCode : 1
StateValue : 0
NumberOfTransitions : 2
Transition01 : 0->0
Transition02 : 1->2

[s2]
StateCode : 2
StateValue : 1
NumberOfTransitions : 2
Transition01 : 0->0
Transition02 : 1->2
```

FIGURE 7.3 Coupled model definition of the FSM in Figure 7.2 in CD++.

value to be sent as an output). The *NumberOfTransitions* indicates the number of legal transitions in the current state. The number of *TransitionXX* items depends on the *NumberOfTransition*s. For instance, for *s0,* if the event 0 is received, the state with *StateCode* 0 is active next (the self-transition in Figure 7.2). If event 1 is received, the next active state is the one with *StateCode* 1. Figure 7.4 shows the execution of this model when we trigger the following events.

Initially, the model receives a 0 and remains in state *s0* (generating 0 as output). We then receive a 1 that produces a change to *s1* (and an output of 0), another 1 that produces a change to *s2,* and a 0 producing a state change to *s0.* We then show the results of the sequence `010110`, which triggers the execution of states `s0, s1, s0, s1, s2,` and `s0`. The input events with bold fonts in Figure 7.4 cause the FSM to generate 1 as the output, and the output with bold fonts have the expected results.

```
00:00:00:10 in 0                    00:00:00:010 s0 0
00:00:00:20 in 1                    00:00:00:020 s1 0
00:00:00:30 in 1                    00:00:00:030 s2 1
00:00:00:40 in 1                    00:00:00:040 s2 1
00:00:00:50 in 0                    00:00:00:050 s0 0
...                                 ..
00:00:00:100 in 0                   00:00:00:100 s0 0
00:00:00:110 in 1                   00:00:00:110 s1 0
00:00:00:120 in 0                   00:00:00:120 s0 0
00:00:00:130 in 1                   00:00:00:130 s1 0
00:00:00:140 in 1                   00:00:00:140 s2 1
00:00:00:150 in 0                   00:00:00:150 s0 0
```

FIGURE 7.4 Executing the FSM in Figure 7.2 in CD++.

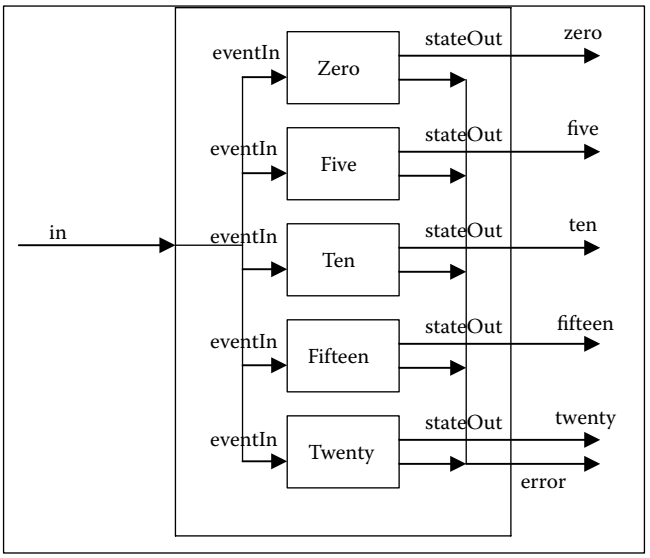


FIGURE 7.5 Vending machine model.

Exercise 7.3

Define the FSM introduced for the traffic light model in Chapter 1 and simulate it.

Figure 7.5 shows an FSM of a vending machine [4]. The machine accepts 5¢ and 10¢ coins and sells candy bars worth 15¢ or 20¢. The simplified coupled model definition (the connections among the *transitionIn*s and *transitionsOut*s are omitted) is shown in Figure 7.5.

The states *Zero, Five, Ten, Fifteen,* and *Twenty* indicate the amount of the coins that has been inserted before selling a candy bar. The *in* port receives the external events from the environment. The ports *zero, five, ten, fifteen,* and *twenty* generate the output from the respective states. The FSM of this vending machine model is defined in Figure 7.6.

The scenarios for buying one 15¢ candy and two 20¢ candies are simulated in Figure 7.7.

Exercise 7.4

Run the ATM and the Plain Ordinary Telephony Service models included in the FSM library files found in *./fsm2.zip*. Propose improvements and execute the improved FSMs for these examples.

7.3 MODELING PETRI NETS

As discussed in Chapter 1, Petri nets (PNs) are a modeling formalism originally developed by C. A. Petri [5]. They are especially well suited to model concurrent systems using a formal and well-defined graphical representation (Figure 7.8) [6].

Petri nets have the following constraints:

- A place may have zero or more inputs. For example, *P1, P4,* and *P2* have zero, one, and two inputs, respectively.
- A place may have zero or more outputs. *P4, P5,* and *P3* have zero, one, and two outputs, respectively.
- A transition may have zero or more inputs. A transition with no inputs is called a *source* (*t4*).
- A transition may have zero or more outputs. A transition with no outputs is called a *sink* (*t5*).

```
components : zero@moorestate five@moorestate ten@moorestate fifteen@moorestate twenty@moorestate
out : zero five ten fifteen twenty
in : in

Link : in eventIn@zero
Link : in eventIn@five
Link : in eventIn@ten
Link : in eventIn@fifteen
Link : in eventIn@twenty

Link : transitionOut@zero transitionIn@five
Link : transitionOut@zero transitionIn@ten
Link : transitionOut@five transitionIn@ten
Link : transitionOut@five transitionIn@fifteen
Link : transitionOut@ten transitionIn@fifteen
Link : transitionOut@ten transitionIn@twenty
Link : transitionOut@fifteen transitionIn@twenty
Link : transitionOut@fifteen transitionIn@zero
Link : transitionOut@twenty transitionIn@zero

Link : stateOut@zero zero
Link : stateOut@five five
Link : stateOut@ten ten
Link : stateOut@fifteen fifteen
Link : stateOut@twenty twenty

[zero]
StateCode : 0
StateValue : 0
NumberOfTransitions : 2
Transition01 : 0->1
Transition02 : 1->2

[five]
StateCode : 1
StateValue : 5
NumberOfTransitions : 2
Transition01 : 0->2
Transition02 : 1->3

[ten]
StateCode : 2
StateValue : 10
NumberOfTransitions : 2
Transition01 : 0->3
Transition02 : 1->4

[fifteen]
StateCode : 3
StateValue : 15
NumberOfTransitions : 3
Transition01 : 0->4
Transition02 : 1->4
Transition03 : 2->0

[twenty]
StateCode : 4
StateValue : 20
NumberOfTransitions : 1
Transition01 : 3->0
```

FIGURE 7.6 Coupled model definition of the vending machine model.

```
00:00:00:10 in 0                    00:00:00:010 five 5
00:00:00:20 in 0                    00:00:00:020 ten 10
00:00:00:30 in 0                    00:00:00:030 fifteen 15
00:00:00:40 in 2                    00:00:00:040 zero 0
00:00:00:50 in 0                    00:00:00:050 five 5
00:00:00:60 in 0                    00:00:00:060 ten 10
00:00:00:70 in 1                    00:00:00:070 twenty 20
00:00:00:80 in 3                    00:00:00:080 zero 0
00:00:00:90 in 1                    00:00:00:090 ten 10
00:00:00:110 in 1                   00:00:00:110 twenty 20
00:00:00:130 in 3                   00:00:00:130 zero 0
```

FIGURE 7.7 Executing the vending machine model in Figure 7.6 in CD++.

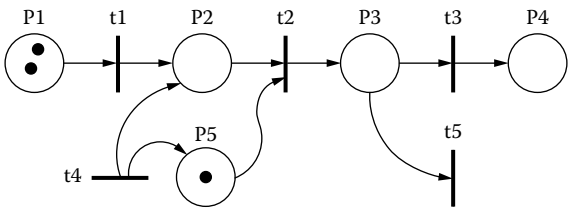


FIGURE 7.8 A sample Petri net.

Tokens (black dots in Figure 7.8) are used to model dynamic behavior: places represent the state of the system, and tokens inside the places define how many entities or resources are available. The movement of tokens from place to place represents the behavior of entities or resources in the system.

- A place may contain zero or more tokens.
- A transition is either *enabled* or *disabled*. It is enabled if all of its input places contain at least one token (in Figure 7.8, *t1* and *t4*). A *source* transition (i.e., *t4*) is always enabled; hence, it can be used as a token generator. A *sink* transition (i.e., *t5*) always consumes the tokens from the input places; thus, it can be used as a consumer of tokens.
- A PN is executed by firing enabled transitions, one at a time, for as long as there is at least one enabled transition.
- When more than one transition is enabled, the one that is fired is selected in a nondeterministic fashion.
- When a transition fires, a token is removed from each one of its input places and a token is deposited in each one of the output places.
- Transition firing is instantaneous, meaning that tokens are removed from input places and deposited in the output places at exactly the same time.

Over the years, extensions to PNs have been introduced to increase their modeling capabilities [5]. Two of these extensions include the definitions of inhibitor and multiple arcs. An inhibitor arc goes from a place to a transition, and it enables the transition only if the place is empty as opposed to containing at least one token. A multiple arc indicates that the number of tokens being transferred is more than one, as shown in Figure 7.9.

We built a library of atomic models that represent PN transitions and places. Creating the DEVS equivalent of PNs requires the PN characteristics explained previously. Thus, we created two DEVS atomic models: one to represent a place and one to represent a transition. Then any PN can be constructed by coupling the two types of DEVS atomic models in a similar manner to how places and transitions are coupled in a PN.

Figure 7.10 illustrates a conceptual description for the DEVS model of a PN place. The model uses one input port and one output port. The input *in* is used to receive tokens (from one or more

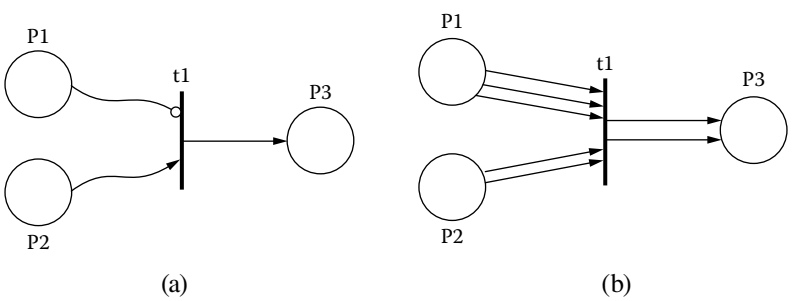


FIGURE 7.9 (a) Inhibitor arc; (b) multiple arcs.

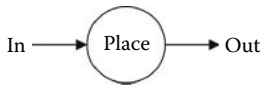


FIGURE 7.10 Place model.

transitions). It is also used to tell the place to lose tokens when a transition fires. The encoding of the messages received from this port contains information regarding the number of tokens and the operation to be performed (subtraction or addition). In this way, the model supports PN with multiple arcs. The output port *out* is used by the place to advertise the number of tokens it contains, so transitions that are connected to it can determine if they are enabled. This process is executed every time the number of tokens in the place is modified and when the model is initialized at the beginning of simulation. Figure 7.11 shows the implementation of the model in CD++, which can be found in *./Petri.zip*.

The external transition function receives external events coming from the input port *in*. The value of the messages coming into the *in* port is the number of tokens to subtract or the number of tokens received by the place. Messages carry the number of tokens to subtract and the ID of the destination place (the format is "XYYY," where X is the model ID of the place and YYY is the number of tokens; i.e., 5003 means that the place with ID 5 must subtract three tokens). When a transition wants to deposit new tokens, the destination will be 0; in those cases, we increase the number of tokens in the place.

The internal transition function passivates the model after activating the output function; then the place waits forever for a transition to deposit or remove new tokens (when all models are passive, the simulation ends). The output function transmits the number of tokens contained in the place on the output port *out*. The maximum number of tokens a place can advertise is 999. However, the proper number of tokens is kept internally.

The transition model uses five input ports and five output ports, described in Figure 7.12. Port *in1* is used to obtain the number of tokens contained in the places that have their *out* ports connected to it (this port is used for single connecting arcs; if the transition fires, only one token will be removed

```
Model &PnPlace::externalFunction( const ExternalMessage &msg ) {
  Real destId;                          // Place the message is for.

  if( msg.port() == in )   {
     destId = trunc( msg.value() / 1000 );        // Check who the message is for

     if( (ModelId) destId.value() == placeId )       {
        if( numOfTokens >= ((int) msg.value() % 1000) )  // Message specifically for this
                                                            place.
            numOfTokens -= (int) msg.value() % 1000;     // Decrement number of tokens.
        else  {
           MException e( string("Attempt to remove more tokens than no. of tokens
                                 available ");
           e.addLocation( MEXCEPTION_LOCATION() );
           throw e;            // Throw an exception
        }
      }
     else if( (ModelId) destId.value() == 0 ) // A transition wants to deposit tokens

           numOfTokens += (int) msg.value();
  }
  holdIn( active, Time::Zero );       // Immediately inform there are new tokens
}

Model &PnPlace::internalFunction( const InternalMessage & ) {
  passivate();
}

Model &PnPlace::outputFunction( const InternalMessage &msg ) {
  // If there are more than 999 tokens, show only 999.
  sendOutput( msg.time(), out, placeId * 1000 + ( numOfTokens > 999 ? 999 : numOfTokens )
);
}
```

FIGURE 7.11 Place model definition in CD++.

from the corresponding input place). Ports *in2, in3,* and *in4* serve the same function for double, triple, and quadruple connecting arcs. Finally, port *in0* is an inhibitor arc (i.e., the input place connected to it must contain zero tokens for the transition to be enabled, and when it fires, no token is removed from the place).

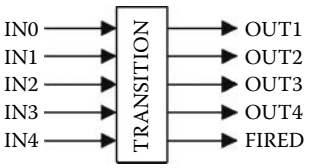


FIGURE 7.12 Transition conceptual model.

The output port *out1* is used to feed a token to all the places that have their *in* port connected to it. The *ID* of the messages sent on this port is always zero (which causes all places receiving it to update the number of tokens they hold). Ports *out2, out3,* and *out4* serve the same purpose with two, three, and four tokens. Finally, the *fired* port is used to request to remove tokens from the input places, which must have their *in* port connected to this output port.

The external transition function shown in Figure 7.13 receives external events with the format "XYYY" discussed earlier. The ID is used to keep track of all the places that feed tokens to this transition. We search this ID in the places array *pInArray* and determine if the transition is now enabled due to this message. If this place is not in the array yet, we add it and decide if the transition is now enabled.

Note that there is a case where *transEnabled* is set to *true* but should not be: the first time a message is received from *placeId* and all other input places in the array have enough tokens to make the transition enabled. If there was no match for *placeId*, it is because this is the first message received from that place. Therefore, we store *placeId* in the array along with the arc width and determine if the transition is potentially enabled from that place. In that case, we schedule an internal event to fire the transition in the future. Because this transition schedules its firing independently of the others, it may fire at the same time as others. However, because of the *select* function, the firings would not be processed simultaneously (although the log and output files would show more than one firing at the same time).

The output function is activated when the transition fires. The function deposits tokens in the places that have their *in* port connected to the out_i ports. The number of tokens deposited in the output places depends on the port to which they are connected. The messages sent out contain the number of tokens to be deposited (we use only one of the output ports when coupled to the corresponding place). The routine also indicates that it fired by sending a message to all of its inputs' places that are connected through the *fired* port. The transition keeps track of the inputs' places using their IDs.

When the internal transition function is activated, if the transition has input places, the model passivates, waiting for those places to re-advertise the number of tokens they contain. In the case of a source transition (always enabled), the next firing is scheduled. The function *randNumGet* returns a random number to schedule firings.

Figure 7.14(a) illustrates how transition/place models are coupled (two *fired* ports were used to make the figure clearer, but the model actually uses only one). The figure shows a transition that is enabled when *P1* has no token, *P2* has at least two tokens, and *P3* receives three tokens when the transition fires. Additionally, *P2* loses two tokens because the *fired* port of the transition is connected to its *in* port. Even though the *fired* port of *t1* is connected to the *in* port of *P1*, the latter does not lose tokens when *t1* fires because it is connected via an inhibitor arc.

Figure 7.14(b) shows the CD++ coupled model file that is equivalent to the coupled model illustrated in Figure 7.14(a). Places use the *pnPlace* atomic model and transitions use *pnTrans*. By default, a place is created with zero tokens (unless we declare it otherwise). The *inputplaces* parameter associated with the transitions indicates the maximum number of input places that can be connected (in order to limit the amount of memory the transition model allocates to keep information about input places), and the *tokens* parameter defines the initial number of tokens for each of the places.

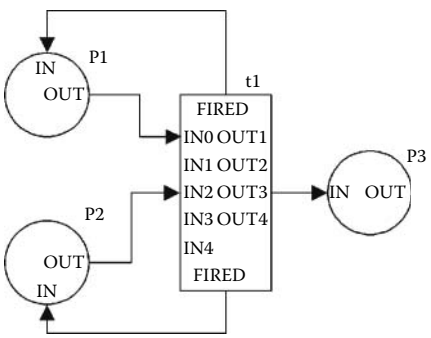
The model package also includes a Tcl/tk script to assist modelers in analyzing test results. Because we are interested in the marking of the PN and the firing of the transitions, the script returns this information in a clear and concise manner, as can be seen in Figure 7.15 (the tool parses

```
Model &PnTrans::externalFunction( const ExternalMessage &msg ) {
  Real         placeId;       // Place the message came from
  unsigned int numOfTokens;   // Number of tokens in that place
  unsigned int arcWidth;      // Width of arc connecting input place to this transition
  bool         placeIdMatch;  // Flag indicating if placeId match occured
  unsigned int i;             // array index

  placeIdMatch = false;
  transEnabled = true;

  if( msg.port() == in0 ) {arcWidth = 0;}  // Width of the connecting arc depends on the
                                              port
  else if( msg.port() == in1 ) {arcWidth = 1;}
  else if( msg.port() == in2 ) {arcWidth = 2;}
  else if( msg.port() == in3 ) {arcWidth = 3;}
  else if( msg.port() == in4 ) {arcWidth = 4;}

  // Determine which input place sent message and how many tokens are contained in that
     place.
  placeId = trunc( msg.value() / 1000 );
  numOfTokens = (int) msg.value() % 1000;

  pInArray = pArrayStart;

  for(i=0; i<numOfInputs ; i++ ) { // Check if place message comes from is in input place
     array
      if( (!placeIdMatch) && (pInArray->placeId == (int) placeId.value()) )  {
         if( (arcWidth != 0) && (numOfTokens >= arcWidth) )  // Determine if the
                                                                transition is
            pInArray->enabled = true;                        // potentially enabled
                                                                because of
         else if( (arcWidth == 0) && (numOfTokens == 0) )    // this message.
            pInArray->enabled = true;
      else
         pInArray->enabled = false
      placeIdMatch = true;
    }

   transEnabled = transEnabled && pInArray->enabled;        // Determine if transition
                                                               enabled.
   ++pInArray;
   } // End of for loop

    if( !placeIdMatch )       {
      if( numOfInputs == inPlaces ) {  // Check space left in array before adding new
                                          placeId.
          ++numOfInputs;
          MException e( string("inputplaces is too small to handle all the places);
          e.addLocation( MEXCEPTION_LOCATION() );
          throw e;
       }
      pInArray->placeId = (int) placeId.value();
      pInArray->arcWidth = arcWidth;

      if( (arcWidth != 0) && (numOfTokens >= arcWidth) )
          pInArray->enabled = true;
      else if( (arcWidth == 0) && (numOfTokens == 0) )
          pInArray->enabled = true;
        else
          {
          pInArray->enabled = false;
          transEnabled = false;
          }
     numOfInputs++;
    }  // if !placeIdMatch

  if( transEnabled )
      holdIn( active, (float) this->randNumGet() );
  else
      passivate();
}
```

FIGURE 7.13 Petri Net implementation.

```
Model &PnTrans::internalFunction( const InternalMessage & ) {
  if( numOfInputs == 0 ) // Is this is a source transition? Schedule the next firing.

      holdIn( active, (float) this->randNumGet() );
  else
      passivate();   // Wait for input places inform the number of tokens they contain.
}

Model &PnTrans::outputFunction( const InternalMessage &msg ) {
  unsigned int i;              // array index

  pInArray = pArrayStart; // Set pInArray to the start of the array of input places.


  sendOutput( msg.time(), out1, 1 );         // Deposit tokens in all output places
  sendOutput( msg.time(), out2, 2 );
  sendOutput( msg.time(), out3, 3 );
  sendOutput( msg.time(), out4, 4 );

  for( i = 0; i < numOfInputs; i++ )     {   // Remove tokens from all input places
      sendOutput( msg.time(), fired, pInArray->placeId * 1000 + pInArray->arcWidth );
      ++pInArray;
  }

  if( numOfInputs == 0 )  // Even source transitions send a "fired" message to indicate
                          firing.
      sendOutput( msg.time(), fired, 0 );
}
```

FIGURE 7.13 (continued).



(a)

```
[top]
components : P1@pnPlace P2@pnPlace P3@pnPlace
t1@pnTrans

Link : out@P1 in0@t1
Link : out@P2 in2@t1
Link : out3@t1 in@P3
Link : fired@t1 in@P1
Link : fired@t1 in@P2

[P2]
tokens : 3

[t1]
inputplaces : 2
```

(b)

FIGURE 7.14 (a) Coupling places and transition; (b) model definition file.

```
Petri Net places: p1 p2 p3
Petri Net transitions: t1

 (5,3,0)„ t1->(2,1,4)
```

FIGURE 7.15 Output of the Petri net marking tool.

the coupled model file to determine the names of the places and transitions used in the model and the log files resulting from a simulation).

The first two lines in Figure 7.15 list the names of the places and transitions of the PN. Then we show the initial marking; in this case, *p1* = 5, *p2* = 3, and *p3* = 0. Then *t1* is fired, which results in

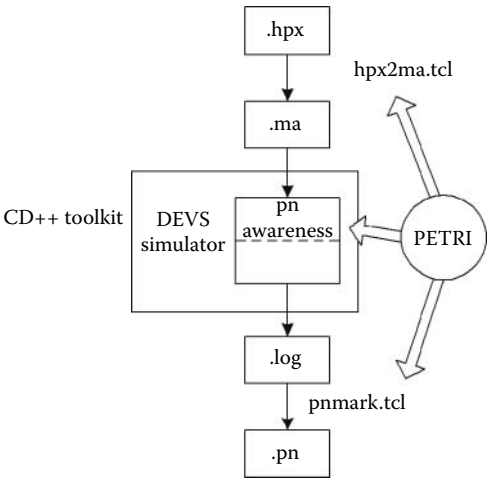


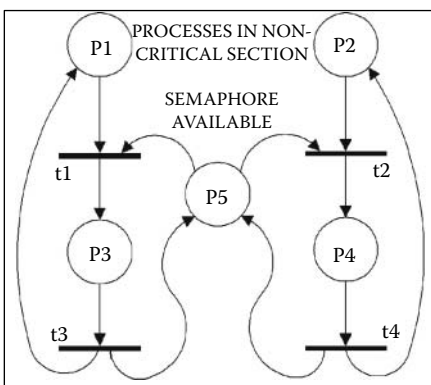
FIGURE 7.16 Architecture of the PN modeling and simulation environment.

the marking (2,1,4). In this example, *p1* and *p2* are input places to *t1* and *p3* is an output place of *t1*. Furthermore, *p1* has a triple arc to *t1* (it lost three tokens due to the firing), *p2* has a double arc, and *p3* has a quadruple arc.

We also integrated our models with the HPSIM© modeling environment for PN (which includes a graphical editor). HPSIM, which can be found at http://www.winpesim.de/, is a graphical interface to define PN models. Figure 7.16 shows the three main components in this environment: PN models in CD++, the *.hpx* files generated by HPSIM (representing the PN graphical notation), and the TCL script to generate results (stored in *.pn* files).

The first step of the cycle is to create a PN using HPSIM. Then we export the file and translate it into a CD++ model file (*.ma*) using the *hpx2ma.tcl* script (which allows an HPSIM model definition file to be converted to a model definition file for CD++). The model is then executed using CD++, and *pnmark.tcl* translates the simulation log file into a PN marking file (*.pn*).

Figure 7.17 illustrates the use of a PN to model a classical mutual exclusion problem [5]. The idea is that two processes must execute critical sections (places *P3* and *P4*) but are not allowed to do it at the same time (i.e., *P3* and *P4* must be mutually exclusive). To enforce this rule, the processes use a




```
components : P1@pnPlace P2@pnPlace P3@pnPlace
P4@pnPlace P5@pnPlace T1@pnTrans T2@pnTrans
T3@pnTrans T4@pnTrans

Link : out@P1  in1@T1
Link : out1@T1 in@P3
Link : out@P3  in1@T3
Link : out1@T3 in@P1
Link : out@P2  in1@T2
Link : out1@T2 in@P4
Link : out@P4  in1@T4
Link : out1@T4 in@P2
Link : out@P5  in1@T1
Link : out@P5  in1@T2
Link : out1@T3 in@P5
Link : out1@T4 in@P5

Link : fired@T1 in@P1
Link : fired@T1 in@P5
Link : fired@T2 in@P2
Link : fired@T2 in@P5
Link : fired@T3 in@P3
Link : fired@T4 in@P4
```

FIGURE 7.17 Mutual exclusion scenario.

```
(1,1,0,0,1)-t1->(0,1,1,0,0)-t3->(1,1,0,0,1)-t2->(1,0,0,1,0)-t4->(1,1,0,0,1)-t2->(1,0,0,1,0)
-t4->(1,1,0,0,1)-t1->(0,1,1,0,0)-t3->(1,1,0,0,1)-t1->(0,1,1,0,0)-t3->(1,1,0,0,1)-t1->(0,1,1,0,0)
...
-t4->(1,1,0,0,1)-t1->(0,1,1,0,0)-t3->(1,1,0,0,1)-t1->(0,1,1,0,0)
```

FIGURE 7.18 Mutual exclusion scenario.

semaphore (*P5*) that is locked before entering the critical section (*t1* or *t2*) and released after exiting the critical section (*t3* or *t4*).

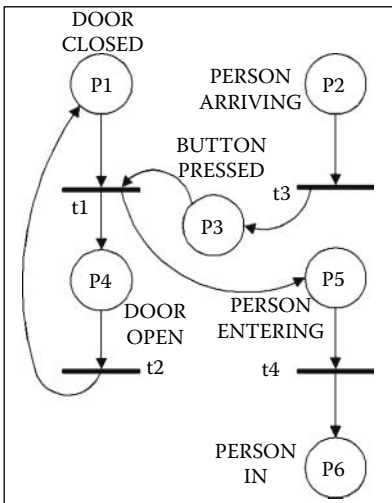
We implement the PN using atomic models *pnPlace* and *pnTrans,* as before, and configuration parameters `inputplaces` and `tokens` (Figure 7.18). We start with one token on each of the processes (in the noncritical section), and the semaphore is unlocked. Then *t1* is fired, resulting in marking (0, 1, 1, 0, 0); that is, the first process is in the critical section (and the other cannot execute it). Following this, *t3* fired to bring the Petri net back to its initial marking. This is followed by the firing of *t2,* resulting in marking (1, 0, 0, 1, 0). Then *t4* is fired, bringing the net back to its initial marking. The rest of the markings are simply a repetition of the preceding, except for the order in which the processes lock the semaphore. Sometimes a process gets the semaphore just after releasing it. This is also expected because transitions *t1* and *t2* are always enabled at the same time and the decision to fire one or the other is made in a nondeterministic manner (permitting study of cases of starvation).

Exercise 7.5

Modify the initial conditions of the model, including two tokens in the semaphore. Study the simulation results.

Figure 7.19 illustrates a simple model of an elevator's door controller. This PN (which is faulty) was created to show a model representing a real system with problems (or the problems we face when creating a wrong PN specification). We show how to use the PN to find such problems and fix them.

The PN represents the behavior of the elevator's door (*P1* = closed, *P4* = open) and of people using the elevator (*P2* = people arriving, *P3* = person pressing a button, *P5* = person entering the elevator, and *P6* = person inside the elevator). The simulation starts with the door closed and a pool




```
(1,10,0,0,0,0)-t3->(1,9,0,0,1,0)-t1->
(0,9,1,1,0,0)-t3->(0,8,1,1,1,0)-t4 ->
(0,8,1,0,1,1)-t2->(1,8,0,0,1,1)-t3->
(1,7,0,0,2,1)-t1->(0,7,1,1,1,1)-t3->
(1,6,0,1,2,1)-t4->(1,6,0,0,2,2)-t3->
(1,5,0,0,3,2)-t3->(1,4,0,0,4,2)-t1->
(0,4,1,1,3,2)-t3->(0,3,1,1,4,2)-t4->
(0,3,1,0,4,3)-t2->(1,3,0,0,4,3)-t3->
(1,2,0,0,5,3)-t1->(0,2,1,1,4,3)-t3->
(0,1,1,1,5,3)-t4->(0,1,1,0,5,4)-t3->
(0,0,1,0,6,4)-t2->(1,0,0,0,6,4)-t1->
(0,0,1,1,5,4)-t2->(1,0,0,1,5,4)-t4->
(1,0,0,0,5,5)-t1->(0,0,1,1,4,5)-t2->
(1,0,0,1,4,5)-t1->(0,0,1,2,3,5)-t4->
(0,0,1,1,3,6)-t2->(1,0,0,1,3,6)-t1->
(0,0,1,2,2,6)-t2->(1,0,0,2,2,6)-t4->
(1,0,0,1,2,7)-t4->(1,0,0,0,2,8)-t1->
(0,0,1,1,1,8)-t4->(0,0,1,0,1,9)-t2->
(1,0,0,0,1,9)-t1->(0,0,1,1,0,9)-t4->
(0,0,1,0,0,10)-t2->(1,0,0,0,0,10)
```

FIGURE 7.19 Elevator system.

of 10 people arriving and waiting to use the elevator (1, 10, 0, 0, 0, 0). When we fire *t3*, the button is pressed, and then the door is opened (*t3*). A new person presses the button (*t3*); we then fire *t4,* and there is one person in the elevator.

If we study the set of markings generated, we can find the following problems:

- The door can close (*t2* fires) even if the button is pressed (*P3* not empty). Typically, the door of an elevator stays open as long as the button is pressed.
- The door can be closed (*P1* = 1) while someone is entering the elevator (*P5* not empty).
- The button pressed state records how many times the button was pressed (*P3* can be >1). Although it is physically possible for the button to be pressed many times, in reality an elevator button does not count how many times it is pressed. That is, if it is pressed and then someone presses it again, it stays pressed—nothing more. Thus, its state is really Boolean.
- Every time someone presses the button, a person disappears.

Exercise 7.6

Add a sink transition to eliminate the persons leaving the elevator and a source transition to bring new people to the area.

Exercise 7.7

Use CD++Modeler to define graphical icons for transitions and places. Use these icons to create a coupled model with PN structure for all the models in this section. Use the animation facilities to visualize the simulation results.

Exercise 7.8

Fix the aforementioned problems in the elevator PN. Run the simulation again until the model reproduces the normal behavior of an elevator's door. Use the graphical interface created in Exercise 7.7 to visualize the results.

Exercise 7.9

Create a PN that models an elevator moving up and down in a three-floor building. The PN should only model the behavior of the elevator's motor (ignore opening and closing doors), and the elevator should stop on each of the floors according to the users' requests.

Exercise 7.10

Combine the elevator model created in Exercise 7.9 with the button controller created in Exercise 7.8. Study the PN simulation results and modify your models until obtaining the proper behavior for the whole elevator system.

Exercise 7.11

Define and simulate the producer/consumer PN introduced in Chapter 1.

7.4 LAYERED QUEUING NETWORKS

As discussed in Chapter 2, queuing networks are based on a customer–server paradigm: customers request service to servers, which queue the requests until they can be serviced. Traditional queuing networks model only a single layer of customer–server relationships. Layered queuing networks (LQNs), however, allow for an arbitrary number of client–server levels. LQNs can model intermediate software servers and can be used to detect software deadlocks as well as hardware and software performance bottlenecks [7]. The layered aspect of LQNs makes them suitable for evaluating the performance of distributed systems [8,9].

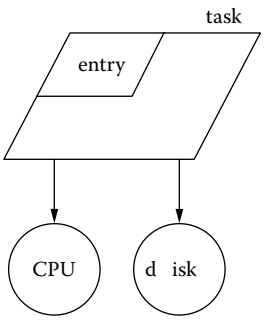


FIGURE 7.20 LQN task, entry, CPU, and disk devices.

LQNs model both software and hardware resources. The basic software resource is a *task,* which represents any software object with a thread of execution. Tasks have *entries* that act as interface units. The basic hardware resource is a *device.* Typical devices are CPUs and disks.

Figure 7.20 shows the graphical notation for LQN tasks. Tasks receive service requests at designated entries (corresponding to the service access points for a task; we use a different entry for every kind of service a task provides). An entry may be defined to be atomic (including its own hardware service demands and calls to other tasks), or it may be defined by blocks of smaller computational blocks called *activities.* Service calls can be made from entries in one task to entries in other tasks. In that case, entries can be atomic or they can be subdivided into phases (dividing the workload into a first phase executed before sending a reply and a second phase executed after sending the reply).

As shown in Figure 7.21, the LQN notation supports three types of service calls: *asynchronous, synchronous,* and *forwarded* calls (the graphical notation for service calls uses arrows for messaging activities). Figure 7.22 shows the timing semantics for these different types of calls.

Asynchronous calls do not involve any blocking of the sending task, which continues executing concurrently. A synchronous call must send a reply after the request has been completed. The replies are implicit at the end of the first phase for atomic entries but must be explicitly specified for entries defined by activities. Entries receiving a synchronous service request may also forward it to entries in other tasks, which then become responsible for sending the reply to the original caller. In a forwarding call, the sending client task makes a synchronous call and blocks waiting for a reply; the intermediate receiving server task then partially processes the call and forwards it to another

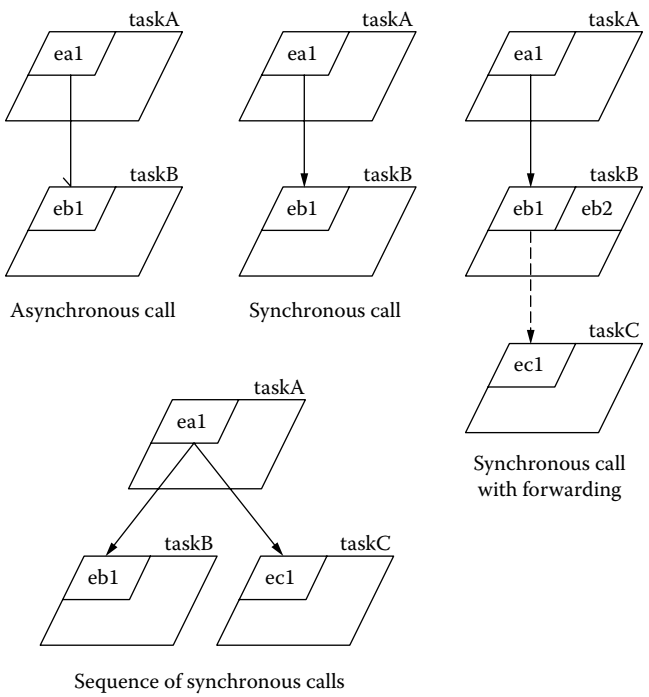


FIGURE 7.21 LQN messaging.

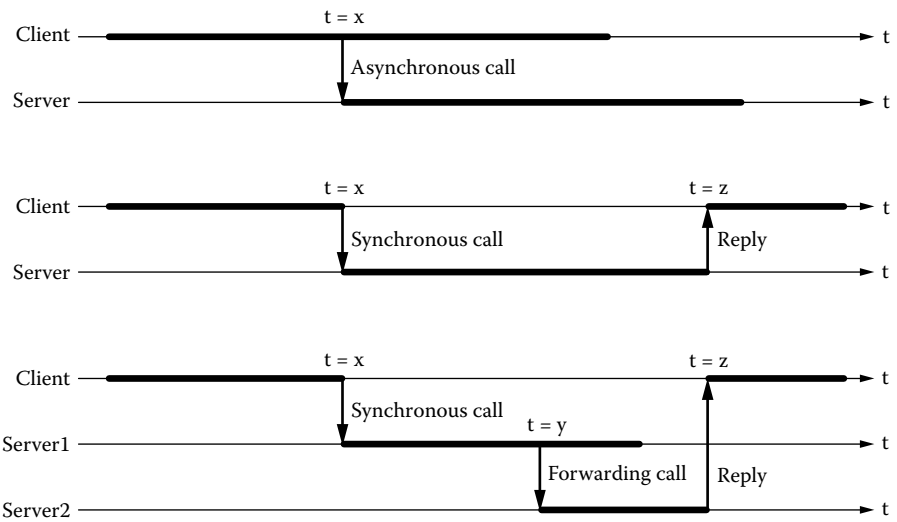


FIGURE 7.22 Timing semantics of LQN calls. (From Petriu, D., and G. Wainer. 2004. *Proceedings of Mediterranean Multiconference on Modeling and Simulation,* Bergeggi, Italy.)

server, which becomes responsible for sending a reply to the blocked client. The intermediate server can continue operating after forwarding the call (there can be any number of forwarding levels).

A DEVS library for LQNs was introduced in Petriu and Wainer [10], and the complete set of models can be found in *./DevsLQN_code.zip.* The library includes models for processors, tasks, and entries with phases. Additionally, the library can represent disks and activities. The library provides simulation results for:

- average entry service time, throughput, and utilization;
- average phase service time;
- processor throughput and utilization; and
- average queue waiting time and average queue length.

LQN elements use queues; thus, first in, first out (FIFO) queues were explicitly incorporated. Because queues behave the same way for software or hardware elements, we implemented a universal queue as a separate atomic model to be coupled with the processor or entry atomic models. LQN calls are made using entry names to identify the call target. Therefore, we implemented a DEVS version of a multiplexer and demultiplexer to gather calls into a given queue (either for an entry or a processor) or to distribute calls from an entry to other entries. DEVS atomic models were built for each of these elements. Figure 7.23 describes the behavior of the *queue* atomic model.

The initial state (*ready_to_process*) represents a queue ready to receive a request. When an external transition is executed and an external event arrives through the *in* port, the element is added to the end of the queue. If all the fields have been received, we calculate the average queue size. We use different queues: one to record the arrival time of the input, another to record the source of the message, and a third to save the input value. We also record the current time (which will be used in the next input to compute the average queue size). At this point, if the processing entity is ready, we change the state to inform that a call is ready to be processed, and we activate the internal transition function immediately. If a *reply* message is received, we record the fact that a reply is pending and activate the internal transition function.

```
LqnQueue::LqnQueue( const string &name ) : Atomic( name ), in( addInputPort( "in" ) ),
  response( addInputPort( "response" ) ), ready( addInputPort( "ready" ) ),
  out( addOutputPort( "out" ) ), reply( addOutputPort( "reply" ) ),
  averagesize( addOutputPort( "averagesize" ) ), averagewait( addOutputPort(
  "averagewait" ) ){}


Model &LqnQueue::initFunction() {
  for( fnum = 0; fnum < LQNFIELDS; fnum++ )  lqnmsg[ fnum ] = 0; // initialize the LQN
                                                                    message
  fnum = 0;

  time_queue.erase( time_queue.begin(), time_queue.end() ) ;    // make sure the queues
                                                                   are empty
  val_queue.erase( val_queue.begin(), val_queue.end() ) ;
  src_queue.erase( src_queue.begin(), src_queue.end() ) ;

  elements_through = sum_wait = av_size = 0;     // initialize the queue counters
  last_qxtime = "0:0:0:0";

  rep_dest = 0;
  ready_to_process = 1;  // assume that to begin with the entity is ready
  rep_pend = call_pend = 0;  // no reply or call is pending yet
}


Model &LqnQueue::externalFunction( const ExternalMessage &msg ) {
  if( msg.port() == in ) {   // new call arrives at the queue
    if (fnum >= LQNFIELDS) fnum = 0; // if fnum >= number of LQN fields, reset fnum

    lqnmsg[fnum++] = msg.value();    // assemble the LQN fields in order

    if( fnum == LQNFIELDS ) {        // if all the fields have been received
      // calculate the average queue size up to now
      av_size = ( ( av_size * last_qxtime.asMsecs() ) +
                val_queue.size() * ( msg.time() - last_qxtime ).asMsecs() ) /
                                   msg.time().asMsecs();

      // add to the back of the queues
      time_queue.push_back( msg.time() );
      src_queue.push_back( lqnmsg[ LQNSRC ] );
      val_queue.push_back( lqnmsg[ LQNVAL ] );

      // update the last queue change time
      last_qxtime = msg.time();

      if( ready_to_process ) {              // if the processing entity is ready
        ready_to_process = 0;               // no longer ready to process
        call_pend = 1;                      // a call is ready to be processed
        holdIn( active, Time::Zero );       // instantaneous internal transition
      }
    }
  }
  else if( msg.port() == response ) {    // reply message
    rep_pend = 1;                        // a reply is pending
    holdIn( active, Time::Zero );        // reply right away
  }
  else if( msg.port() == ready ) {       // ready message
    if( val_queue.size() ) {             // if there are elements waiting in the queue
      ready_to_process = 0;                // not ready to process
      call_pend = 1;                       // a call is ready to be processed
      holdIn( active, Time::Zero );        // send right away
    }
    else
       ready_to_process = 1;             // else the queue is empty: now ready to process
  }
}

Model &LqnQueue::internalFunction( const InternalMessage & ) {
```

FIGURE 7.23 DEVS queue atomic model.

```
  passivate();
}

Model &LqnQueue::outputFunction( const InternalMessage &msg )  {
  if( rep_pend ) {
    sendOutput( msg.time(), reply, rep_dest );       // send the reply
    rep_pend = 0;                                    // a reply is no longer pending
  }

  if( call_pend ) {     // if a call is ready to be processed

    elements_through++;    // increment the number of elements that have gone through the
    queue

    // add the element's waiting time in ms to the cumulative waiting time
    sum_wait += ( msg.time() - time_queue.front() ).asMsecs();

    // calculate the average queue size up to now
    av_size = ( ( av_size * last_qxtime.asMsecs() ) +
              val_queue.size() * ( msg.time() - last_qxtime ).asMsecs() ) /
              msg.time().asMsecs();

    sendOutput( msg.time(), averagesize, av_size );       // send the performance metrics
    out
    sendOutput( msg.time(), averagewait, sum_wait / elements_through );
    sendOutput( msg.time(), out, val_queue.front() );    // send the call value to be
    processed

    rep_dest = src_queue.front();     // use the call's source as the reply destination


    time_queue.pop_front();              // de-queue the first elements
    src_queue.pop_front();
    val_queue.pop_front();
    last_qxtime = msg.time();            // update the last queue change time
    call_pend = 0;                       // a call is no longer ready to be processed
  }
}
```

FIGURE 7.23 (continued).

Finally, when a *ready* event arrives, we check the queue to see if there are more elements. If not, we change state to *ready_to_process,* waiting for a new element arriving through the *in* port. Otherwise, we change the state to record the fact that there is a call pending and activate an internal transition function.

The output function reacts in different ways according to the last event received. If a reply is pending, we send it and inform that the reply is no longer pending. If there is a call to be processed, we increase the number of elements that have been in the queue, sum the waiting time, and compute the average queue size. These values are transmitted through the corresponding output ports, using the call's source as the reply destination. We also eliminate the elements from the queue and update the change time. Then we go to the state *wait for response,* in which we wait for a response event (which generates a response in the reply output port) or a new element, which is added at the end of the queue.

The internal function passivates the model, waiting for the next input message.

Exercise 7.12

Construct a testing frame for the LQN queue model just presented. Analyze the simulation results.

Figure 7.24 shows the structure of the DEVS coupled models used to model LQN entries and processors. Both of them are coupled models that incorporate LQN queues and message routing multiplexers/demultiplexers. The *in* ports of the *processor* and *entry* atomic models are connected

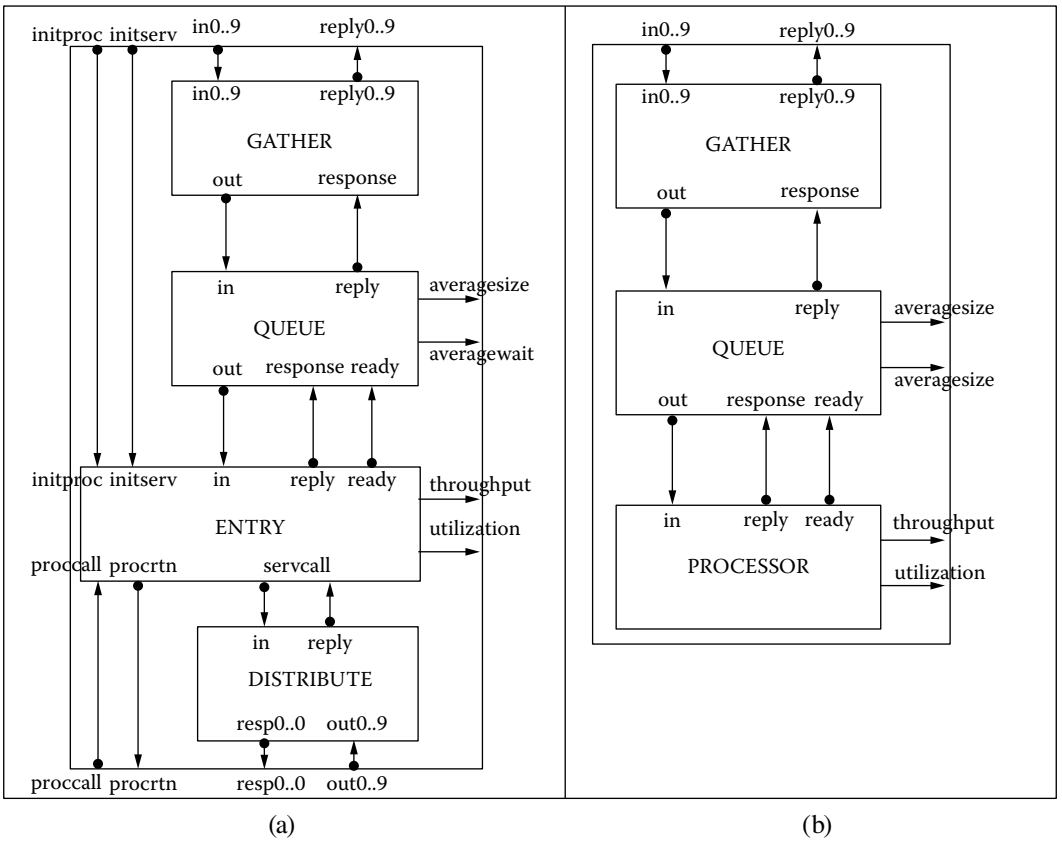


FIGURE 7.24 (a) Entry coupled model structure; (b) DEVS processor coupled model structure.

to the *out* ports of their dedicated *queue* atomic models. The *in* port of the *queue* is connected to the output port of the *gather* multiplexer model.

The *gather* model collects the different inputs and routes them to the LQN queue, which will store the messages, respond with the average size and wait times, and reply when the message has been processed. When a complete entry is ready, it is transmitted to the *entry* model, which will also compute throughput and utilization. The *distribute* model acts as a demultiplexer, returning the corresponding responses through the output ports associated with the entry. For entries, the *servcall* output port is connected to the *in* port of the *distribute* demultiplexer, which sends it on the appropriate *resp* port for the intended call target. The same sorts of connections are repeated for the *reply* ports but with the reply messages going in the opposite direction. The *processor* model is simpler because it does not need to forward the input messages: it receives a call and executes, and the entry can forward the input call.

Table 7.1 lists the different DEVS models for LQN elements included in the library.

LQN messages can be thought of as having a *source* field denoting the entity making the call, a *destination* or *target* field denoting the entry for which the call is destined, and a *demand* field denoting the workload associated with the call (Table 7.2).

Figure 7.25 describes a model in which each component is defined as an LQN element representing a client–server system. In this case, the client *ref* calls *entry1* in server *e1* and *entry2* in server *e2*. The *entry1* has a mean processor demand of 1,100 ms, and *entry2* has a mean processor demand of 2,100 ms. All three tasks run on the same processor, *P1*.

TABLE 7.1
DEVS Models for the LQN Simulation Library

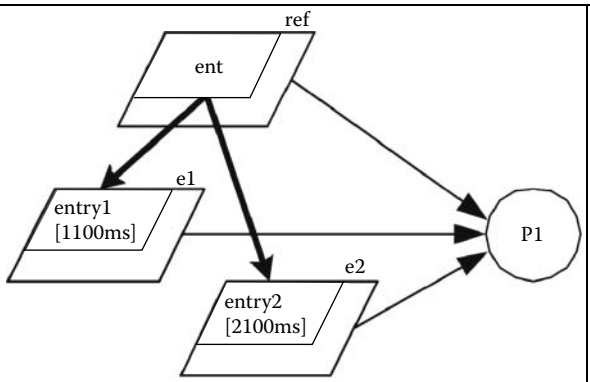
LQN Aspect–Element	Atomic Model	Coupled Model	Functionality
Processor	Processor		Receives call, executes it for the specified time
			Replies when done
			Calculates utilization and throughput
		Processor	Combines gather, queue, and atomic processor for LQN processor functionality
Entry with phases	Entry		Receives call, executes associated workloads (phase 1/2 processing), makes calls, and replies when done
			Processor demands for phase 1/2 must first be initialized through *initproc* port
			Server calls for phase 1/2 must first be initialized through the *initserv* port
		Entry	Combines gather, queue, atomic entry, and distribute (LQN entry functionality)
Implied queue	Queue		Adds call to queue
			Sends first element in queue to attached idle processor or entry
			Passes reply backup to the call source
Aggregating calls (multiple sources)	Gather		Aggregates calls from multiple input ports and sends them out to a single output port
			Adds a message with the input port index
			Passes reply from port *output end* through to appropriate response port *input end*
Distributing calls (different entries)	Distribute		Receives calls on single input port and distributes them to the appropriate output port
			Sends reply from the reply port at the *output end* to the single response port at the *input end*
Task		Task	Coupled model composed of multiple entries
Disk		Processor	Reuses the functionality of a processor
Activity	N/A	N/A	Further subdivides the workload of an entry (currently not implemented)

We will analyze the execution results of this model of the input events presented in Figure 7.25. We first set up the client *ref* by making phase 1 of entry *ent* in task *ref* to be initialized to make one call to entry *entry1* in task *e1* and one call to entry *entry2* in task *e2* (a call initialization is assembled from three messages: one for the phase making the call, one for the number of calls, and one for the call target). Then we see that phase 1 of entry *entry1* in task *e1* is initialized with a mean workload of 1,100 ms and phase 1 of entry *entry2* in task *e2* is initialized with a mean workload of 2,200 ms (a processor demand initialization is assembled from two messages: one for the phase and one for processing workload). Finally, 10 calls are made to entry *ent* in task *ref* at 1-s intervals. Figure 7.27 shows the execution results of this model.

Initially, we can see entries being initialized with their call and workload parameters. At 01:000, we see the execution of the first call made to entry *ent* in task *ref* and subsequent calls to entries *entry1* in task *e1,* which generates an actual processor workload of 1,087 ms, and to *entry2* in task *e2,* which generates an actual processor workload of 2,081 ms. Then we see the execution of the second call made to entry *ent* in task *ref* and the subsequent calls to entries *entry1* in task *e1* (which generates an actual processor workload of 880 ms) and to *entry2* in task *e2* (which generates a workload of 278 ms). Finally, we see the execution of the ninth call made to entry *ent* in task *ref* and subsequent calls to *entry1* in *e1,* which generates an actual processor workload of 2,256 ms, and to *entry2* in *e2,* which generates an actual processor workload of 4,947 ms. The last call made to *ent* in

TABLE 7.2
DEVS LQN Simulation Library Messages

Sender (Port)	Receiver (Port)	LQN Equiv. Msg.	DEVS Messages	Interpretation
Processor (reply)	Queue (response)	Done	Reply	Notify source entry that processing is done; message value represents actual processing time (milliseconds)
Processor (ready)	Queue (ready)	Done	Ready	Ready for another job; the message value is irrelevant
Processor (throughput)		Throughput	Throughput	Message value represents the processor throughput in number of jobs per millisecond
Processor (utilization)		Utilization	Utilization	Message value represents the fraction/percentage of time that the processor has been busy
Entry (proccall)	Distribute (in[0..9])	Processor call	Processor svc demand	Message value represents processor demand in milliseconds
Entry (servcall)	Gather (in)	Service call	Service call	Message value represents index of the target server
Entry (avservtime)		Avg entry svc time	Avg entry svc time	Message value represents avg entry svc time in milliseconds
Entry (avph1time)		Avg phase1 time	Avg phase 1 svc time	Message value represents avg phase1 svc time in milliseconds
Entry (avph2time)		Avg phase2 time	Avg phase 2 svc time	Message value represents avg phase2 svc time in milliseconds
Entry (throughput)		Throughput	Throughput	Message value represents entry throughput in jobs/millisecond
Entry (utilization)		Utilization	Utilization	Message value: percentage of busy time for entry
Queue (out)	Processor (in)	Processor call	Proc. service demand	Message value represents the service demand in milliseconds
Queue (out)	Entry (in)	Service call	Service call	Service call, the message value is irrelevant
Queue (reply)	Gather (resp.)	Reply	Reply	Message value: index of source to be replied to
Queue (avgsize)			Avg queue size	Message value: avg number of elements in the queue at the time the message was sent
Queue (avgwait)			Avg queuing wait	Message value: avg number of milliseconds a message has spent in the queue at the time the message was sent
Gather (out)	Queue (in)	Service call	Source of service call demand	Message value: index of the call source; if attached to a processor, represents processor svc demand in milliseconds
Gather (reply[ ])	Distrib.(resp[ ])	Reply	Reply	Reply; the message value is irrelevant
Distribute (out[0..9])	Gather (in[0...9])	Service call	Service call	If attached to a processor, message value represents processor service demand in milliseconds
Distribute (reply)	Entry (response)	Reply	Reply	Message value: index of call target returning the reply
	Entry (initproc)		Phase no.	Message value: phase number to initialize
			Processor demand	Message value: processor demand in milliseconds
	Entry (initserv)		Phase no.	Message value: phase number to initialize
			Calls	Number of calls to make to target server
			Call target	Index of the target server




```
components : ref e1 e2 p1
in : refinitp refinits refin e1initp
e1inits
in : e2initp e2inits
link : refinitp initp@ref
link : refinits inits@ref
link : refin in0@ref
link : pcall@ref in0@p1
link : rpl0@p1 prtn@ref
link : e1initp initp@e1
link : e1inits inits@e1
link : out0@ref in0@e1
link : rpl0@e1 rsp0@ref
link : pcall@e1 in1@p1
link : rpl1@p1 prtn@e1
link : e2initp initp@e2
link : e2inits inits@e2
link : out1@ref in0@e2
link : rpl0@e2 rsp1@ref
link : pcall@e2 in2@p1
link : rpl2@p1 prtn@e2
```

FIGURE 7.25 A simple queuing model using LQNs.

```
00:00:00:000 refinits 1
00:00:00:001 refinits 1
00:00:00:002 refinits 0
00:00:00:003 refinits 1
00:00:00:004 refinits 1
00:00:00:005 refinits 1
00:00:00:006 e1initp 1
00:00:00:007 e1initp 1100
00:00:00:010 e2initp 1
00:00:00:011 e2initp 2100
00:00:01:000 refin 1
00:00:02:000 refin 1
00:00:03:000 refin 1
00:00:04:000 refin 1
00:00:05:000 refin 1
00:00:06:000 refin 1
00:00:07:000 refin 1
00:00:08:000 refin 1
00:00:09:000 refin 1
00:00:10:000 refin 1
```

FIGURE 7.26 Input events for the model.

ref generates an actual processor workload of 1,727 ms and to *entry2* in task *e2* generates an actual processor workload of 874 ms.

7.5 VHDL-AMS

VHDL is a hardware description language that has become very popular in the field of design of digital circuits and was standardized by the IEEE. The standard VHDL-AMS (IEEE Standard 1076.1) included extensions to model mixed-signal circuits [11]. The basic component is the design *entity* declaration, which describes the interface to a VHDL-AMS design unit:

```
entity entity_name is { port ( [signal | terminal | quantity]
identifier{, identifier}: [mode | signal_type | electrical]; }+
end [entity] [entity_name] ;
```

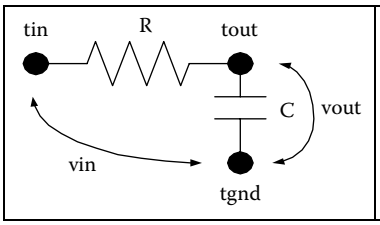
```
[00:00:00:002] entry: init phase1 call stmt 1 = 1 calls to server 0
[00:00:00:005] entry: init phase1 call stmt 2 = 1 calls to server 1
[00:00:00:007] entry: init phase1 proc demand mean = 1100 ms
[00:00:00:011] entry: init phase1 proc demand mean = 2100 ms
[00:00:01:000] entry: start; entry: phase1 server call <server 0>
[00:00:01:000] entry: start; entry: phase1 proc call <mean 1100 ms, actual 1086.79 ms>; processor:
<demand 1086.79 ms, rounded to 1087 ms>

[00:00:02:087] entry: reply; entry: done <phase1 1087 ms, phase2 0 ms>; entry: phase1 server call
<server 1>; entry: start
[00:00:02:087] entry: phase1 proc call <mean 2100 ms, actual 2080.5 ms>
[00:00:02:087] processor: <demand 2080.5 ms, rounded to 2081 ms>
[00:00:04:168] entry: reply; entry: done <phase1 2081 ms, phase2 0 ms>
[00:00:04:168] entry: reply; entry: done <phase1 3168 ms, phase2 0 ms>

[00:00:04:168] entry: start; entry: phase1 server call <server 0>
[00:00:04:168] entry: start; entry: phase1 proc call <mean 1100 ms, actual 879.851 ms>; processor:
<demand 879.851 ms, rounded to 880 ms>

[00:00:05:048] entry: reply; entry: done <phase1 880 ms, phase2 0 ms>
[00:00:05:048] entry: phase1 server call <server 1>; entry: start

[00:00:05:048] entry: phase1 proc call <mean 2100 ms, actual 278.087 ms>
[00:00:05:048] processor: <demand 278.087 ms, rounded to 278 ms>
[00:00:05:326] entry: reply; entry: done <phase1 278 ms, phase2 0 ms>
[00:00:05:326] entry: reply; entry: done <phase1 1158 ms, phase2 0 ms>
[00:00:05:326] entry: start; entry: phase1 server call <server 0>
[00:00:05:326] entry: start; entry: phase1 proc call <mean 1100 ms, actual 705.584 ms>; processor:
<demand 705.584 ms, rounded to 706 ms>
...
[00:00:30:284] entry: reply; entry: done <phase1 2375 ms, phase2 0 ms>
[00:00:30:284] entry: reply; entry: done <phase1 2920 ms, phase2 0 ms>
[00:00:30:284] entry: start; entry: phase1 server call <server 0>; entry: start; entry: phase1 proc
call <mean 1100 ms, actual 2256.45 ms>; processor: <demand 2256.45 ms, rounded to 2256 ms>
[00:00:32:540] entry: reply; entry: done <phase1 2256 ms, phase2 0 ms>; entry: phase1 server call
<server 1>; entry: start

[00:00:32:540] entry: phase1 proc call <mean 2100 ms, actual 4946.97 ms>
[00:00:32:540] processor: <demand 4946.97 ms, rounded to 4947 ms>
[00:00:37:487] entry: reply; entry: done <phase1 4947 ms, phase2 0 ms>
[00:00:37:487] entry: reply; entry: done <phase1 7203 ms, phase2 0 ms>

[00:00:37:487] entry: start; entry: phase1 server call <server 0>
[00:00:37:487] entry: start; entry: phase1 proc call <mean 1100 ms, actual 1726.73 ms>; processor:
<demand 1726.73 ms, rounded to 1727 ms>
[00:00:39:214] entry: reply; entry: done <phase1 1727 ms, phase2 0 ms>
[00:00:39:214] entry: phase1 server call <server 1>; entry: start

[00:00:39:214] entry: phase1 proc call <mean 2100 ms, actual 874.252 ms>
[00:00:39:214] processor: <demand 874.252 ms, rounded to 874 ms>
[00:00:40:088] entry: reply; entry: done <phase1 874 ms, phase2 0 ms>
[00:00:40:088] entry: reply; entry: done <phase1 2601 ms, phase2 0 ms>
```

FIGURE 7.27 Output events generated during model execution and their interpretation.

The entity declaration contains a list of ports, each of which is assigned a type and an optional mode. Ports of type *std_logic* or *std_logic_vector* (a standardized type for digital logic) are used for digital signals, and ports of type *electrical* are used for analog signals. In the case of digital signals, ports will have mode *in, out, inout,* or *buffer.* Analog ports do not require a mode.

Figure 7.28 shows the entity declaration of the input/output ports of a digital flip-flop and an analog circuit (low-pass filter). In the flip-flop declaration, *d* and *clk* are input ports of type *std_logic,* and *q* is an output port of type *std_logic.* In addition to the basic *std_logic* type, vectors of *std_logic* signals may be declared using the *std_logic_vector* type. This allows digital signals to be operated on by referencing only one signal name. In the declaration for the analog low-pass filter, *tout, tin,* and *tgnd* are *electrical* ports.

A design **architecture** describes the functionality of a design unit (it may be a structural, dataflow, or behavioral description). A single architecture is associated with exactly one entity, whose syntax is



(a)

```
entity d_flip_flop is
   port( d, clk : in std_logic; q : out std_logic; );
end entity d_flip_flop ;

entity LPF is
  port (terminal tout, tin, tgnd: electrical);
end entity LFP;
```

(b)

FIGURE 7.28 (a) Low-pass filter. (b) VHDL flip-flop and low-pass filter definitions.

```
architecture architecture_name of entity_n is signal_declaration
     | constant_declaration | component_declaration
begin
     {process_statement | concurrent_signal_assignment_statement
     | component_instantiation_statement | simultaneous_statement}
end [architecture] [architecture_name] ;
```

The body of an architecture is made up of statements that may be categorized as **concurrent**, **sequential**, or **simultaneous**. These statements operate on signals/quantities declared within the scope of the architecture and ports that are declared in the entity with which the architecture is associated.

Signals and **quantities** are declared in the declarative region of an architecture. They belong to the scope of the architecture in which they are declared and may be referenced only within that architecture. Signals and quantities have types (similar to ports in the entities). Types *std_logic* and *std_logic_vector* are used for digital logic. Signals and quantities are defined as follows:

```
signal signal_name : std_logic_vector
(upper_bound downto lower_bound) | std_logic ;
quantity identifier: REAL | Voltage | Current | Charge ;
```

Quantities can also be declared as relative to terminals in an entity, defined as *across* or *through* quantities. *Across* quantities represent the voltage at the free terminal relative to the reference terminal. *Through* quantities represent the current from the free terminal into the reference terminal:

```
quantity identifier {, identifier} across identifier
{, identifier} through free_terminal to reference_terminal ;
```

Concurrent statements within an architecture body execute concurrently. They include statements for *process, simultaneous, concurrent assignment,* and *conditional concurrent assignments.* The conditional concurrent assignment assigns a target signal using a condition. The unconditional concurrent assignment always assigns the value of the source signal to the target signal:

```
target_signal <= expression1 when condition
  else expression2; // conditional
target_signal <= source_signal; //unconditional
```

A **process** executes the statements between *begin* and *end process* when an event occurs on a signal in its sensitivity list. All signals modified by the process are updated only when the process body is completed. The statements between *begin* and *end* (sequential statements) are executed in sequence:

```
[process_name:]
process (sensitivity_list) { type_declaration }
begin
  {signal_assignment_statement | if_statement | case_statement
end process [process_name] ;
```

The **if-then-else** statement has the same semantic found in most programming languages.

```
[ if_name: ] if condition then sequence_of_statements
             {elsif condition2 then sequence_of_statements }
             [else sequence_of_statements ]
end if [ if_name ] ;
```

The **case-when** statement runs the sequence of statements listed under the *when* clause whose expression matches that of the expression in the *case* statement:

```
[ case_name: ] case expression is
 {when identifier | expression | discrete_range | others =>
 sequence_of_statements}+
end case [ case_name ] ;
```

The **sequential assignment** assigns the value of the driver signal to the target signal. When executed from within a process, the target will not get the value of the driver until the end of the process:

```
[ label: ] target <= driver ;
```

Simultaneous statements are used for describing differential algebraic equations and may consist of quantities or signals, including a minimum of one quantity per simultaneous statement. Simultaneous statements may appear anywhere a concurrent statement may appear, and they have no order.

Components facilitate hierarchical design. A component instance is a copy of the named entity and its associated architecture that interacts with the architecture it is instantiated within. The *port* map clause specifies which ports of the entity are connected to which signals in the enclosing architecture body:

```
Instantiation_label : entity entity_name
port map ( {port_name => signal_name | expression | variable_name
| open }+ );
```

In [12] we defined a library based on VHDL-AMS that is targeted toward register transfer level modeling of digital circuits (with limited behavioral modeling and analog constructs). The library (found in *./VHDL.zip*) is called sAMS-VHDL and integrates many of the features of VHDL-AMS [11]. Each of the sAMS-VHDL constructions was converted into a DEVS model and made

it available for execution in CD++. **Process** models are translated into CD++ by converting its sequential statements to C++ code and instantiating ports for every signal that is read or driven from within the process and for every signal in the process's sensitivity list. Figure 7.29 illustrates the structure of a DEVS model generated from a flip-flop process.

The process body is implemented within the external transition function. The values received from external events generated on the input ports (representing read and sensitivity list signals) are buffered within the model. If the process body contains a reference to *rising_edge(signal_name)* or *falling_edge(signal_name)* operations, the values received from the external events are stored on a buffer of length two within the model (keeping the previous and current values of the signal).

The sequential statements in the process body are converted to C++ and inserted into the external transition, as is the case with *if, case,* and *assignment* statements.

The Boolean expression that refers to read and sensitivity list signals in a VHDL *if* statement is replaced with an equivalent Boolean expression that refers to port buffers for those signals. If the condition within an *if* statement contains a sensitivity list signal, then we instruct the process model to change to the *active* state in 0-time (causing an instantaneous output and internal transition). The output event will update all driven signals (by sending the value of each output port buffer), and the internal transition will cause the model to return to the *active* state.

Figure 7.30 shows parts of the sAMS VHDL code for a process used in a 4-bit counter and its translation into CD++. This process has one sensitivity list signal (*clk*), four read signals (*d1*…*d4*), and four driven signals (*q1*…*q4*). The process body contains an *if* sequential statement with a Boolean expression that contains the *rising_edge* operation acting on signal *clk,* and four sequential assignment operations.

We show a fragment of the C++ code generated in which *o_clk, n_clk, _d1, _d2, _d3,* and *_d4* are input port buffers; *_q1, _q2, _q3,* and *_q4* are output port buffers; and *_1164and, _1164not,* and *_1164xor* are functions that implement *and, not,* and *xor* operators in C++.

Signals are used to determine how to interconnect the ports on the many process model instances for each component. This information is then used during model file generation to create links between the models. DEVS links provide instantaneous communication between the components, so a signal model is created to implement transport delays on messages sent between process model ports. The signal model receives and buffers data on its input port, enters the active state for the time specified by the assignment statement transport delay, and then outputs the buffered data on its output port, as shown in Figure 7.31.

Simultaneous statements allow the definition of ordinary differential equation systems with initial conditions. In this library, the problem of simulating an *n*th-order differential equation is solved by reducing the equation into a set of first-order differential equations. For example,

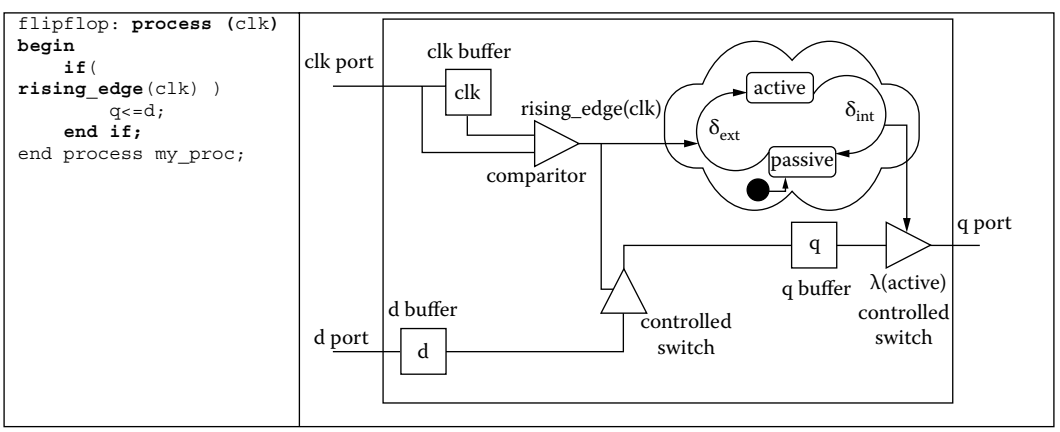


FIGURE 7.29 VHDL process model for *flip flop* DEVS definition.

```
Counter: process (clk) is
begin
     if(rising_edge(clk))
  q1<=not d1;
  q2<=d1 xor d2;
  q3<= d3 xor (d1 and d2);
  q4<=d4 xor (d1 and d2 and d3);

     end if;
end Counter;
```

```
if (msg.port()==clk) {  // clk is in the trigger list
  o_clk=n_clk;
  n_clk=msg.value();
}
...
if(o_clk==0 && n_clk==1) { // if rising_edge(clk)
   _q1=_1164not(_d1);  //port buffer code for d1 d2 d3 d4
   _q2=_1164xor(_d2,_d1);
    q3=_1164xor( d3, 1164and( d1, d2));
   _q4=_1164xor(_d4,_1164and(_d3,_1164and (_d1,_d2)));
   holdIn(active,0);
}
```

FIGURE 7.30 Translating process models.

```
signal my_signal, x, y, z: std_logic;
…
my_signal<=x after 5ns;
```

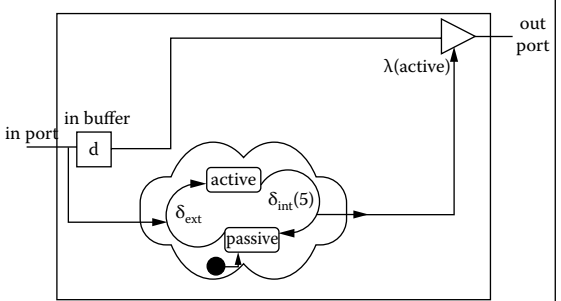


FIGURE 7.31 Signal model and DEVS definition.

$$\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} = q(x)$$

can be written as two first-order differential equations:

$$\frac{dy}{dx} = z(x),\ \frac{dz}{dx} = q(x) - p(x)z(x).$$

In general, an *n*th-order ordinary differential equation of form

$$F(t, y, y', y'', ..., y^{(n)}) = 0 \tag{7.1}$$

may be decomposed into a set of first-order differential equations:

$$\frac{dy_i(t)}{dt} = f_i(t, y_1, ..., y_N),\quad i = 1, ..., N \tag{7.2}$$

where each $f_i(t, y_1 \ldots y_N)$ is known. A solution for each $y_i(t)$ is obtained for some $t > 0$ and $y_i(0)$ set by integrating each $dy_i(t)/dt$. We have used both Euler's and fourth-order Runge–Kutta (which is more accurate and stable) methods for the numerical integration [13] combined with quantized DEVS. The Runge–Kutta method not only relies on the derivative at the beginning of the interval but also uses the derivative at two trial midpoints and the derivative at a trial end point, as seen in Figure 7.32.

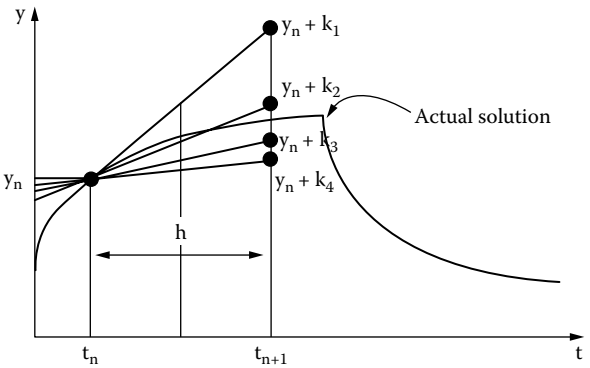


FIGURE 7.32 Runge–Kutta integration.

The idea is to compute a weighted sum of k_1, k_2, k_3, and k_4 that is added to y_n to determine y_{n+1}:

$$k_1 = hf(t_n, y_n),\ \ k_2 = hf\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right),\ \ k_3 = hf\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right),$$
$$k_4 = hf(t_n + h, y_n + k_3),\ \ y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \tag{7.3}$$

In order to use the fourth-order Runge–Kutta method in a quantized state system, this equation must be modified to determine h when $y_{n+1} - y_n = Q/2$ (Q is the quantum size). Then substitute $k_1 = Q/2$, $k_2 = Q/2$, $k_3 = Q/2$, and $k_4 = Q/2$, in (7.3) to get

$$h_1 = \frac{\frac{Q}{2}}{f(t_n, y_n)}$$

$$h_2 = \frac{\frac{Q}{2}}{f\left(t_n + \frac{h_1}{2}, y_n + sign(h_1)\frac{Q}{4}\right)}$$

$$h_3 = \frac{\frac{Q}{2}}{f\left(t_n + \frac{h_2}{2}, y_n + sign(h_2)\frac{Q}{4}\right)} \tag{7.4}$$

$$h_4 = \frac{Q}{f\left(t_n + h_3, y_n + sign(h_3)\frac{Q}{2}\right)}$$

If we rearrange the sum in the original equation, and substitute for k_1, k_2, k_3 and k_4 we obtain:

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \quad y_{n+1} - y_n = \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \left| y_{n+1} - y_n \right|$$

$$= \frac{Q}{2} = \left| h \left[\frac{\frac{Q}{2}}{h_1 6} + \frac{\frac{Q}{2}}{h_2 3} + \frac{\frac{Q}{2}}{h_3 3} + \frac{\frac{Q}{2}}{h_4 6} \right] \right| h = \left| \left[\frac{1}{h_1 6} + \frac{1}{h_2 3} + \frac{1}{h_3 3} + \frac{1}{h_4 6} \right]^{-1} \right| \tag{7.5}$$

This equation determines at what time (relative to the present time) the integral of the first order differential equation will cross a threshold (i.e., it will enter the quantized state above or below its current quantized state).

The conversion process must first determine which quantities and signals are exogenous and endogenous to the ODE system. Endogenous quantities are those on the left-hand side of the simultaneous statement as well as all quantities on the right-hand side of the simultaneous statement with the same quantity name as the left-hand side quantity. All other quantities or signals will be exogenous. For example, the following simultaneous statement describes a first-order low-pass filter with input voltage *vin* and output voltage *vout:*

```
vout'dot = (1/(R*C))*(vin-vout);
```

In this statement, *vin* is an exogenous quantity, and *vout* and *vout′dot* are endogenous. In the previous example, the *′dot* notation denotes the derivative with respect to time of the quantity. For example, *signal′dot* is the first derivative with respect to the time of the signal, and *signal′dot′dot* is the second derivative. Once all endogenous and exogenous quantities and signals have been identified, the ODE specified in the simultaneous statement must be decomposed into a set of first-order differential equations as outlined in Equation (7.2). Each of these equations is then converted into a fourth-order Runge–Kutta quantized integrator. Each integrator must have an input port for each exogenous and endogenous quantity or signal on the right-hand side of its first-order differential equation and an output port for the integral of the left-hand side of its first-order differential equation. For example, the preceding low-pass filter requires only a single integrator with input ports for *vin* and *vout,* as well as an output port for *vout.*

If the model is passive and it receives an input, the integrator's external transition function is triggered, and the function computes Runge–Kutta integration for a quantized state system. The right-hand side of the first-order differential equation is converted to C++, substituting the signal buffer name for the signal name and multiplying this buffer by the quantum size.

The following is the fourth-order Runge–Kutta method code for the low-pass filter presented previously:

```
p1 = (1.0/(C*R))*(_vin*QuantumSize -  (_vout*QuantumSize));
p2 = (1.0/(C*R))*(_vin*QuantumSize -  (_vout*QuantumSize + sign(p1)*(HalfQuantumSize/2.0)));
p3 = (1.0/(C*R))*(_vin*QuantumSize -  (_vout*QuantumSize + sign(p2)*(HalfQuantumSize/2.0)));
p4 = (1.0/(C*R))*(_vin*QuantumSize -  (_vout*QuantumSize + sign(p2)*(HalfQuantumSize)));

h1 = HalfQuantumSize / p1;
h2 = HalfQuantumSize / p2;
h3 = HalfQuantumSize / p3;
h4 = HalfQuantumSize / p4;

h = 1.0/(1.0/(6.0*h1) + 1.0/(3.0*h2) + 1.0/(3.0*h3) + 1.0/(6.0*h4));
```

```
entity LPF is
   port (terminal tout, tgnd:
electrical);
   end entity LFP;

architecture top of LPF is
  signal clk : std_logic;
  signal vin : std_logic;
quantity vout across tout to tgnd;

begin
vout´dot = (1/(R*C))*(vin-vout);
clk: entity clk
port map (clk=>clk);
vin<=clk;
end architecture top;
```

```
components : int@rkIntegModel clock
out : clk y
Link : y@int y
Link : y@int dydt@int
Link : out@clock clk
Link : out@clock vin@int

[int]
y0 : 0
dydt0 : 0
C : 1.0E-6
R : 1000

[clock]
components : inv@Process_Inv sig1@Signal
components : qm@QuantumMultiply
out : out
Link : out@sig1 in@inv
Link : out@inv  in@sig1
Link : out@sig1 in@qm
Link : out@qm   out

[sig1]
Transport_Delay : 00:00:1:000

[qm]
Transport_Delay : 00:00:00:000
Attenuation : 100
```

FIGURE 7.33 Hierarchical sAMS-VHDL model and translation to CD++.

The model then transitions to the active state for a time determined by h, which is calculated as in (7.4) and (7.5). The output function simply outputs the current state of the output buffer plus or minus one: plus one if the slope over the interval was positive and minus one if the slope over the interval was negative. The internal transition function similarly increases/decreases the state of the output buffer, depending on the slope over the interval, and then sends the model into the passive state.

Following compilation, the VHDL models' hierarchies are converted to CD++ coupled models. The components that constitute the design hierarchy must first be differentiated based on whether they are a basic or an aggregate component. Basic components do not contain subcomponent instances in their architectures; aggregate components may have one or more. Figure 7.33 contains the complete architecture defnition and the CD++ coupled model for the sAMS-VHDL design of Figure 7.28; note that the order of component declaration begins with the top-level model and is followed by models that approach the leaves in the dependency tree. As we can see, there are two basic components: a digital clock (a component built as the clock defined in Figure 7.30) and an integrator, built as in Figure 7.32.

sAMS-VHDL subcomponent instances are connected to the architecture in which they are instantiated as defined by the port map clause in their component instantiation statement. This clause will connect either a signal within the architecture or a port on the architecture's entity definition to each of the ports on the component instance. In the case of a signal, the linking is termed *structural;* in the case of another port, the linking is termed *hierarchical.* In both cases the mode of the subcomponent port specified in the *port* map clause must be determined prior to generating link statements in the coupled model definition. In structural links, if the ports mode is out, it is linked to the input port on the signal model specified in the clause; if the ports mode is in, the output port on the specified signal model is linked to it. In hierarchical links, if the subcomponents port mode is out, it is linked to the component port; if the subcomponents port mode is in, the component port is linked to it. Figure 7.33 illustrates all four of these cases.

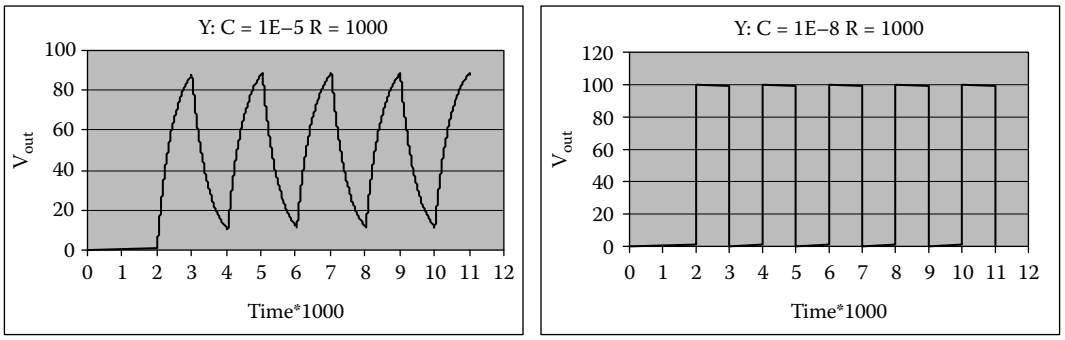


FIGURE 7.34 Simulation results: low-pass filter.

Once the complete model is defined and has been translated, it can be simulated in CD++. Figure 7.34 shows the execution results for the filter using different input parameters.

7.6 BOND GRAPHS

As discussed in Chapter 2, the continuous behavior of dynamic systems is usually described in terms of differential algebraic equations (DAEs), ordinary differential equations (ODEs), and partial differential equations (PDEs). Simulation based on these formalisms is done numerically by solving the set of equations describing the system and finding consistent initial conditions [14,15]. In recent years, new techniques have focused on how to apply concepts of system decomposition (i.e., to divide the system into a number of smaller subsystems interfaced by distinct connections).

One of them, the Bond Graph (BG), provides a modeling formalism and a graphical notation that allows domain-independent description of the dynamic behavior of continuous systems (i.e., a BG can be used to specify systems within the electrical, mechanical, thermodynamic, and hydraulic domains, etc.). BG allows hierarchical description of the system of interest, using BG submodels connected via ports through their interfaces [16].

A BG represents a system as a set of elements interacting with each other by an ideal exchange of energy, and this exchange determines the dynamics of the system. *Power* (the derivative of energy over time) is the product of *effort* and *flow*. For example, in electrical systems, power is the product of voltage and current, and in hydraulics, power is the product of pressure and volume flow rate. We can define generalized flow and effort variables whose product gives the power exchanged by the components for any system.

BG modeling concepts are based on two assumptions for dynamic systems representation using network-like descriptions: the Energy Conservation law and the use of a lumped approach. This allows the system properties to be separated from each other and then integrated using ideal connections that represent energy flow; (guarantee continuity and ensuring that no energy is generated or dissipated).

The physical processes are represented as vertices in a directed graph whose edges represent the ideal exchange of energy. The energy flow is represented via *bonds* with direction, and the elements exchange effort and flow through them. The exchange of power is assumed to occur through abstract entities called *energy ports*. One-port elements are components with one energy port (represented with a bond). Two-port elements have two energy ports (represented with two bonds). Interactions between components are also restricted, and the connectors implement constrained exchanges between elements (Figure 7.35).

BG models are noncausal. Nevertheless, in order to compute the exchange of power between elements, we need causality (we cannot compute the values of the two power variables—effort and flow—at the same time). Causal analysis is essential to describe a BG model in computational terms

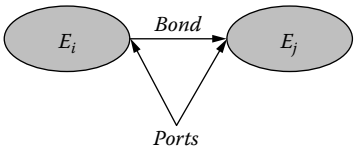


FIGURE 7.35 BG representation of energy flow from E_i to E_j.

and to derive the set of equations that represent the system, as seen in Figure 7.35. Therefore, given a pair of elements connected through a bond, a *causal* bond determines which of the components causes the flow and which causes the effort.

Bond graphs can represent a varied set of standard components, including *capacitors* (*c*), *inductors* (*i*), *resistors* (*r*), *effort sources* (*se*), *flow sources* (*sf*), *transformers* (*tf*), *gyrators* (*gy*), *1-junctions,* and *0-junctions.* We will briefly present some of these components (further details can be found in references 17–19):

- **Resistor (R element)** is a component with two terminals that resists flow (current in the electrical domain), producing effort to be reduced between the input and output terminals (voltage in the electrical domain) while dissipating energy. *R* elements can be used to model phenomena in varied domains (i.e., resistors in the electrical domain, dampers in the mechanical context, etc.). The constitutive equation is defined by an algebraic equation relating *flow* and *effort*: $e = R(f)$ (Figure 7.37). The electrical resistor is mostly linear and the corresponding equation is $\mu = R.i,$ where *R* is the resistance's constant.
- **Gyrator** is a two-port element and, like a transformer, it is power continuous (no power is stored or dissipated). The gyrator converts flow to effort and vice versa (e.g., an electrical motor). The gyrator establishes the relation between the effort on one side to the flow on the other and vice versa, indicated by $e_2 = \mu \times f_1$; $e_1 = \mu \times f_2$. In a gyrator, a vertical force produces motion in a horizontal direction (i.e., in a DC motor where the output torque is proportional to the input current, as defined by the equations in Figure 7.38).
- **Junctions** represent the constrained interactions between elements. Junctions couple components in a power-continuous way (with no energy dissipation or storage). Because there are only two ways in which components can exchange power, only two types of junctions are needed:
  - The *0-junction* (*parallel*) represents a node where all the efforts of the connecting bonds are equal (e.g., the parallel connections in electrical circuits). Power direction on a bond determines its flow sign (inward-pointing bonds: positive flow; outward-pointing bonds: negative flow). If we sum all the flows and consider the power direction, we obtain zero (corresponding to Kirchhoff's current law in electrical networks [16]) (Figure 7.39).
  - The *1-junction* (*serial*) represents a node where all the flows of the connecting bonds are equal—for instance, serial connections in electrical circuits (Figure 7.40). Due to power continuity, all the efforts sum to zero (considering the power direction associated with the bond). This summation is Kirchhoff's voltage law for electrical

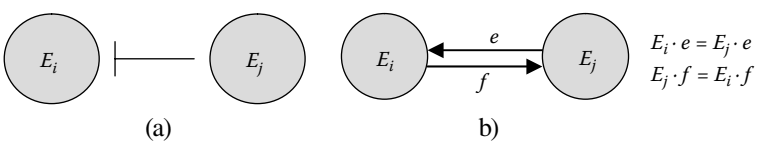


FIGURE 7.36 (a) Causal bond; (b) equivalent graph; (c) associated equations.

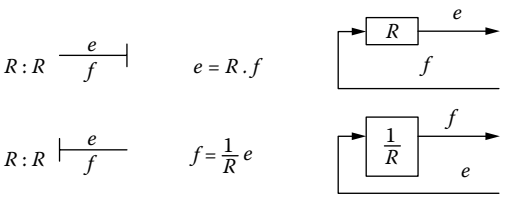


FIGURE 7.37 *R* element in causality, equations, and block diagram representation: flow; effort.

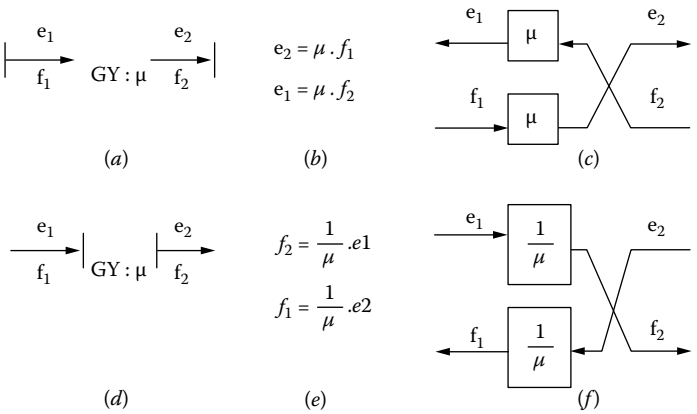


FIGURE 7.38 Gyrator element, related equations, and block diagram for the two causality types.

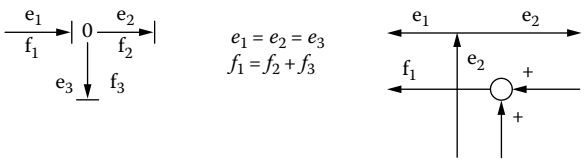


FIGURE 7.39 *0-junction* in causality, equations, and block diagram representation.

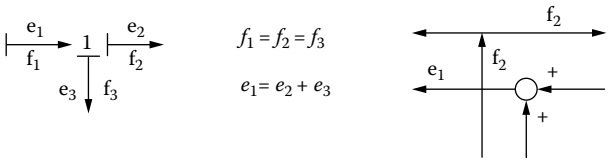


FIGURE 7.40 *1-junction* in causality, equations, and block diagram representation.

networks [20]: the sum of the voltages' differences on a closed loop (mesh) is zero (in the mechanical domain, for instance, 1-junctions represent the force balance, also known as the D'Alembert principle).

As we can see, every primitive BG element defines one or more equations that involve the flow or effort variable values received by the bonds connected to it. Bonds are two-signal connections (effort and flow) that have opposite directions. Passive elements like capacitors, resistances, and inductors have a power direction pointing inward; on the other hand, active components like sources have the power pointing outward. This signal direction determines the bond causality.

We used CD++ to build a library of BG primitive elements based on these concepts, using QDEVS and GDEVS models with polynomial functions of degree one. The components of the library are the following:

- *BG* is an abstract model used as a base for all the primitive Bond Graph elements. It introduces the basic functionality that permits adding new bonds to the components.
- *Bond:* Although this is not a Bond Graph primitive element, it was included to provide functionality beyond component connections. Different element types (one port, two port, or junction) can have one or more bonds associated. Every bond element has one input and one output port; these transport effort and flow variables between components. Attributes of the bond model specify power direction and causality restrictions.
- *Resistance* calculates an effort value according to the resistance equation (*effort* = *R.flow,* with *R* the resistance constant), computed when the flow value is received in an input port. The time instants of new input arrivals (t_1,…, t_n) are associated with the pair (a_i,b_i), which define the coefficients used to approximate the effort curve by the polynomial function: effort(t) = a_i t + b_i $\forall$ t $\in$ <t_i,t_j>. The model's internal transition implements the polynomial approximation of the continuous effort curve (this behavior is common to every GDEVS model in the library in which the curve approximation is done using a polynomial function of order one).
- *Capacitor* models the static relation between effort and displacement. Storage elements as capacitors impose a preferred causality.
- *EffortSource* and *FlowSource* generate signal values according to an emission frequency. *EffortSource* sends the effort through an output port, while *FlowSource* sends the flow value. Several signals were implemented in order to provide functions to be used in different contexts: *Constant, Step, Ramp, Sine, ExpSine, Exponential,* and *Pulse.*
- *Inductor* defines the static relation between flow and momentum. The model transition functions are similar to those used for the capacitor, but in this case, the inductor load (flow) is calculated as the integral of effort value.
- *Transformer* conserves power and transmits the power factors with the proper scaling defined by the transformer modulus. The modulus equation defines the following relations: $f_j = r\, f_i$ and $e_j = (1/r)\, e_i$, where r is the transformer modulus and (e_i,f_i) and (e_j,f_j) are the (*effort*,*flow*) values transported by $bond_i$ and $bond_j$ attached to the component. This element has two bonds connected to it, so both output effort and flow values must be calculated by the model.
- *Gyrator* establishes the relationship between flow to effort and effort to flow, keeping the power unchanged.
- *Junctions:* The 0-junction (1-junction) model processes the arrival of new effort (flow) data in the model's external transition function, sending the value received to all the output effort (flow) ports. On the other hand, the arrival of new flow (effort) by one of the bonds generates the recalculation of the equation. Once the value is recalculated, the flow (effort) is sent by the output port.

The complete hierarchy of Bond Graph models integrating the library was presented in D'Abreu and Wainer [17], and the library can be found in *./Hybrid.zip.* We will show how the capacitor model has been implemented (and the remaining components here described were defined following a similar approach). The *QBGCapacitor* is defined as in Kofman [21]:

$$QBGCapacitor = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{7.6}$$

where

$X = \Re$;

$Y = \Re \times N$;

$S = \Re^2 \times Z \times \Re^+$;
$\delta_{int}(c, d_c, j, \sigma) = (c + \sigma \times d_c, d_c, j + sign(d_c), \sigma_1)$;
$\delta_{ext}((c, d_c, j, \sigma), e, x_v) = (c + e \times d_c, x_v, j, \sigma_2)$;
$\lambda(c, d_c, j, \sigma) = (Q_{j+sign(dc)}, 1)$; and
$ta(c, d_c, j, \sigma) = \sigma$

$$\sigma_1 = \begin{cases} \dfrac{Q_j + 2 - (c + \sigma \times d_c)}{d_c} & \text{if } d_c > 0 \\ \dfrac{(c + \sigma \times d_c) - (Q_j - 1 - \varepsilon)}{|d_c|} & \text{if } d_c < 0 \\ \infty & \text{if } d_c = 0 \end{cases} \qquad \sigma_2 = \begin{cases} \dfrac{Q_i + 1 - (c + e \times d_c)}{x_v} & \text{if } x_v > 0 \\ \dfrac{(c + e \times d_c) - (Q_j - \varepsilon)}{|x_v|} & \text{if } x_v < 0 \\ \infty & \text{if } x_v = 0 \end{cases}$$

This model evolves based on the detection of effort value changes considering the input flow and the output effort as piecewise constant (and the displacement trajectory as piecewise linear). The model's state variables include the current capacitance and the previously computed value (two real numbers), a quantized state (an integer number), and the time when the next threshold change is scheduled to happen. Input flow changes are associated with external events. In case of input flow variations, the time to the next effort change must be recalculated. When the capacitor receives an input (x), it stores the input value, and it computes the new capacitance value as the current capacitance added to the product of the last input and the elapsed time. The current displacement is then computed according to the elapsed time and the previous flow value and then used as the new initial value. Then it schedules the next internal transition according to the formulas defined in σ_2. That is, if the input is positive, we compute the difference between the current capacitance value and the next threshold (Q represents the quantized signal). If the input is negative (i.e., there was a change of sign in the input), we check to see if the state change is larger than the hysteresis value ε. If the input is 0, we passivate the model.

When delay time is consumed, it means we have crossed a threshold, and we trigger the output function (which transmits the current threshold, depending on the sign of d_c). Then the internal transition function updates the capacitance value according to the time advance, and it schedules the next internal transition according to the update value and the sign of the update.

Figure 7.41 shows the implementation of this model in CD++. As we can see, when flow arrives at the component, an external transition function is activated and the flow is integrated in order to calculate the effort value, which is sent to the rest of the system through the effort port. The external transition function calculates the effort value as the integral of the input flow data, generating the capacitor's load. If the flow input arrives during an active state, the value is computed according to the elapsed time since the last internal transition function. An internal transition is immediately scheduled and will be in charge of computing the next state. Before executing the internal transition function, the output function transmits the previously computed value. The *quantizer* model provides the representation of output trajectories as piecewise constant functions through the quantization function.

The libraries were used to execute some examples of application [17]. For instance, the electrical circuit in Figure 7.42 is built as components connected in serial and parallel, and it can be used to measure current [22]. In order to simulate the circuit within CD++, the components in the diagram had to be replaced by the corresponding BG atomic models in the library. All the components were connected using input/output (effort/flow) ports, according to the causality defined by every element, generating a coupled model. The structure of the coupled model associated with the circuit just presented is shown in Figure 7.43.

```
QBGCapacitorFlowIn::QBGCapacitorFlowIn( const string &name )  : Atomic( name )
  , f1p( addInputPort( "f1p" ) )  , e1p( addOutputPort( "e1p" ) )   {

      string capacitance( MainSimulator::Instance().getParameter( description(), "C" ) );
      C = atof( capacitance.data() );

      string load( MainSimulator::Instance().getParameter( description(),"initialLoad" ) );

      initialLoad = atof( load.data() );

      string quant( MainSimulator::Instance().getParameter( description(), "quantum" ) );
      quantum = atof( quant.data() );

      string hystW( MainSimulator::Instance().getParameter( description(),"hystWindow" ) );

      hystWindow = atof( hystW.data() );

      quantizer = new ((UniformQuantizer *)quantizer)->setQuantum( quantum );
}


Model &QBGCapacitorFlowIn::initFunction() {
  c = quantizer->quantize( initialLoad );
  qValue = c + quantum;
  time = der = 0;
  init = true;
  eps1 = quantum/2;
  eps2 = ( quantum - hystWindow )/2;

  holdIn( active, Time::Zero );
}

Model &QBGCapacitorFlowIn::externalFunction( const ExternalMessage &msg ){

  RealValue currTime = msg.time().asMsecs();
  RealValue nextTime, e, func;
  RealValue val = msg.value();

  flow = val;
  e = currTime - time;
  val = val * TICK_VALUE;   // in ms
  val = val * 1/C;

  if( TRUNCATE( fabs( val ), TOLERANCE ) == 0 ) {
    time = currTime;
    c = c + der * e;
    der = 0;
    passivate();
  }
  else    {
    func = c + der * e;

    if( val > 0 ) {
      if ( der < 0 ) qValue += quantum;
      nextTime = fabs( ( qValue + eps1 - func ) / val );
    }
    else {
      if ( init ) {
        qValue -= quantum;
        init = false;
      }
      if ( der > 0 ) {
        qValue -= quantum;
      }
      nextTime = fabs( ( func - ( qValue - hystWindow - eps2 ) ) / val );
    }

    RealValue waitTime = nextTime * TICK_VALUE;

    time = currTime;
    c    = func;
```

FIGURE 7.41 Implementation of capacitor element in CD++.

```
    der   = val;
    nc    = c + der * nextTime;

    holdIn( active, Time( static_cast<float>( waitTime ) ) );
  }
}

Model &QBGCapacitorFlowIn::internalFunction( const InternalMessage &msg ) {
  RealValue currTime = msg.time().asMsecs();
  RealValue nextTime, e, func;

  e = currTime - time;
  func = c + der * e;

  if( TRUNCATE( fabs( der ), TOLERANCE ) == 0 )
    passivate();
  else
    if( der > 0 ) {
      qValue += quantum;
      nextTime = fabs( ( qValue + eps1 - func ) / der );
    }
    else {
      qValue -= quantum;
      nextTime = fabs( ( func - ( qValue - hystWindow - eps2 ) ) / der );
    }

    RealValue waitTime = nextTime * TICK_VALUE;

    time = currTime;
    c    = func;
    nc   = c + der * nextTime;
    holdIn( active, Time( static_cast<float>( waitTime ) ) );
  }
}

Model &QBGCapacitorFlowIn::outputFunction( const InternalMessage &msg ) {
  sendOutput( msg.time(), e1p, nc ) ;
}
```

FIGURE 7.41 (continued).

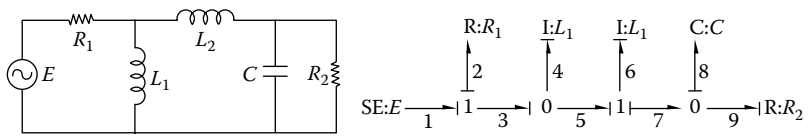


FIGURE 7.42 Circuit diagram and Bond Graph.

Figure 7.44 shows the simulation results for this example, using a period of 1 ms; resistance, R_1 = 1; inductors, $L_1 = 48$, $L_2 = 48$; capacitance, $C = 65$; conductance, $R_2 = 0.001$; EffortSource, emitting a pulse with a period of 2,500 ms and duration of 2 ms; and pulse amplitude = 220 V.

7.7 MODELICA

Many of the concepts introduced by Bond Graphs were adopted and applied to the design of various modeling and simulation tools for continuous systems modeling, and they were extended to include concepts of object-oriented modeling (OOM). OOM permits decreasing the abstraction gap between the real system and the representation model, which enables specifying the models in a

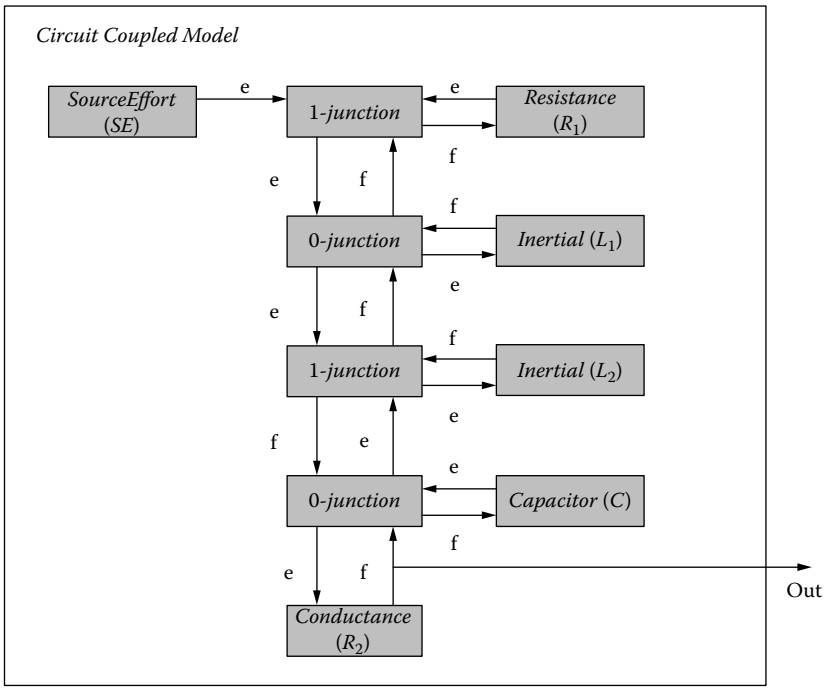


FIGURE 7.43 DEVS coupled model associated with the circuit.

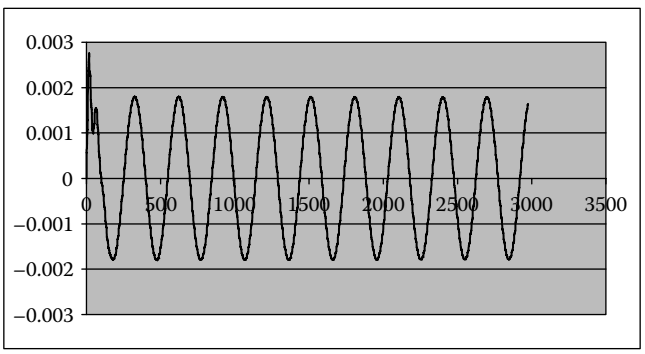


FIGURE 7.44 Circuit current.

more natural way while improving reusability of models in a hierarchical construction process. One such language is Modelica [23].

A model in Modelica is a noncausal construct defined by mathematical equations and OOM organization (using classes that can be developed hierarchically, allowing component reuse, library development, and model exchange). Modelica includes libraries of standard components in varied application domains including ODEs, block diagrams, electrical, hydraulics, mechanics, etc. The semantics of the models is specified by a set of rules used to translate the model to its corresponding flat hybrid DAE. An example of an electrical circuit specified using the electrical library provided by Modelica is presented in Figure 7.45.

The *circuit* model uses a pulse voltage generator (with a voltage of 200 V, a period of 1 Hz, and an amplitude of 10% of the frequency), a capacitor (with capacitance of 200 F), a resistor (with

```
model Ecircuit
  Modelica.Electrical.Analog.Sources.PulseVoltage V(V=200, period=1, width=10);
  Modelica.Electrical.Analog.Basic.Capacitor C(C=200);
  Modelica.Electrical.Analog.Basic.Resistor R(R=1.5);
  Modelica.Electrical.Analog.Basic.Inductor I(L=40);
  Modelica.Electrical.Analog.Basic.Ground Gnd;
 equation
  connect(V.p, R.p);
  connect(R.n, I.p);
  connect(R.n, C.p);
  connect(I.n, V.n);
  connect(C.n, V.n);
  connect(C.n, Gnd.p);
 end circuit;
```

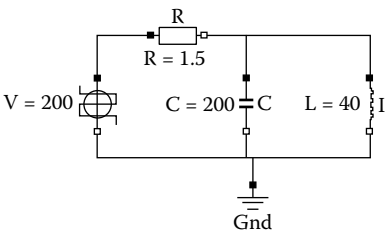


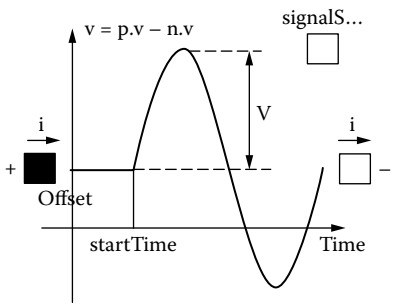
FIGURE 7.45 Modelica specification of a circuit.

resistance of 1.5 Ω), and one inductor (with inductance of 40 H). The circuit is connected to ground. The *connect* constructions permit defining the structure of the circuit. As we can see, we can have serial (*V.p, R.p*) or parallel connections (*R.n, I.p; R.n, C.p*).

D'Abreu and Wainer [24] presented the design and implementation of M/CD++, a tool to construct continuous systems based on Modelica, using DEVS as the underlying formalism and CD++ as the support tool. M/CD++ permits simulating electrical circuit models like the one in Figure 7.45 by implementing a subset of Modelica language specification. In particular, M/CD++ provides language support for a subset of Modelica v2.1, including the components needed to allow electrical circuit construction provided by the Modelica electrical library. These components are described according to Modelica specifications [23].

Figure 7.46 shows one of these components (a complete description of the grammar supported can be found in [24] and [25]). In this example, the sine voltage construction defines the amplitude, phase, and frequency (in hertz) of the sine wave (defaults: 1, 0, 1) and generates a voltage using those values (which vary over time).

M/CD++ is composed of a set of components to parse, compile, verify, and execute the model. The process starts with an electrical circuit model specified using Modelica and finishes with a




```
model SineVoltage "Sine voltage source"
  parameter SI.Voltage V=1 "Amplitude";
  parameter SI.Angle phase=0 "Phase ";
  parameter SI.Frequency freqHz=1"Frequency ";
extends Interfaces.VoltageSource(redeclare Modelica.Blocks.Sources.Sine
   signalSource(amplitude={V}, freqHz={freqHz}, phase={phase} ) );
end SineVoltage;
```

FIGURE 7.46 Definitions: Modelica.Electrical.Analog.Sources.SineVoltage.

CD++ log file including the simulation results. We also generate an intermediate BG to check for algebraic loops and singularities (elements that have discontinuities—e.g., diodes). Then we generate an optimized BG corresponding to the electrical circuit, which, in turn, is used to generate a DEVS simulation using the library introduced in Section 7.6. The following sections will give a brief introduction of the compiler, which is included in the *internal* folder in CD++Builder (and can be downloaded in *./mcd++.zip*). We will introduce the general ideas of each component (interested readers should consult [25]).

7.7.1 Modelica Parser

This component checks and parses the input file, building and validating the electrical circuit model. We used a general-purpose parser generator that takes an LALR context-free grammar and describes the actions that accompany the syntactic rules. These actions are used to build a syntax tree corresponding to the model's input file, which is in turn used to perform semantic validation and electrical circuit construction. In this stage we check:

- specification of valid and supported packages (i.e., those in the electrical library);
- specification of valid and supported types and classes;
- undeclared symbols; and
- specification of valid component attributes.

If the complete syntax tree is successfully validated, we build an electrical circuit. Several verifications are considered in order to be able to preserve the model properties.

The definitions of *pin* (*positive* and *negative*), *port, one-port element, two-port element, electrical component* (*resistance, capacitor, source,* etc.), and *circuit* generate the model associated with these components. Every electrical component on a circuit is represented as a graph using *n* nodes (where *n* corresponds to the number of pins of the element). *One-port* elements are represented by two nodes: *element.port*$_1$*.p* (positive pin) and *element.port*$_1$*.n* (negative pin). *Two-port* elements are represented by four nodes: *element.port*$_1$*.p*, *element.port*$_1$*.n, element.port*$_2$*.p* and *element.port*$_2$*.n*. Generalizing, *k-port* elements will be represented by *2.k* nodes as: *element.port*$_1$*.p, element.port*$_1$*.n, …, element.port*$_k$*.p,* and *element.port*$_k$*.n,* as seen in Figure 7.47. There are two types of connections between nodes: physical and logical. The former corresponds to the physical coupling between elements of the circuit (solid lines). Logical connections correspond to the associations between pins and ports of an element; the pins of a given port connector are linked by dashed lines and port connectors of a given component are linked by dotted lines.

Figure 7.48 shows the electrical circuit objects constructed by the parser given the corresponding Modelica specification file presented in Figure 7.46. The electrical circuit object, *EC* (E Circuit), is modeled as the composition of *R* (an instance of *Resistance,* which is a *one-port* element), *V* (an instance of *VoltageSource* with signal *s,* a *one-port* element), *C* (a *Capacitor* component, a *one-port* element), *I* (an *Inductor,* a *one-port* element), and *Gnd* (an instance of *Ground,* an element with one positive pin).

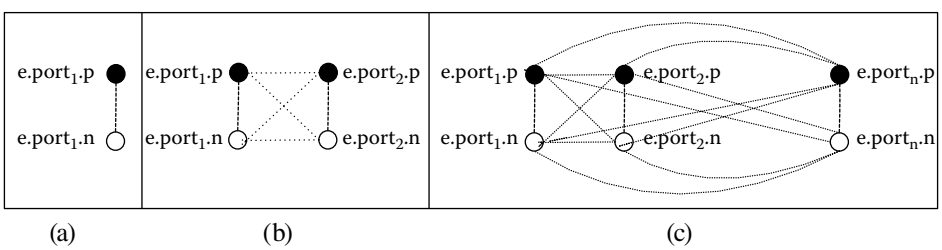


FIGURE 7.47 Node representation of port elements: (a) one-port; (b) two-port; (c) k-port elements.

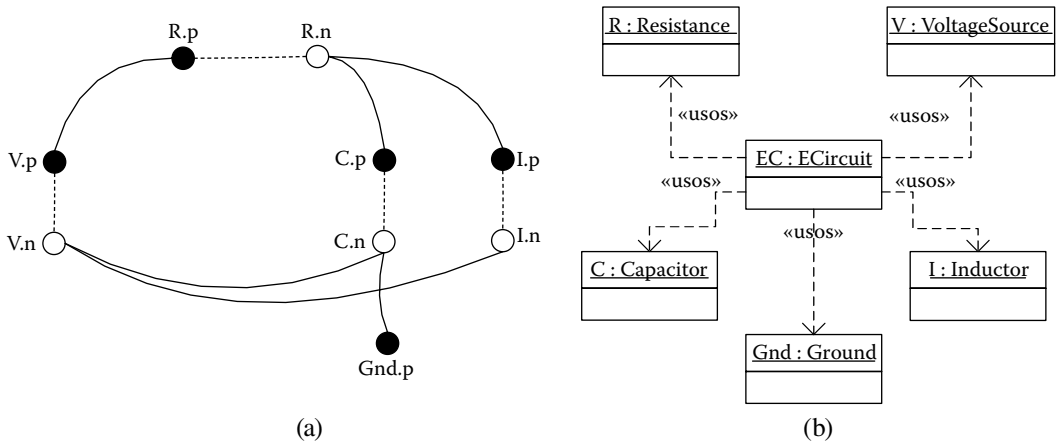


FIGURE 7.48 (a) Electrical circuit graph representation; (b) objects model generated by the M/CD++ parser.

```
model circuit
Modelica.Electrical.Analog.Sources.PulseVoltage V(V=10,width=50,period=2.5);
Modelica.Electrical.Analog.Basic.Resistor R1(R=0.001);
Modelica.Electrical.Analog.Basic.Inductor I1(L=500);
Modelica.Electrical.Analog.Basic.Inductor I2(L=2000);
Modelica.Electrical.Analog.Basic.Capacitor C(C=10);
Modelica.Electrical.Analog.Basic.Resistor R2(R=1000);
Modelica.Electrical.Analog.Basic.Ground Gnd;

Equation
 connect(V.p, R1.p);
 connect(R1.n, I1.p);
 connect(R1.n, I2.p);
 connect(I2.n, C.p);
 connect(I2.n, R2.p);
 connect(C.n, I1.n);
 connect(R2.n, C.n);
 connect(I1.n, V.n);
 connect(V.n, Gnd.p);
end circuit;
```

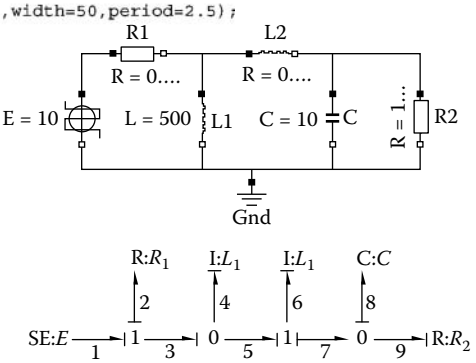


FIGURE 7.49 (a) Modelica specification of a circuit; (b) circuit; (c) generated BG.

The electrical circuit is modeled using the internal graph introduced (Figure 7.49(a)). This data structure and the model representation can help in the following phase, when we define a BG generation algorithm.

After the BG is constructed and simplified, we apply different error detection techniques to the resulting BG. **Causalization** is the process where the signal direction of the bonds is determined. Once this process is applied to the graph, each bond can be interpreted as a bidirectional signal flow. A port element can impose four different causal constraints on its connected bonds:

- *Fixed causality* appears when the equations allow only one of the two port variables to be the outgoing variable—that is, source effort (*Se*) and source flow (*Sf*) components.
- *Constrained causality* appears when relations between causalities of the different ports within the component define causal constraints. For instance, at 0-junctions (where all efforts are equal), exactly one bond has flow causality (flow-out causality). The causal constraint at a 1-junction is the dual form of the 0-junction. At a Transformer element, one bond has effort causality (effort-out causality) and the other flow causality. At Gyrators, both bonds have either effort causality or flow causality.

- In *preferred causality,* the causality on storage elements determines whether integration or differentiation with respect to time will be used. Integration has preference above differentiation, representing the preferred causality. Then, at *C* elements, the preferred causality is effort causality and, at *I* elements, flow causality.
- *Arbitrary causality* is used when no causal constraints exist (i.e., at *R* elements).

7.7.2 Mapping Electrical Circuits to BG

Our BG generation algorithm is based on Karnopp's circuit construction method [26]. We have applied the Sequential Causality Assignment Procedure (SCAP) to assign causality to the bonds of a given BG. The method starts by choosing a fixed causality element (*source*) and then propagates the assignment through the structural components (*Junctions*, *Transformer*, and *Gyrator*), according to causality restrictions. Once all sources have been processed, a storage element (*C* or *I*) is selected and the preferred causality applied, restarting the propagation step. That is repeated until all storages have their causalities assigned. Last, if the graph is not completely causalized, the iteration is repeated beginning with a resistor (*R*). If the last step is reached, the model contains algebraic loops.

The idea is to construct a BG that resembles the circuit structurally, simplifying the BG based on selected circuit properties. The construction method is as follows:

- For each node in the circuit with a distinct potential, write a 0-junction.
- Insert each one-port circuit element by adjoining it to a 1-junction and inserting the 1-junction between the appropriate 0-junctions (*C, I, R, Se,* and *Sf* elements).
- Assign power directions to all bonds.
- If the circuit has an explicit ground potential, delete those 0-junctions and their bonds from the graph. If no explicit ground potential is shown, choose any 0-junction and delete it.
- Simplify the resulting BG.

Once this process is applied to the graph, each bond can be interpreted as a bidirectional signal flow. Structural singularities and algebraic loops in the model are detected. Figure 7.49 shows a graphical representation of an electrical circuit and its transformation to a BG.

7.7.3 BG Compiler for CD++

Once the BG model has been generated and causalized, it is transformed into a DEVS model. The first step of the BG compilation is the transformation to its equivalent quantized BG (QBG); that is, it is converted to a BG where all the storages and sources are quantized BG elements. To do so, the compiler follows these steps:

- For each component **u** of the QBG, add **u** to the declaration section within the CD++ coupled model file.
- Select a valid implementation class for the component.
- For each bond, **b = (u,v)**, of the QBG, generate the coupling information between **u** and **v** of the links section within the CD++ coupled model.
- For each component **u** of the QBG, generate the component's configuration information within the parameterization section of the CD++ coupled model file.

Figure 7.50 shows the translation of the model presented in Figure 7.49 into CD++.

7.7.4 Simulation Examples

In this section, we present the various simulation results of different electrical circuits using M/CD++. We first present the results of the circuit introduced in Figure 7.49, using a simulation run of 1 min, a quantum = 0.0002, and an hysteresis window size = 0.01 applied to all of the quantizable components within the circuit (I_1, I_2, C_1).

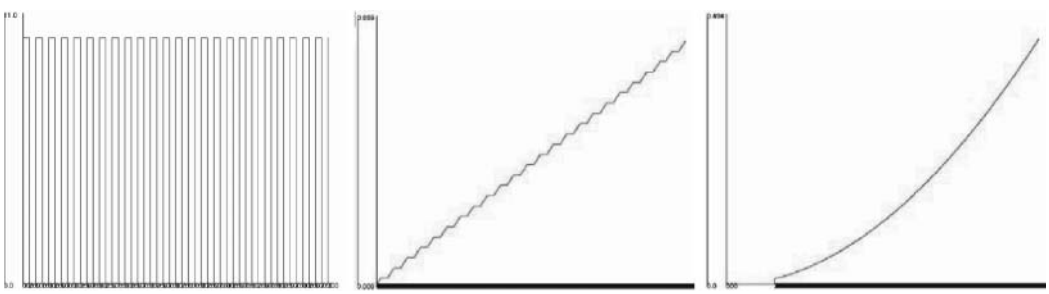
Figure 7.51 shows the simulation results for the model executed. For the given pulse voltage source, we obtained the desired voltage on capacitor C and the expected current on the inductor I_i. In order to check the results obtained by M/CD++, we compared them with simulations executed using *Dymola* [27], a commercial toolkit with full support for Modelica (Dymola, version 5.1b, whose simulator, *Dymosim,* provides a number of different integration methods for the simulation of dynamic systems). We used a fixed-step algorithm (*Euler,* a first-order algorithm) and a variable step-size algorithm (*DASSL* with variable order 1–5) used to integrate DAE and ODE systems [28].

When we simulated the previous example using the DASSL integration method on Dymola (10 s of simulation time, intervals of 500 time units, and tolerance of 0.0001) and compared the results with those obtained by M/CD++ (using a quantum size $q = 0.0001$ and a hysteresis window of q/2), we obtained the information shown in Figure 7.52.

Figure 7.53 shows a different example that we simulated using the DASSL integration method on Dymola for 60 s of simulated time, with 500 intervals and a tolerance of 0.0001. We compared the results obtained with a QSS DEVS model implemented by M/CD++.

```
components : $PJ2@QBGParallelJunction $PJ3@QBGParallelJunction C@QBGCapacitorFlowIn
$SJ2@QBGSerialJunction $SJ3@QBGSerialJunction ...

link : e2n@$PJ2 e2p@$SJ2
link : f2p@$SJ2 f2n@$PJ2
...
link : e1n@V    e3p@$SJ3
link : e3n@$PJ2 e1p@I1

[C]  C : 10.000  initialLoad : 0.000  quantum : 0.0002   hystWindow : 0.01
[L1] I : 500.000 initialLoad : 0.000  quantum : 0.0002   hystWindow : 0.01
[L2] I : 2000.00 initialLoad : 0.000  quantum : 0.0002   hystWindow : 0.01
[R1] R :    0.001
[R2] R : 1000.000
[V]  signal : Pulse  offset : 000  startTime : 000  amplitude : 010 period : 2.5  width :
050
quantum : 0.0002   hystWindow : 0.01
```

FIGURE 7.50 Coupled DEVS model representation and CD++ notation.



(a) (b) (c)

FIGURE 7.51 (a) Pulse voltage source; (b) Current on inductor I_1; (c) voltage on capacitor C.

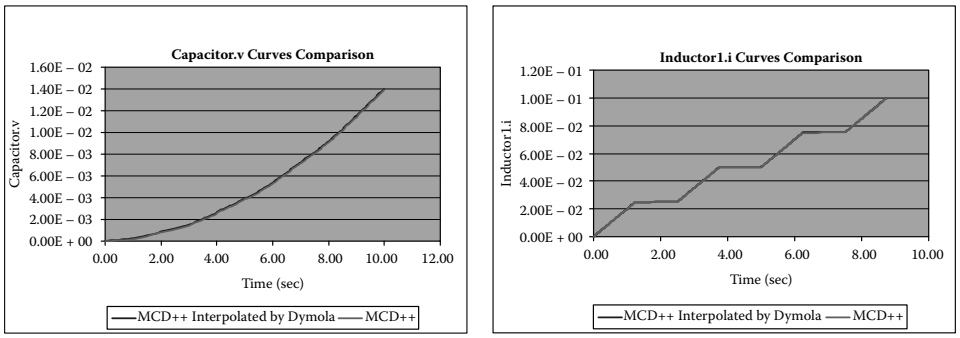


FIGURE 7.52 Comparison for voltage on capacitor C and current on inductor I_1.

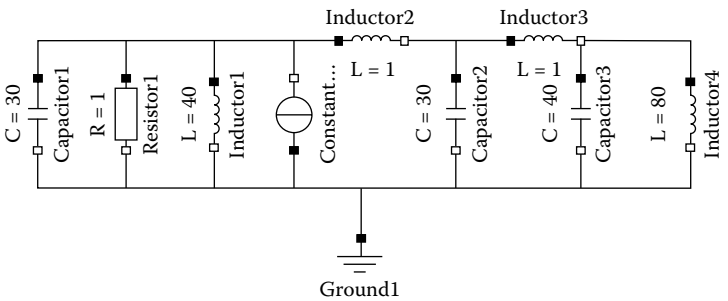


FIGURE 7.53 Model of an electrical circuit.

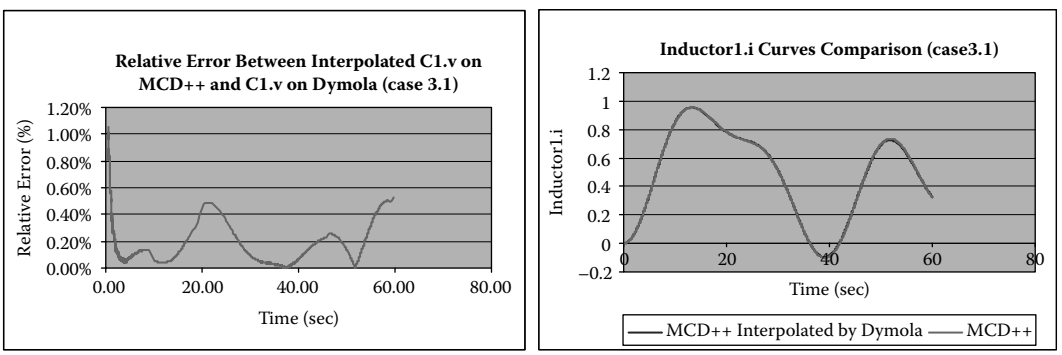


FIGURE 7.54 (a) Relative error for v. on C_1; (b) current on I_1.

Figure 7.54 shows the error for the capacitor (C_1) and the state trajectories for the inductor (I_1) on M/CD++ and Dymola. Figure 7.54(a) shows the relative error curve for voltage on capacitor C_1 and Figure 7.54(b) shows the trajectories for the current on inductor I_1. It can be seen that M/CD++ approximates the model trajectories well and that the relative error is constrained (below 0.5%). Larger relative errors are obtained for points near zero, given the fixed quantum size used through the entire simulation.

We also defined the electrical circuit in Figure 7.55. We show two test cases executed for this model, varying the integration method used in the simulation with Dymola. Initially, we simulated this sample circuit using the DASSL integration method on Dymola. The simulation time was 30 s, and we used 500 intervals at a precision of 0.0001. We also repeated the studies using the

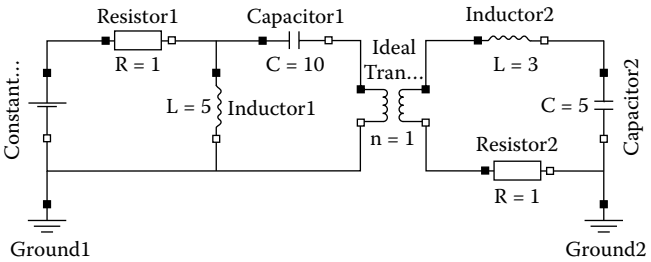


FIGURE 7.55 Electrical circuit.

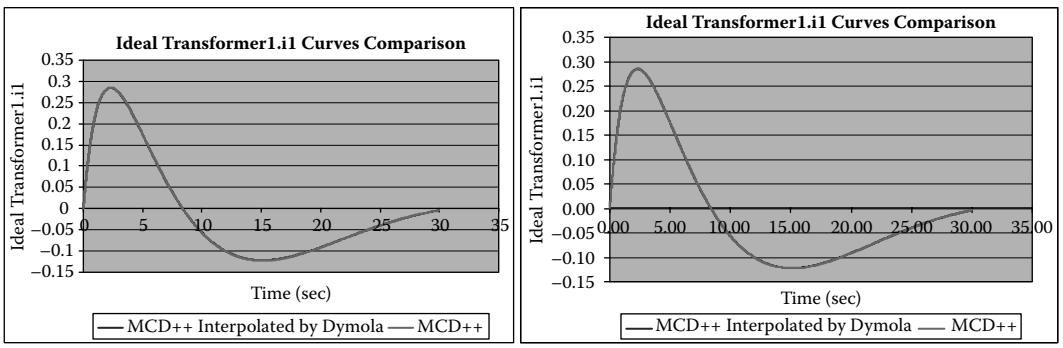


FIGURE 7.56 Current on the ideal transformer using (a) DASSL and (b) Euler.

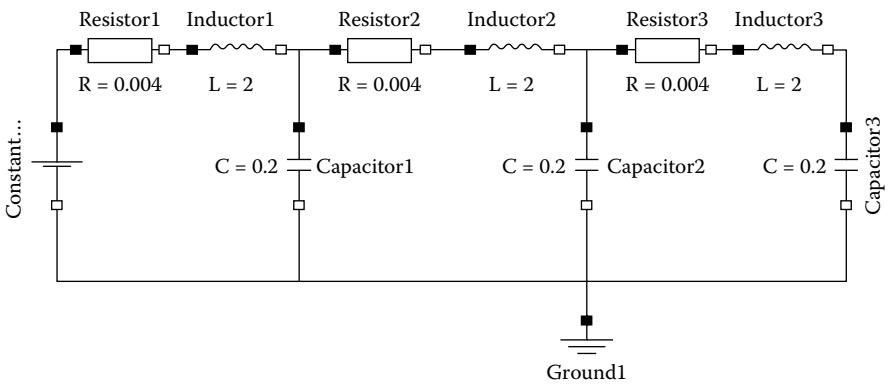


FIGURE 7.57 Electrical circuit.

Euler integration method on Dymola, using an integration step of 0.005 and a tolerance of 0.0001. Figure 7.55 shows the error between M/CD++ and Dymola for the current trajectories on the ideal transformer (input and output flow). A similar error to the one produced in the previous case is given using the Euler integration method on Dymola, with a step size equal to 0.005.

Finally, we present the results of the sample circuit in Figure 7.57. Two test cases were executed for this model, varying the quantization parameters used for state trajectories on M/CD++. We simulated this sample circuit using the DASSL integration method on Dymola for 15 s. Again, we used 500 intervals with a precision of 0.0001. Figure 7.58 shows the error, for the state trajectories on capacitor (C_1) and inductor (I_1), between M/CD++ and Dymola. This test case was simulated

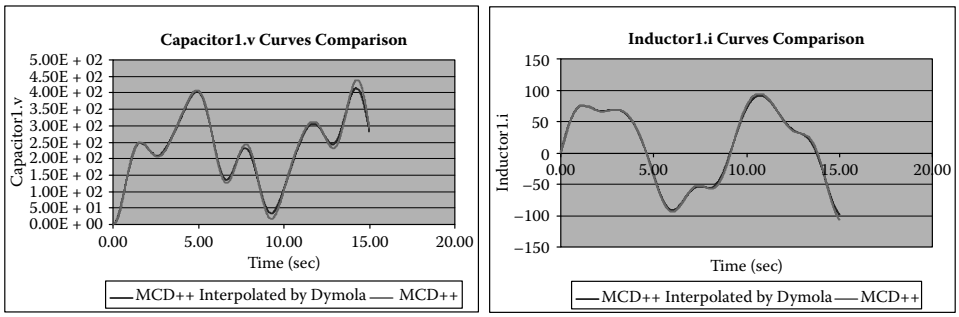


FIGURE 7.58 (a) Voltage curve on C_j; (b) current curve on I_1.

using the DASSL method on Dymola and decreasing the quantum and hysteresis window size on M/CD++ simulation.

Exercise 7.13

Repeat the previous examples, changing the quantum size and the hysteresis window. Analyze the new simulation results.

7.8 SUMMARY

In this chapter we introduced varied modeling techniques and their mapping to DEVS. We first showed how to define models of finite state machines and discussed some basic examples. We then introduced Petri Nets and Layered Queuing Networks, including a number of models to simulate them in CD++. We also introduced a variety of models of continuous systems based on GDEVS and QSS approximations, including a simplified version of VHDL-AMS, Bond Graphs, and Modelica. Our BG library allows users to create advanced models of continuous systems in different fields of applications and to integrate them with discrete-event models (like the PN and DEVS models introduced in previous sections) within the context of a coupled model definition.

This library was used as the starting point to create a Modelica compiler that enables the user to create models in Modelica and to execute them using CD++. We also showed the use of DEVS to facilitate simulation of mixed-signal HDL models using VHDL. In order to permit the execution of these models within a DEVS simulator, generic DEVS models and conversion procedures were required. Hierarchical models written in sAMS-VHDL that utilize processes, signals, and simultaneous statements may be simulated in CD++ by elaborating the model and converting the model hierarchy into an equivalent CD++ model.

Although they were not discussed in this section, other libraries have been built for different modeling techniques, including timed Petri nets (found in *./Petri-Timed.zip*), queuing networks (*./queuingmodels.zip*), Mealy finite state machines (*./fsm.zip*), a cellular automata-based Turing machine model (*./TuringMachine.zip*), and quantum dot cells (*./BrainMachine.zip*). The nature of DEVS permitted the creation of these multiple methods, seamless integration between models' components, and integration with continuous signal models into a hierarchical model definition.

REFERENCES

1. Vangheluwe, H. L. M. 2000. DEVS as a common denominator for multiformalism hybrid systems modeling. *Proceedings of Computer-Aided Control System Design, 2000, IEEE International Symposium on CACSD 2000,* Anchorage, AK, 129–134.
2. Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.

3. Zeigler, B. P., and S. Vahie. 1993. DEVS formalism and methodology: Unity of conception/diversity of application. *Proceedings of WSC '93: Proceedings of the 25th Winter Simulation Conference,* Los Angeles, CA, 573–579.
4. Zheng, T., and G. Wainer. 2003. Implementing finite state machines using the CD++ toolkit. *Proceedings of the 2003 SCS Summer Computer Simulation Conference,* Montreal, Quebec, Canada.
5. Peterson, J. L. 1981. *Petri net theory and the modeling of systems.* Englewood Cliffs, NJ: Prentice Hall.
6. Jacques, C., and G. Wainer. 2002. Using the CD++ DEVS toolkit to develop Petri nets. *Proceedings of Summer Computer Simulation Conference,* San Diego, CA.
7. Neilson, J. E., C. M. Woodside, D. C. Petriu, and S. Majumdar. 1995. Software bottlenecking in client–server systems and rendez-vous networks. *IEEE Transactions on Software Engineering* 21 (9): 776–782.
8. Woodside, C. M., J. E. Neilson, D. C. Petriu, and S. Majumdar. 1995. The stochastic rendezvous network model for performance of synchronous client–server-like distributed software. *IEEE Transactions on Computers* 44 (1): 20–34.
9. Woodside, C. M., S. Majumdar, J. E. Neilson, D. C. Petriu, J. Rolia, A. Hubbard, and G. Franks. 1995. A guide to performance modeling of distributed client-server software systems with layered queuing networks. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada.
10. Petriu, D., and G. Wainer. 2004. A library of layered queuing networks using the DEVS formalism. *Proceedings of Mediterranean Multiconference on Modeling and Simulation,* Bergeggi, Italy.
11. Christen, E., K. Bakalar, A. Dewey, and E. Moser. 1999. DAC'99 VHDL-AMS tutorial. *Proceedings of 36th Design Automation Conference,* New Orleans, LA.
12. Mehta, S., and G. Wainer. 2005. sAMS-VHDL: A tool for modeling hybrid hardware description languages. *Proceedings of the 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference,* San Diego, CA.
13. Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1986. *Numerical recipes.* Cambridge: Cambridge University Press.
14. Taylor, M. 1996. *Partial differential equations: Basic theory.* New York: Springer–Verlag.
15. Brenan, K. E., S. L. Campbell, and L. R. Petzold. 1989. *Numerical solution of initial-value problems in differential algebraic equations.* New York: Elsevier.
16. Åström, K. J., H. Elmqvist, and S. E. Mattsson. 1998. Evolution of continuous-time modeling and simulation. *12th European Simulation Multiconference, ESM'98*, Manchester, UK.
17. D'Abreu, M., and G. Wainer. 2003. Defining hybrid system models using DEVS quantization techniques. *Proceedings of the Winter Simulation Conference,* New Orleans, LA.
18. Cellier, F. E., and E. Kofman. 2006. *Continuous system simulation.* New York: Springer Science+ Business Media.
19. Samantaray, A. 2007. About bond graph—The system modeling world. URL: http://www.bondgraph.info/about.html
20. Paul, C. R. 2001. *Fundamentals of electric circuit analysis.* New York: John Wiley & Sons.
21. Kofman, E. 2003. Discrete event based simulation and control of continuous systems. PhD thesis, Universidad Nacional de Rosario, Argentina.
22. Banerjee, S. 2003. Dynamics of physical systems—The language of bond graphs. URL: http://www.Ee.Iitkgp.Ernet.In/~soumitro/dops/chap4.Pdf
23. Banerjee, S. 2005. *Dynamics for Engineers*. New York: Wiley.
24. D'Abreu, M., and G. Wainer. 2005. M/CD++: Modeling continuous systems using Modelica and DEVS. *Proceedings of MASCOTS 2005,* Atlanta, GA.
25. D'Abreu, M. 2004. Defining a compiler for discrete-event simulation of continuous systems. MSc thesis, Computer Science Dept., Universidad de Buenos Aires, Argentina.
26. Karnopp, D., D. Margolis, and R. Rosenber. 1990. *System dynamics: A unified approach.* New York: Wiley-Interscience.
27. Dynasim Laboratories. 2004. Dymola. Available online via: http://www.Dynasim.com/dymola.htm
28. Petzold, L. R. 1993. A description of DASSL: A differential/algebraic system solver. *IMACS Transactions Scientific Computing* 1:65–68.
29. Mehta, S., and G. Wainer. 2005. SAMS-VHDL: A tool for modeling hybrid hardware description languages. *Proceedings of the 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference*. San Diego, CA.

Section 3

Applications

8 Applications in Biology

8.1 INTRODUCTION

This chapter will focus on different applications in the field of biology (and medicine), one of the most popular areas for the use of simulation. The complexity of biological processes makes computer simulation an adequate tool to study them under particular experimental conditions. Likewise, DEVS (a discrete-event hierarchical and modular formalism) is ideal for describing these systems, which are hierarchical and asynchronous in nature. DEVS also uses explicit timing information; hence, we can adequately represent timing of the reactions occurring in the organisms.

We will introduce different models at the organelle level. We will begin by presenting a model of the interaction between synapsin and vesicles in nerve cells. We will then discuss a model that defines various reactions in the liver, whose design demonstrates the process of substance transformations occurring within the liver's lobule. We then introduce a model of bacteria in food and the spreading of viruses in mobile populations. After that, we introduce a detailed model of the heart tissue, and we conclude with a model of the biological pathways in mitochondria. The models presented show how to use our simulation environment for biological models, and the results show the potential for creating more advanced applications in this area.

8.2 SYNAPSIN AND VESICLE INTERACTION IN A NERVE CELL USING CELL-DEVS

Synapsin is a neuron-specific phosphoprotein that binds to small synaptic vesicles and actin filaments in a phosphorylation-dependent pattern [1]. Microscopic models have demonstrated that synapsin inhibits neurotransmitter release either by forming a cage around synaptic vesicles (cage model) or by anchoring them to the F-actin cytoskeleton of the nerve terminal (cross-linking model) [2].

The model presented here (previously introduced in Bain et al. [2] and Wainer et al. [3] and found at *./nerveCell.zip*) describes the behavior of the reserve pool of synaptic vesicles in a presynaptic nerve terminal. It can be used to predict the number of synaptic vesicles released from the reserve pool as a function of time under the influence of action potentials at different frequencies [2,3]. The actual biochemistry of the terminal incorporates five key components: vesicles, synapsin (a protein that regulates neurotransmitter release), kinase (an enzyme that transfers phosphate from high-energy to lower-energy molecules), phosphatase (an enzyme that removes phosphate), and actin (a protein), which interact to produce exocytosis and endocytosis. Endocytosis, shown in Figure 8.1(a), is a process where cells absorb molecules (i.e., proteins) from the outside. Exocytosis is the opposite process: an intracellular vesicle moves to plasma, as seen in Figure 8.1(b), permitting a cell to release large molecules (for instance, to eliminate waste and in signaling).

Our model focuses on the molecular interaction of *synapsin* (**S**) with *vesicles* (**V**) that occur inside a nerve cell, and it describes the behavior of synapsin movements until reaching a vesicle and binding to it [2]. Once a binding has occurred, they can separate again and break their bindings. Two values, the *onrate* and *offrate,* describe how often bindings occur or break. The following formula describes the nature of the reaction:

$$S + V \leftrightarrow SV$$

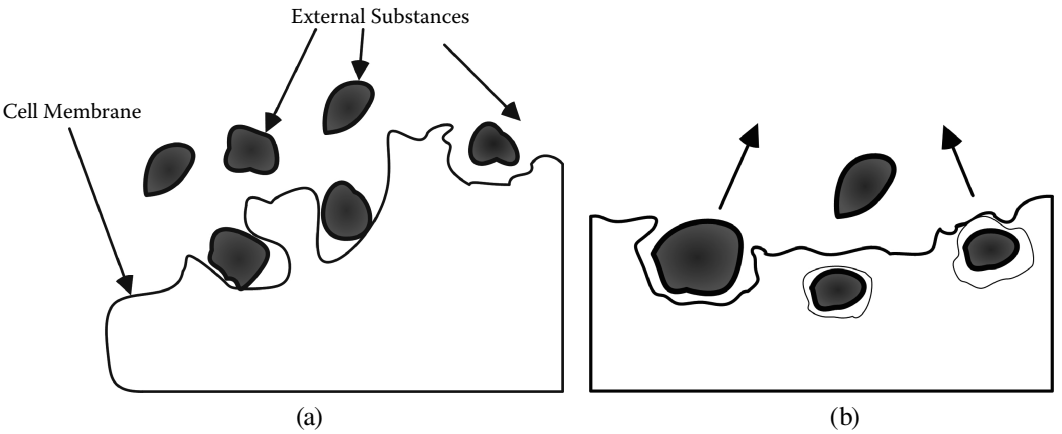


FIGURE 8.1 (a) Endocytosis; (b) exocytosis.

The left-hand side of the equation defines the binding scenario where *synapsin* and *vesicles* perform a bind (at a rate specified by *onrate*); the right-hand side of the equation illustrates the bind–break scenario (where the *synapsin–vesicle* binding breaks apart at an *offrate*—which is always smaller than the *onrate*—releasing *synapsin* and *vesicles*). S*ynapsin* and *vesicles* can then continue binding and breaking. This equation shows an ongoing process of binding and breaking that depends on the *offrate/onrate*. The larger the *offrate* is, the more bindings are broken apart. Similarly, the larger the *onrate* is, the more V–S binds are produced.

Three different scenarios are considered: (1) V is stationary (with a fixed position on the cell space) and S is mobile; (2) V is mobile and S is stationary; and (3) V and S are both mobile (leading to the maximum number of total movements/bindings). Binding patterns are in such a way that each S can bind to more than one V, and V can bind to more than one S. An example of such binding would be

```
    S-V-S-V
    |   |
V-S-V-S-V-S-V
```

Each cell space in our Cell-DEVS model is used to represent one S or V. The neighboring pattern of V and S is such that they can be adjacent cells or diagonal cells, as shown in Figure 8.2 (gray cell = S, black cell = V).

The coupled Cell-DEVS model for this application can be formally described as

$$\text{Neuron} = \langle I, X, Y, Xlist, Ylist, \eta, N, \{m,n\}, C, B, Z, \text{select} \rangle \tag{8.1}$$

where

$Xlist = Ylist = \Phi$; $\eta = 9$; $I = \langle P^X, P^Y \rangle$, with $P^X = P^Y = \{\Phi\}$;
$N = \{ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1) \}$;
$X = \{0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44\}$;

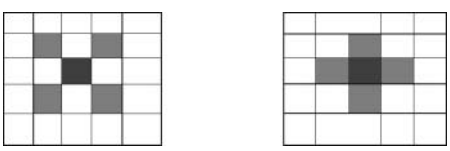


FIGURE 8.2 Neighborhood definition: (a) diagonal neighbors; (b) adjacent neighbors.

Y = {0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};
m = 26; n = 22; B = {Φ}; C = {$Cij/i \in$ [1,26], $j \in$ [1,22]};
select = { (–1,-1), (–1,0), (–1,1), (0, –1), (0,0), (0,1), (1, –1), (1,0), (1,1) }; and
Z is defined as in Cell-DEVS specifications.

The Cell-DEVS atomic model can be defined as

$$\text{Synapsin} = < X, Y, S, N, \text{delay}, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D > \tag{8.2}$$

where
X = Y = {0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};
S = {0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44, 91,92,93,94};
delay = transport;
d and τ are defined by the rules presented in Figure 8.3; and
δ_{int}, δ_{ext}, λ, and D are defined as in the formal specification of Cell-DEVS.

Here, we considered an initial value of 1 to represent V and a value of 2 to represent S. The number 0 represents an empty cell that a mobile S can occupy. To give direction to V or S (although our example assumes fixed V), a two-digit number was used. For example:

11	up-moving V	21	up-moving S
12	right-moving V	22	right-moving S
13	down-moving V	23	down-moving S
14	left-moving V	24	left-moving S

Initially, S and V can move. Once bindings occur, cells change their values; 11–14 are replaced with 31–34, and 21–24 are replaced with 41–44. Also for synapsins, four intermediate values, 91–94, are used to represent a moving cell that has not yet settled down. Once it settles down, its value changes back to 21–24 (depending on its direction of movement) and gets ready to bind to a vesicle in its neighborhood.

In Figure 8.3, we show an extract of the model definition in CD++. The model uses 100 V and 100 S molecules in a 26 × 22 cell space. Mobile S or V changes position to up, down, left, and right at random. The first group of rules assigns a direction to each V and S at random. Once V and S are adjacent or diagonal, they bind at an *onrate* = 0.10 (*random* > 0.1 represents the *onrate*).

Then we define rules for molecule movement. First, we check whether there is a moving synapsin (values 21, 22, 23, or 24) and a vesicle in its neighborhood; then the synapsin will move toward this vesicle and a binding will occur soon. The value of the synapsin is changed to 31, 32, 33, or 34 to represent a synapsin that is bound to a vesicle.

Similarly, the following rule checks whether there is a moving vesicle (value 11, 12, 13, or 14) and a synapsin in its neighborhood that could be an adjacent cell or a diagonal cell. Because the synapsin will come toward this vesicle and binding will occur soon, the value gets changed to 41, 42, 43, or 44.

The next rules represent the movement of synapsin (each movement is performed in three steps):

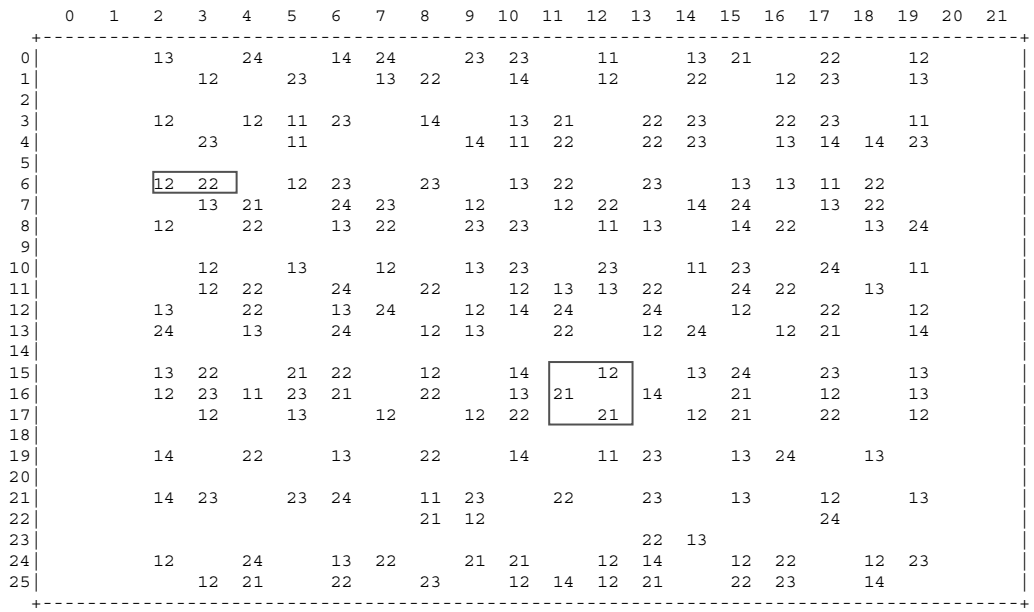
- **Step 1:** Check to see if there is an empty cell so that the synapsin can move into it. For example, if the synapsin direction is upward (value = 21), then at first we need to check whether an empty cell is right above it (91 is used as an intermediate value to occupy the empty cell).
- **Step 2:** Once an empty cell is found, it is occupied by the synapsin (i.e., the cell's value changes from 0 to a random number 21–24).
- **Step 3:** The previous position of the synapsin that just moved to an empty cell is cleared by setting the value of the cell to 0.

```
[chemCell]
type : cell            dim : (26,22)          delay : transport      border : wrapped
neighbors :  (-1,-1) (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
localtransition : chemCell-rules

[chemCell-rules]
rule : {round(uniform(11,14))} 100 { (0,0) = 1  }
rule : {round(uniform(21,24))} 100 { (0,0) = 2  }

%movement of Synapsin
rule : {round(uniform(31,34))} 100 {((0,0)=21 or (0,0)=22 or (0,0)=23 or (0,0)=24) and
( ((-1,0)- 10 = 1 or (-1,0)- 10 = 2 or (-1,0)- 10 = 3 or (-1,0)- 10 =4 ) or
  ((1,0)- 10 = 1 or (1,0)- 10 = 2 or (1,0)- 10 = 3 or (1,0)- 10 = 4)     or
  ((0,-1)- 10 = 1 or (0,-1)- 10 = 2 or (0,-1)- 10 = 3 or (0,-1)- 10 = 4) or
  ((0,1)- 10 = 1 or (0,1)- 10 = 2 or (0,1)- 10 = 3 or (0,1)- 10 = 4 )    or
  ((-1,1)- 10 = 1 or (-1,1)- 10 = 2 or (-1,1)- 10 = 3 or (-1,1)- 10 = 4) or
  ((1,-1)- 10 = 1 or (1,-1)- 10 = 2 or (1,-1)- 10 = 3 or (1,-1)- 10 = 4) or
  ((1,1)- 10 = 1 or (1,1)- 10 = 2 or (1,1)- 10 = 3 or (1,1)- 10 = 4)     or
  ((-1,-1)-10=1 or (-1,-1)-10=2 or (-1,-1)-10=3 or (-1,-1)-10 = 4) )  and random > 0.10}

%movement of Vesicles
rule : {round(uniform(41,44))} 100 {((0,0)=11 or (0,0)=12 or (0,0)=13 or (0,0)=14) and
( ((-1,0)- 30 = 1 or (-1,0)- 30 = 2 or (-1,0)- 30 = 3 or (-1,0)- 30 = 4) or
((1,0)- 30 = 1 or (1,0)- 30 = 2 or (1,0)- 30 = 3 or (1,0)- 30 = 4) or
((0,-1)- 30 = 1 or (0,-1)- 30 = 2 or (0,-1)- 30 = 3 or (0,-1)- 30 = 4)  or
((0,1)- 30 = 1 or (0,1)- 30 = 2 or (0,1)- 30 = 3 or (0,1)- 30 = 4 ) or
((-1,1)- 30 = 1 or (-1,1)- 30 = 2 or (-1,1)- 30 = 3 or (-1,1)- 30 = 4) or
((1,-1)- 30 = 1 or (1,-1)- 30 = 2 or (1,-1)- 30 = 3 or (1,-1)- 30 = 4) or
((1,1)- 30 = 1 or (1,1)- 30 = 2 or (1,1)- 30 = 3 or (1,1)- 30 = 4) or
((-1,-1)-30 = 1 or (-1,-1)-30 = 2 or (-1,-1)-30 = 3 or (-1,-1)-30 = 4)) and random >
0.10}

%moving up
rule : 91 100 {(0,0)=21 and (-1,0)=0 }
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (1,0)=91 }
rule : 0 0 {(0,0)=91}

%moving right
rule : 92 100 {(0,0)=22 and (0,1)=0}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (0,-1)=92}
rule : 0 0 {(0,0)=92}

%moving down
rule : 93 100 {(0,0)=23 and (1,0)=0 }
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (-1,0)=93 }
rule : 0 0 {(0,0)=93}

%moving left
rule : 94 100 {(0,0)=24 and (0,-1)=0 }
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (0,1)=94 }
rule : 0 0 {(0,0)=94}

%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or (0,0)=31 or (0,0)=34) and
random < 0.10}
```

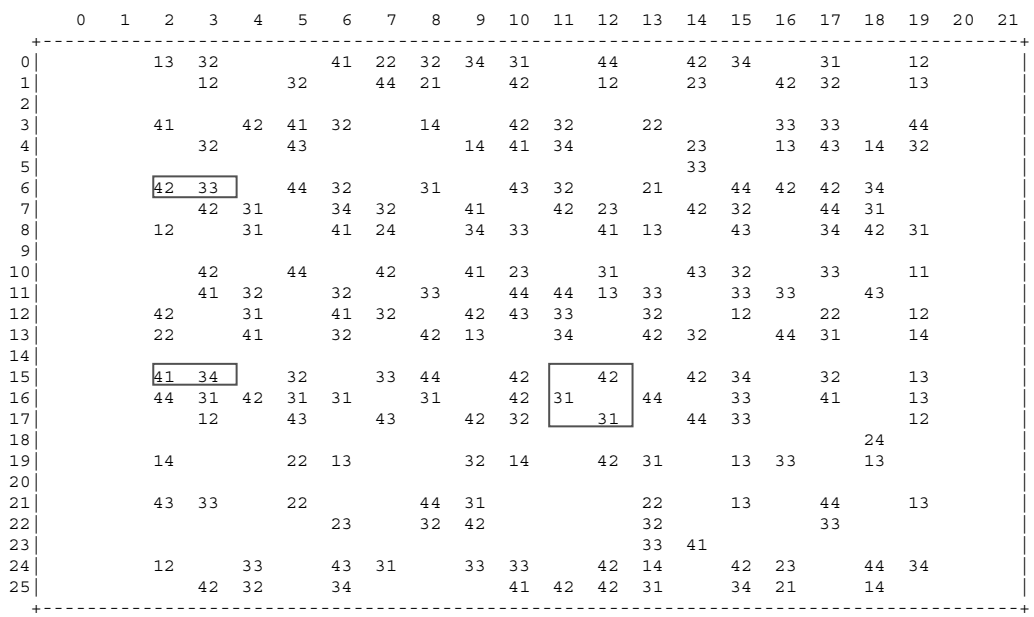
FIGURE 8.3 Excerpt of the synapsin model definition in CD++.

The last rule is used to break the S–V bindings using an *offrate* = 0.10. According to this rule, 10% of the bindings get broken and, as a result, synapsins are released and will be given another direction. They will move around until they find a vesicle and bind to it. Once the binding has occurred, depending on the *offrate,* V and S can break their binding and S can move around and find another V to bind to it. This is defined by releasing the V, assigning it a new direction, and letting it move away if there is an empty cell around it. Because the *offrate* is too small compared to *onrate,* choosing a different *offrate* results in having the same S–V bound patterns.

The simulation results can be seen in Figure 8.4. In this example, the *onrate* was set to 0.9 and the *offrate* to 0.1. Therefore, more bindings occur compared to breaking the bind. The *offrate* can



(a)



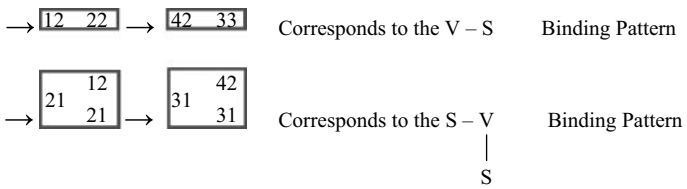
(b)

FIGURE 8.4 V and S (a) before binding at time: 00:00:00:100; (b) after binding at time: 00:00:00:300 (represent examples of binding structures).

be modified, so the larger it is, the more mobile S will be observed. However, the *onrate* is kept constant, which therefore results in having the same number of bindings at the end of any execution.

The bold boxes illustrated in Figure 8.4 show bindings between synapsin (31–34) and vesicle (41–44). Figure 8.4(a) represents the initial scenario where synapsins (21–24) and vesicles (11–14) are free and have not yet performed bindings. Once synapsins move toward vesicles, the values of the corresponding cells change to 31–34 (bound synapsins) and 41–44 (bound vesicles). Vesicles

can be surrounded by more than one synapsin, but each synapsin can bind to only one vesicle at any time. From the preceding figure, we can see the following possible binding scenarios:



Exercise 8.1

Use the *ToVal* command to generate different initial values for this model (the source code of *ToVal* can be adapted to this application in particular) and run different simulation scenarios.

Exercise 8.2

Write a program (or script) to count the number of S/V particles at the end of the simulation.

Exercise 8.3

Extend the synapsin model to include the movement of both synapsin (S) and vesicles (V) as well as defining different off and on rates. Aside from V–S reactions, the model can also include *actins*, which bind to *synapsins*. Actins can be represented as a string of cells fixed at their cell space position.

The final execution results in Figure 8.5 present a stable image of synapsin–vesicle bindings where single, double, and multiple bindings occurred within the neuron.

8.3 A MODEL OF THE HUMAN LIVER

The liver is considered the largest gland in the body, and it is responsible for many functions, including regulating blood amino acids, sugar, and lipids; forming bile (for the digestion of lipids); storing blood; removing hormones, toxins, and hemoglobin molecules; producing heat; forming cholesterol; making heparin (a substance that prevents the blood from clotting); storing vitamins; and forming plasma proteins.

In terms of its basic anatomy, the liver consists of two wedge-shaped lobes. Two blood vessels enter the liver: the portal vein and the hepatic artery. The portal vein carries dissolved food

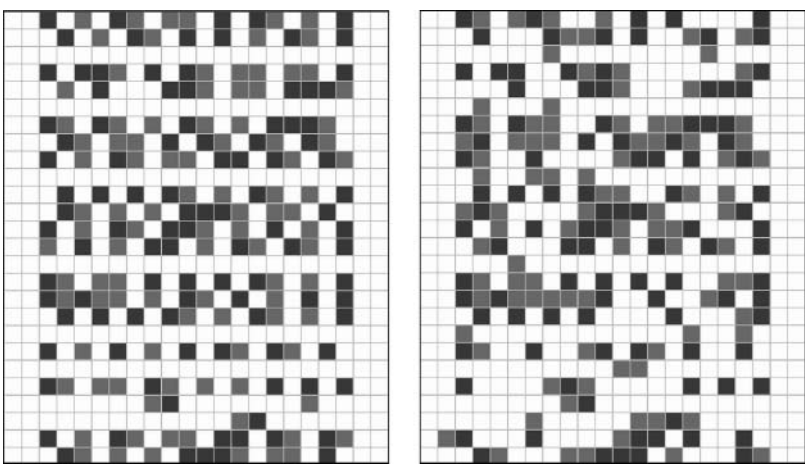


FIGURE 8.5 Model execution results: initial values; final execution.

substances from the small intestine, and the hepatic artery carries oxygenated blood from the lungs. Each lobe is further divided into many small lobules; each is about the size of a pinhead and consists of many liver cells, with bile and blood channels between them.

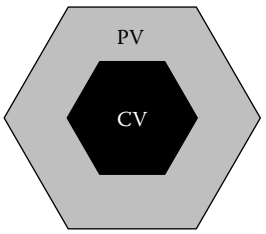


FIGURE 8.6 Structure of the lobule.

Lobules make up the main functional and structural component of the liver, which comprises thousands of them. The lobule can be seen as a tube in which blood flows from the outside to the inside. The outside of the lobule is surrounded by the portal vein (PV), which brings blood into the liver. The inner vein is the central vein (CV), and it carries blood out of the lobule and eventually out of the liver (as shown in Figure 8.6). When blood flows through the lobule, it undergoes several chemical reactions in multiple stages. These transformations occur when substances travel through various zones inside the lobule. Because lobules are the building blocks of the liver, it is important to simulate them and to build a realistic structure of the human liver by connecting thousands of them.

Hunt and colleagues [4] show how to model the inner workings of a lobule in a simulation model. They model the lobule as a hexagonal cylinder, using three stages (*zones*), each containing multiple interconnected *nodes* placed inside the lobule. The number of nodes in each zone is proportional to the approximated lobule volume of that particular zone. Each node is responsible for receiving a substance and transforming it, and each node works interdependently with the others [4]. This organization is introduced in Figure 8.7.

Based on these assumptions, we built a DEVS model that represents the chemical composition of blood entering the liver lobule [3]. A substance enters the PV, and it is then fed to all the nodes in zone I. After the nodes of zone I finish transforming the substance, their output is fed to the nodes of zone II and then zone III. After this, the output is supplied to the CV. Each node has its own set of parameters to determine the output when given a certain input. Each node is given a delay to represent the time it takes for a substance's reaction to reach completion. In Figure 8.8, we present code excerpts showing the definition of the model for NodeF in CD++ (whose full version is found in *./Liver.zip*).

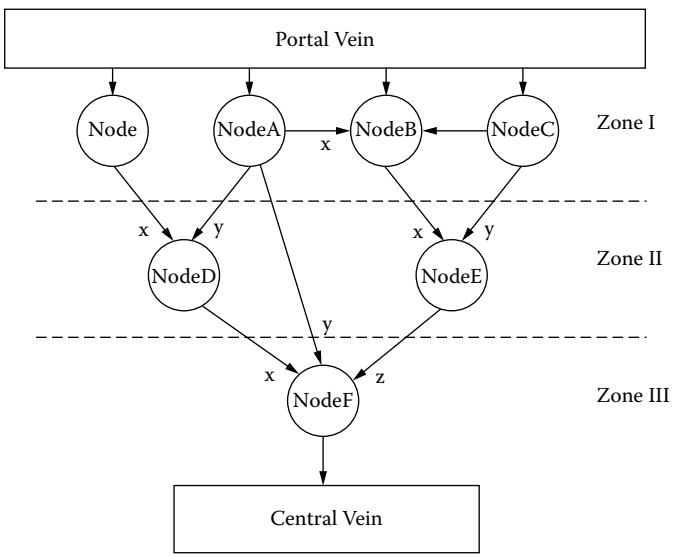


FIGURE 8.7 Zones and nodes. (Adapted from Hunt, C. A. et al. 2005. *Proceedings of Computational Methods in Systems Biology 2004; Lecture Notes in Bioinformatics 3082,* 35–43.)

```
Private:
        const Port &WinFX, &WinFY, &WinFZ;          // output ports
        Port &WoutF;                                // input ports
        Time reactionTime, time_zero;
        int Value, ValueX, ValueY, ValueZ;

Model &livernodeF::externalFunction (const ExternalMessage &msg) {
        if (msg.port == WinFX) ValueX = msg.value();
        if (msg.port == WinFY) ValueY = msg.value();
        if (msg.port == WinFZ) ValueZ = msg.value();

        if (valueX != 0 && ValueY != 0 && ValueZ != 0)
                holdIn(active, reactionTime);
return *this;
}

Model &livernodeF::outputFunction (const InternalMessage &msg) {
        if (ValueX==1 & ValueY ==1)
                switch (ValueZ) {
                        case 1: Value = 7; break;
                        case 2: Value = 5; break;
                        case 3: Value = 3; break;
                        case 4: Value = 1; break;
                        case 5: Value = 2;
                }
        sendOutput(msg.time(), WoutF, Value);
return *this;
}

Model &livernodeF::internalFunction (const InternalMessage &msg) {
        passivate();
}
```

FIGURE 8.8 CD++ definition of a node.

As we can see in Figure 8.8, we initially define the model's I/O ports and state variables. Then we define the external transition function, which, in this case, takes a message received on the port *WinFX* and assigns it to the variable *ValueX*. When the *livernode F* (which has three input ports) receives values from each of the ports, it can compute the reaction, which takes *reactionTime* units. After this time has elapsed, the output function is called. This function, based on the values of *ValueX, ValueY,* and *ValueZ,* will output a value representing the chemical reaction on the variable *Value*.

Exercise 8.4

Using the detailed information provided in Hunt et al. [4] (and any other related references needed), define an extension to the model in Figure 8.8 using the chemical reactions occurring at each of the nodes.

All the nodes were put together in a coupled model to form an entire lobule, as shown in Figure 8.9. We first define all the nodes included in the lobule. Then we define the model's coupling (e.g., the output port of node *A* to the *Y* input port of node *D,* etc.). The *in* port represents the portal vein and the *out* port represents the central vein. The coupled model here is based on the model presented in Figure 8.6 [4].

Exercise 8.5

Use the models created in Exercise 8.4 and create a lobule coupled model based on the new atomic component.

```
components: noder@livernode nodeA@liverNodeA nodeB@liverNodeB nodeF@liverNodeF
in : in
out : out
Link : in win@node                          Link : in winA@nodeA
Link : in winBY@nodeB                       Link : in winC@nodeC
Link : wout@node winDX@nodeD                Link : woutA@nodeA windDY@nodeD
Link : woutA@nodeA windBX@nodeB             Link : woutC@nodeC windBZ@nodeB
Link : woutC@nodeC windEY@nodeE             Link : woutB@nodeB windEX@nodeE
Link : woutA@nodeA windFY@nodeF             Link : woutE@nodeE windFZ@nodeF
Link : woutD@nodeD windFX@nodeF
```

FIGURE 8.9 Structure of the lobule coupled model.

One of the functions of the liver is to keep a steady concentration of glucose in the blood. This is done through three types of reactions: glyconeogenesis, glycogen synthesis, and degradation. Most substance reactions in the liver need energy and the sources for this energy are ATP and ADP. In most cases, ATP is broken down into ADP and energy is released. Oxaloacetate is used in the mitochondria, and it cannot cross the mitochondrial membrane until it is converted to malate. Once malate passes through the membrane, it can then be converted back to oxaloacetate. Oxaloacetate is produced by pyruvate carboxylase and is then converted to malate. These reactions were tested, and we show the results in Figure 8.10.

We also considered the formation of UDP-glucose, which can be attached to glucose chains that can be acted upon by glycogen synthesis. Glucose enters the cells by facilitated diffusion, and then the cell modifies glucose by phosphorylation, as shown in Figure 8.11. Glucose-6-phosphate is used in the synthesis of glycogen: glucose-6-phosphate is first isomerized to glucose-1-phosphate by the enzyme phosphoglucomutase, as seen in Figure 8.12. UDP-glucose has the ability to attach its glucose part to glucose chains. This new chain can be acted upon during glycogen synthesis, as seen in Figure 8.13. Figure 8.14 shows this process in our CD++ simulation.

The model file includes other simulation examples, including glycogen degradation and glyconeogenesis (the synthesis of glucose from other organic compounds, which is catalyzed by pyruvate kinase), glycolysis, etc.

	Starting simulation. Stop at time: 00:05:00:000	(1) LiverNode received (NADH + H^+)
	00:00:10:000/in/1,00000	(2) LiverNodeA received Pyruvate
1	LiverNode Received: 1 at time 00:00:10:000	(3) LiverNodeB received CO_2 on node Y
2	LiverNodeA Received: 1 at time 00:00:10:000	(4) LiverNodeC received ATP
3	LiverNodeB Received on port Y: 1 at time 00:00:10:000	(5) LiverNode produced (NADH + H^+)
4	LiverNodeC Received: 1 at time 00:00:10:000	(6) LiverNodeD received (NADH + H^+) on port X
5	LiverNode Produced: 1 at time 00:00:13:000	(7) LiverNodeA produced Pyruvate
6	LiverNodeD Received on port X: 1 at time 00:00:13:000	(8) LiverNodeB received Pyruvate on port X
7	LiverNodeA Produced: 12 at time 00:00:13:000	(9) LiverNodeD received substance on port Y, ignore
8	LiverNodeB Received on port X: 12 at time 00:00:13:000	(10) LiverNodeF received substance on port Y, ignore
9	LiverNodeD Received on port Y: 12 at time 00:00:13:000	(11) LiverNodeC produced ATP
10	LiverNodeF Received on port Y: 12 at time 00:00:13:000	(12) LiverNodeB received ATP on port Z
11	LiverNodeC Produced: 6 at time 00:00:13:000	(13) LiverNodeE received substance on port Y, ignore
12	LiverNodeB Received on port Z: 6 at time 00:00:13:000	(14) LiverNodeB produced Oxaloacetate
13	LiverNodeE Received on port Y: 6 at time 00:00:13:000	(15) LiverNodeE received Oxaloacetate on port X
14	LiverNodeB Produced: 7 at time 00:00:16:000	(16) LiverNodeD produced (NADH + H^+)
15	LiverNodeE Received on port X: 7 at time 00:00:16:000	(17) LiverNodeF received (NADH + H^+) on port X
16	LiverNodeD Produced: 1 at time 00:00:16:000	(18) LiverNodeE produced Oxaloacetate
17	LiverNodeF Received on port X: 1 at time 00:00:16:000	(19) LiverNodeF received Oxaloacetate on port Z
18	LiverNodeE Produced: 7 at time 00:00:19:000	(20) LiverNodeF produced Glucose-1-P
19	LiverNodeF Received on port Z: 7 at time 00:00:19:000	
20	LiverNodeF Produced: 5 at time 00:00:22:000	
	Simulation ended!	

FIGURE 8.10 Forming glucose-1-P.

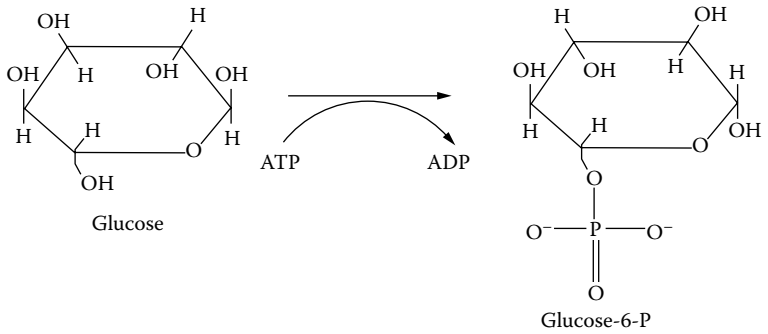


FIGURE 8.11 Diagram for phosphorylation.

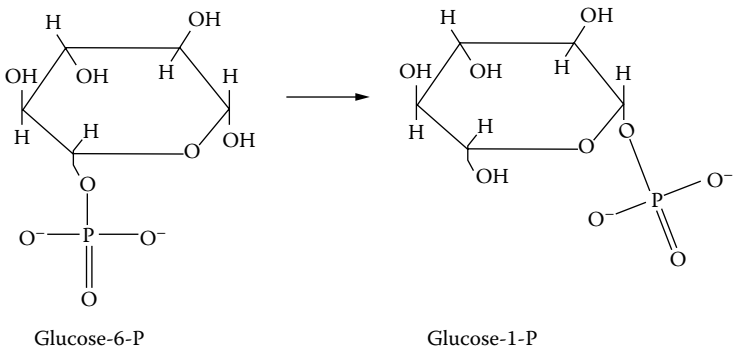


FIGURE 8.12 Isomerization.

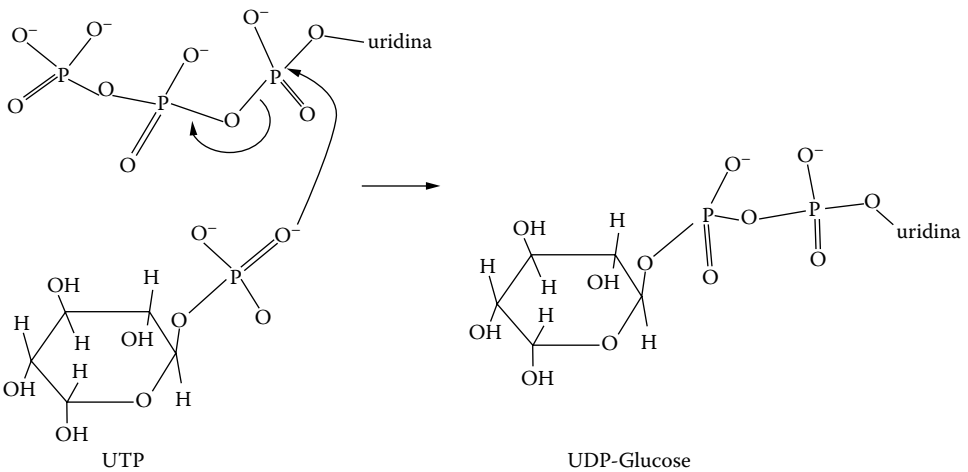


FIGURE 8.13 Forming of UDP-glucose.

Exercise 8.6

Based on the liver model presented in this section, construct a Cell-DEVS definition consisting of 200 × 30 cells. Each cell should follow the definition found in this section, and they should be executed together as a cell space.

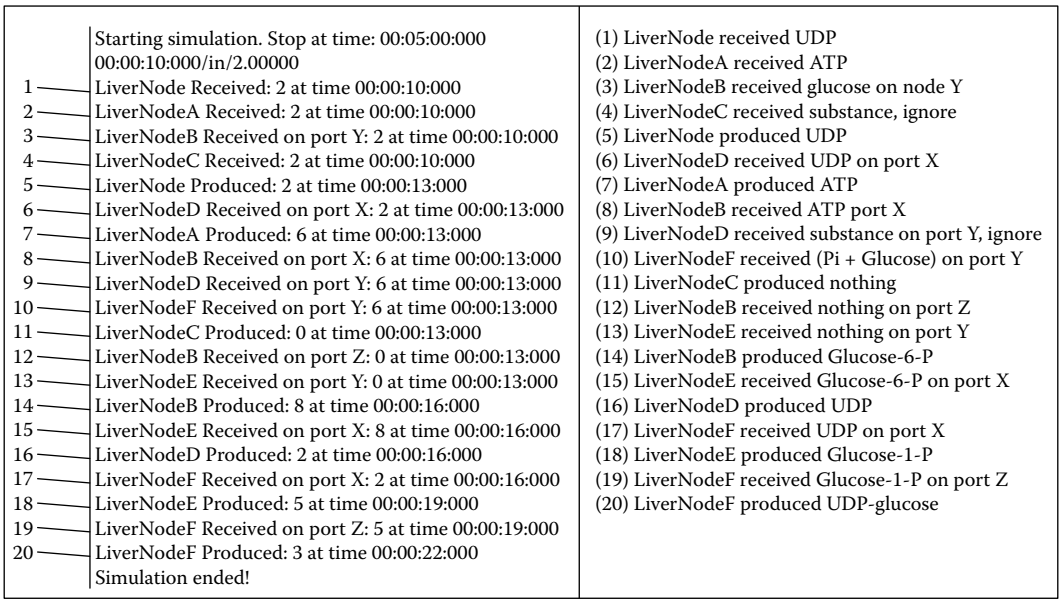


FIGURE 8.14 Forming of UDP-glucose: CD++ simulation.

8.4 SPREADING OF MARINE BACTERIA

Vibrio parahaemolyticus is a marine bacterium living in sediment and plankton found along coasts and in estuaries. In order to survive, these bacteria need a minimal percent of salt and a pH between 7.5 and 8.5. They grow in an environment with temperatures ranging from 15 to 43°C (37°C is the optimal value) and reproduce at 15°C in the scales or the intestines of fish. They need between 20 and 30 min to reproduce; however, they cannot reproduce at temperatures below 8°C. Bacteria are destroyed when exposed to temperatures higher than 60°C for a period of 10 min or to high-acid (pH) environments.

The model presented in Ameghino, Glinsky, and Wainer [5] and found in *./Bacteria.zip* focuses on the bacteria concentration while the temperature varies (the rest of the variables that may affect the experiment are assumed to be appropriate for the normal growth of the bacteria). The evolution of the bacteria over the surface of a fish is modeled using a Cell-DEVS component. We couple a DEVS generator to introduce temperature changes between –10 and 0°C, representing a source of cold (e.g., from a refrigerator). We use a three-dimensional model with two surfaces, the first representing the concentration of bacteria and the second showing the variation of temperature. The temperature in a cell is calculated as the average of its neighbors, and the diffusion time is 1 s. The second plane governs the reproduction of bacteria using the following rules:

1. If the cell temperature is below 8°C for a period of 10 s, the bacterium does not grow.
2. If the cell temperature is between 8 and 60°C for a period of 30 s, then the bacterium grows.
3. If the cell temperature is above 60°C for a period of 10 s, then the bacterium dies.

We use inertial delays and define that a cell reaching the concentration of 100 germs begins infecting the neighboring cells. Figure 8.15 shows the specification of such a model using CD++. We first declare the top model's components, *Coldgenerator* and *contamination,* and their coupling scheme. Then we define external arguments for *Coldgenerator,* a DEVS model that generates cold temperatures using an exponential distribution function with the specified parameters. The Cell-DEVS model *contamination* includes the connections with *Coldgenerator* and the rules for the

```
[top]
components : contamination Coldgenerator@Generator
link : out@Coldgenerator  inputCold@contamination

[Coldgenerator]
distribution : exponential         mean : 3       initial : 1   increment : 0

[contamination]
type : cell  dim : (10, 10, 2)
delay : inertial    border : nowrapped
neighbors : (-1,-1,0) (-1,0,0) (-1,1,0) (0,-1,0)  (0,0,0) (0,1,0) (1,-1,0) (1,0,0) (1,1,0)
(0,1,1) (-1,-1,1) (-1,0,1) (-1,1,1) (0,-1,1) (0,0,1) (1,-1,1) (1,0,1) (1,1,1)

link : inputCold  in@contamination(0,0,1)
localtransition  : Evolution
portInTransition : in@contamination(0,0,1) setCold
zone : Temperatures { (0,0,1)..(9,9,1) }

[Temperatures]
rule: { ( if((-1,-1,0)!= ?,(-1,-1,0),0) + if((-1,0,0)!=?,(-1,0,0),0) + if((-1,1,0)!= ?,
        (-1,1,0),0) + … }              1000     { t }

[Evolution]
rule: 0  10000 { cellpos(2) = 0 and (0,0,1) > 60 }
rule: {round(if((0,0,0)*2 > 99,0.7,1)*(0,0,0)*2) + if((-1,-1,1)!=? and … } 30000 {
        cellpos(2)=0 and (0,0,1) > 8 and statecount(?) = 5 * 2 }
…
rule : {(0,0,0)} 10000    { cellpos(2) = 0 }

[setCold]
rule :     { uniform(-10,0)}  500      { t }
```

FIGURE 8.15 Specification of the bacteria mode.

Cell-DEVS component. For each rule, the value, the delay, and the condition are specified. The *temperature* section represents the local computing function for the temperature plane. A single rule defines the temperature as the average of the temperature of the neighboring cells. The *evolution* rules describe the bacterium's behavior on the second plane. The *setCold* section states the range of temperatures generated by the DEVS component.

Figure 8.16 illustrates the results obtained when this model is executed, showing the evolution of the bacteria over the surface of fish after 4 h. The left side represents the bacteria concentration (in grayscale). The white areas represent regions where bacteria are not present as a result of the extremely low temperature; darker shades represent higher concentrations of bacteria. The right side represents the different temperatures of the surface, where darker shades represent colder temperatures.

Exercise 8.7

Modify the rate to generate different temperatures and the range, and execute different simulations. Then generate different experiments showing what would happen if the resilience of bacteria to temperature changed and analyze the results.

8.5 VIRUS SPREADING IN A POPULATION

The following model shows a Cell-DEVS representation of the competition between population and viruses. The model (presented in Shang and Wainer [6] and found in *./virusSurvival.zip*) is based on the work presented in Dzwinel [7]. It shows the evolution of a colony consisting of individuals on a two-dimensional lattice. The model represents three phases of life: youth, maturity (during which individuals procreate), and old age, whose duration can be variable. The environment is affected by periodic plagues that attack the colony.

The model includes rules for evolution of the population and the interaction between individuals and virus, using a 20 × 20 mesh. Each individual residing in a node is described by a number from

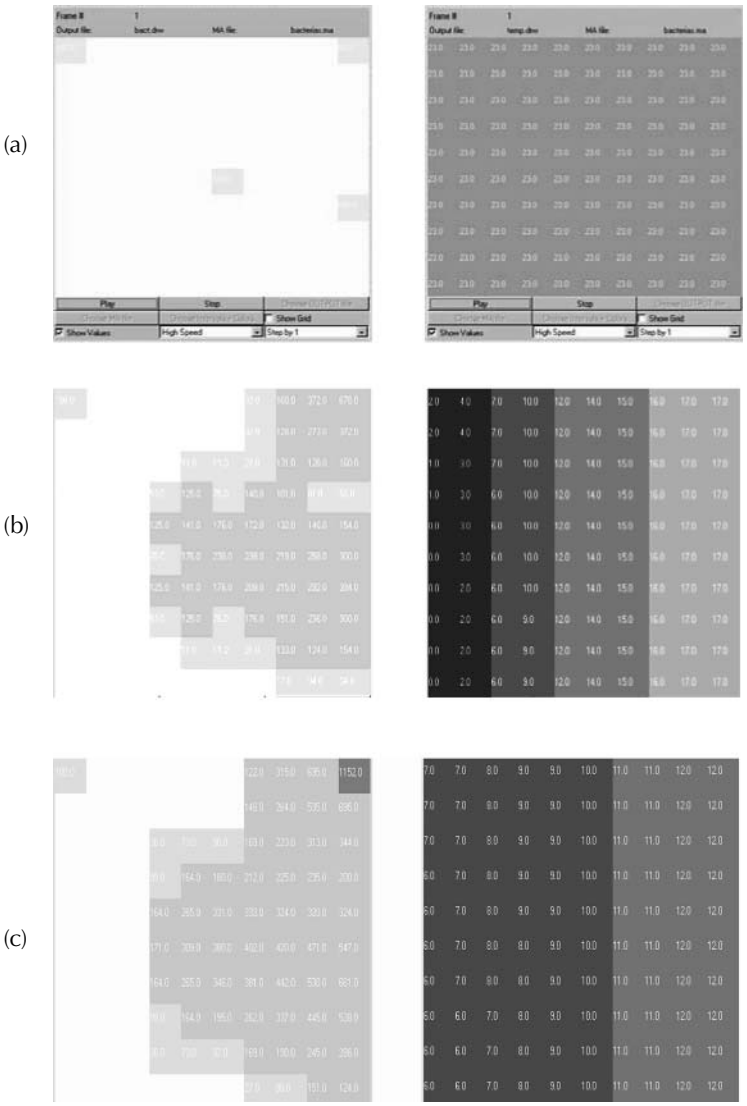


FIGURE 8.16 Results of bacteria propagation: (a) initial concentration; (b) after 1.5 h; (c) after 4 h.

0 to 9, representing a virus (values 8 and 9) or individuals (1 represents youth, 2–5 represent adults, and 6 represents the elderly).

The rules focus on the competition between the population and the plague. Both categories follow their own active rules, sharing the same lattice. Viruses can attack the population, and the plague is initially scattered. Viruses in the active state reproduce and kill individuals; inactive viruses die after a delay.

The rules in Figure 8.17 show the model definition in the parallel version of CD++ (the model in the repository also includes a simpler version of the model, which runs on the stand-alone version). As we can see in the figure, each cell is using two I/O ports. One of them (*pa*) represents the age of individuals and virus. The other (*pd*) represents the directions of moving individuals. These ports are used to define movement, reproduction, and interaction between individuals and a virus. The age port (*pa*) is the dominant port according to which each cell is distinguished.

```
[survival_rules]
rule : {~pa:=(0,0)~pa+1; ~pd:=round(uniform (1,4));} 100  { (0,0)~pa =1 and (not
     ((0,-1)~pa=8 and (0,1)~pa=8)) and not ((-1,0)~pa = 8 and (1,0)~pa = 8)}

%Age increment for newborns not killed by viruses. rule : {~pa:=0;} 100 {(0,0)~pa=1 and
     (((0,-1)~pa=8 and (0,1)~pa=8) or ((-1,0)~pa=8 and (1,0)~pa=8))}
     ; virus killing individuals

%Moving rules for mature cells.
rule : {~pa:=0;} 100 {(0,0)~pa>=2 and (0,0)~pa<=5 and (0,0)~pd=4 and (0,-1)~pa = 0}
   ...

rule:  {~pa:=0;} 100 {(0,0)~pa=6} % Dying individual
rule: {~pa:=0;} 100 {(0,0)~pa=8 and (((0,-1)~pa>=1 and (0,-1)~pa<=6) or ((0,1)~pa>=1 and
     (0,1)~pa<=6) or ((-1,0)~pa>=1 and (-1,0)~pa<=6) or ((1,0)~pa>=1 and (1,0)~pa<=6))}

%Individual kill virus rule
rule:  {~pa:=9;} 100 {(0,0)~pa=8 and not ( ((0,-1)~pa>=1 and (0,-1)~pa<=6) or
     ((0,1)~pa>=1 and (0,1)~pa<=6) or ((-1,0)~pa>=1 and (-1,0)~pa<=6) or
     ((1,0)~pa>=1 and (1,0)~pa<=6))}
...
rule: {~pa:=0;} 100 {(0,0)~pa=9}

rule: {~pa:=(0,1)~pa+1; ~pd:=round(uniform(1,4));} 100 {(0,0)~pa=0 and ((0,1)~pa>=2 and
     (0,1)~pa<=5) and (0,1)~pd=4} % Virus: active to passive

%Individual reproduction rule
rule: {~pa:=1;} 100 {(0,0)~pa=0 and ( ((0,1)~pd !=4 or (0,1)~pa<2 or (0,1)~pa>5) and
     ((-1,0)~pd !=3 or (-1,0)~pa<2 or (-1,0)~pa>5) and ((0,-1)~pd !=2 or (0,-1)~pa<2 or
     (0,-1)~pa>5) and ((1,0)~pd !=1 or (1,0)~pa<2 or (1,0)~pa>5) ) and ((((0,-1)~pa>=2 and
     (0,-1)~pa<=5) and ((0,1)~pa>=2 and (0,1)~pa<=5) ) or ( ((-1,0)~pa>=2 and (-1,0)~pa<=5)
     and ((1,0)~pa>=2 and (1,0)~pa<=5)))}

%Virus reproduction rule
rule: {~pa:=8;} 100 {(0,0)~pa=0 and ( ((0,1)~pd !=4 or (0,1)~pa<2 or (0,1)~pa>5) and
     ((-1,0)~pd !=3 or (-1,0)~pa<2 or (-1,0)~pa>5) and ((0,-1)~pd !=2 or (0,-1)~pa<2 or
     (0,-1)~pa>5) and ((1,0)~pd !=1 or (1,0)~pa<2 or (1,0)~pa>5) ) and not((((0,-1)~pa>=2 and
     (0,-1)~pa<=5) and ((0,1)~pa>=2 and (0,1)~pa<=5)) or (((-1,0)~pa>=2 and (-1,0)~pa<=5)
     and ((1,0)~pa>=2 and (1,0)~pa<=5))) and (((0,-1)~pa=8 or (0,1)~pa=8) or ((-1,0)~pa=8
     or (1,0)~pa=8))}
```

FIGURE 8.17 Definition of the Cell-DEVS model.

The rules in the figure describe in detail the model's behavior:

- **Growth:** periodically, each cell will be increased by 1 to indicate that all individuals age. After reaching the maximum age (6), the cell will be reset to indicate that the individual has died.
- **Reproduction:** for each unoccupied cell with at least two adult neighbors (von Neumann's neighborhood), the cell is set to 1.
- **Virus reproduction:** when an unoccupied cell is surrounded by at least one active virus, the empty cell will be occupied by an active virus.
- **Virus state change:** active viruses become inactive after a delay. After another delay, the inactive virus will die.

Because individuals and viruses share the same living space, they compete for the living space as follows:

- **Viruses killing individuals:** if a cell occupied by individuals is surrounded by at least two viruses and their distribution is vertical or horizontal, individuals die.
- **Individuals killing viruses:** if a cell occupied by active viruses is surrounded by at least two individuals arranged vertically or horizontally and the viruses have no capacity to kill the individuals (i.e., the number of virus neighbors is less than two or they are not

arranged vertically or horizontally), the virus dies. If an individual is killed by a virus, the corresponding virus dies, too.

- **Conflict:** if the current cell is unoccupied and surrounded by two or more mature individuals, the reproduction rule determines that a new individual will be born. However, if the cell is also surrounded by at least one virus, it must be occupied by a virus (according to the virus reproduction rule). In these cases, individuals have higher priority than viruses to reproduce.

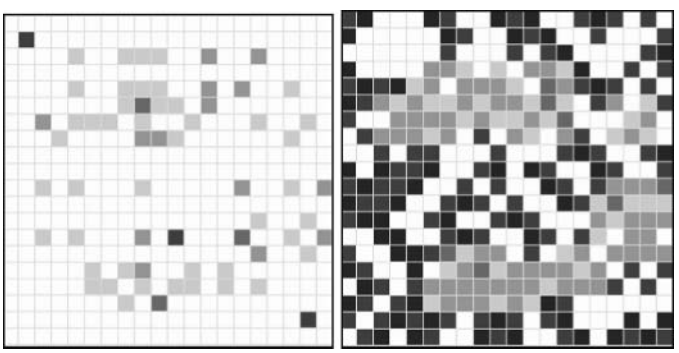
In addition to these basic rules, we modeled displacement of individuals. Movement rules are described as follows:

- Only mature individuals can move. At every step, a random direction will be produced for all mature individuals.
- Movement brings new conflicts. For each empty cell, the cell may be occupied by other mature individuals (due to movement), by a newborn, or by a new virus. The movement of individuals has the highest priority, reproduction of individuals has the next priority, and virus reproduction has the lowest priority.
- If individuals move, they might become out of the range of the virus. In this case, the movement has higher priority (thus, the current neighbor viruses cannot kill moving individuals).
- Reproduction and movement can happen simultaneously. If the given empty cell satisfies the conditions for a new birth, the cell will be occupied by the newborn, and the parents will move to other places.

Figures 8.18–8.20 discuss different execution results for various scenarios, showing the values for the *pa* port. Figure 8.18 considers a partially clustered population, with viruses scattered within the population; no increment rules are applied. The figure shows an execution of this scenario (gray cells change from light to dark to indicate different ages; darker cells present active and inactive viruses). After some time, the virus dominates the population, reflecting the fact that the individuals have stricter reproduction rules than viruses. The conflict rules give higher priority to population over viruses, so the population that aggregates survives; therefore, there is a tendency for individuals to group.

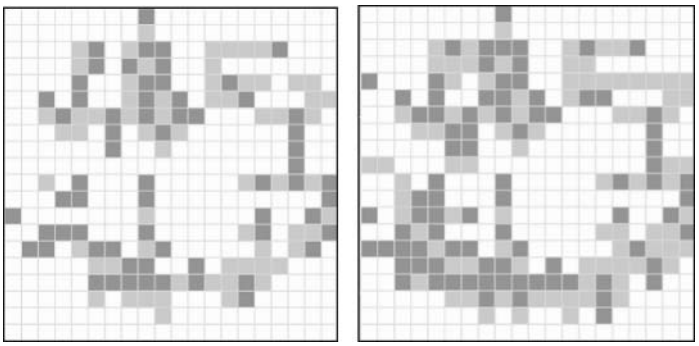
Our second scenario presents a packed population with scattered viruses. As we can see in Figure 8.19, population clustering prevents the individuals from being killed; however, it also restricts reproduction. Because viruses scatter inside the population, their reproduction is also restricted. The population size grows while viruses disappear.

The last scenario presented here shows sparse individuals and viruses, including movement rules. In Figure 8.20, because the individuals are separated, there is more space to reproduce. However,



(a) (b)

FIGURE 8.18 Virus spread scenario: (a) initial population; (b) population and viruses after 100 steps.



(a) (b)

FIGURE 8.19 Concentrated population scenario: (a) initial population; (b) population and viruses after 100 steps.

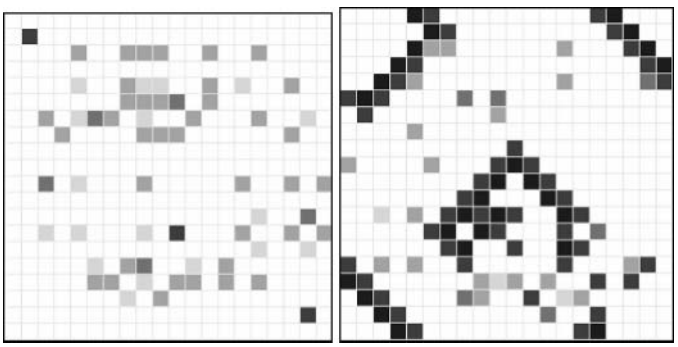


FIGURE 8.20 Movement scenario.

the number of individuals decreases due to the development of viruses, which can reproduce more easily when individuals are scattered.

The addition of movement rules reduces the number of individuals who survive when compared to the previous scenarios. This is attributed to several causes:

- Population tends to cluster, and reproduction rules lead to clustering (which is helpful for survival). However, the introduction of movement has the opposite effect because it allows for more opportunities for individuals to detach, and movement gives more chances to get close to a virus, therefore increasing the chance of infection.
- The movement rules have higher priority than reproduction, so the possibilities to reproduce are smaller.
- In the previous examples, the initial distribution contained more young individuals. Here, the age is uniformly distributed, and older individuals die earlier.

8.6 MODELING THE HEART TISSUE

This section discusses a model of the heart tissue behavior presented in Giambiasi and Wainer [8], which uses different kinds of Cell-DEVS models to build a discrete variable model of the heart tissue conduction.

The heart (Figure 8.21) is a muscle responsible for pumping blood into the circulatory system. Behavior of the phenomena occurring in the heart muscle and tissue has been extensively studied and reported in a wide variety of medical treatises (see, for instance, [9] and [10]).

Heart activity is usually analyzed according to three kinds of activities: mechanical, electrical, and cellular. In terms of mechanical behavior, blood returns to the heart through the superior and inferior vena cava and flows to the right atrium. The blood then flows to the right ventricle, where it is pumped through the pulmonary veins to the lungs to return oxygenated to the left atria through the pulmonary artery. Then it flows to the left ventricle, which returns the oxygenated blood to the body through the aorta.

This mechanical activity is triggered by the electrical impulses of the cells in the heart tissue. The heart muscle is excitable, and the cells in its tissue respond to external stimuli by contracting the muscular cells. If the stimulus is too weak, the muscle does not respond; if the voltage received is high enough, the cells contract at maximum capacity. Cells in the heart tissue are excited when adjacent cells are charged positively. In that case, an upstroke of its action potential is provoked, and it spreads to nearby cells. The electrical conduction system of the heart is responsible for the control of its regular pumping. This activity is originated in the sinoatrial (SA) node (also known as the *pacemaker*), and it spreads through the atria muscle at a speed of 1 m/s (for human beings, 80 ms are needed to activate the atria). This is an electrically active region of the heart that self-activates. All excitable tissue, once activated, exhibits a refractory period before returning to rest. During this period, the muscle does not respond to external stimuli. Before a new contraction is started, the previous one should have finished. After that, the electrical activity is spread to the atrioventricular node, where it propagates slowly (0.1 m/s); then the excitation travels at 2 m/s through the Purkinje fiber.

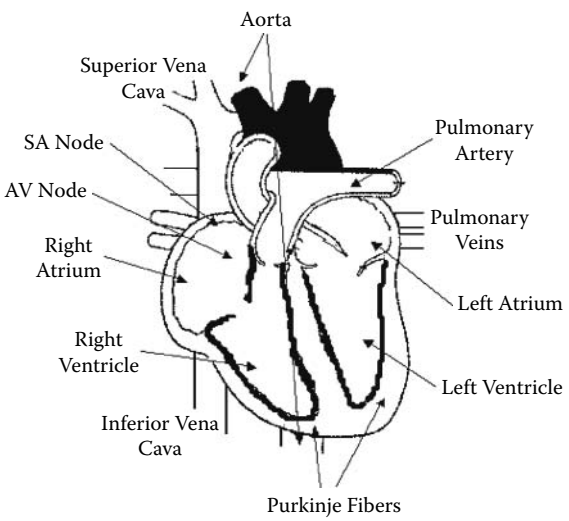


FIGURE 8.21 Basic anatomy of the heart.

This electrical activity is originated by the chemical reactions occurring at the cellular level, which consist of the interchange of ions of potassium and sodium in the walls of the cells. This chemical reaction produces potential differences of millivolts, which trigger the electrical activity. This behavior of cell membrane activity was originally characterized by [11], a foundational article that presented the detailed behavior of the intermembrane action's potential function. They recognized different phases in this function:

- The heart tissue is relaxed, and the interior of the membrane is electrically negative with relation to the surface, with a difference of potential of 50 mV.
- The surface membrane is repolarized, creating two zones with a potential difference.
- Electrical activity starts and the external surface becomes negative, with a potential difference of 30 mV. This phase is called excitation (or depolarization).
- Negative voltage in the surface trespasses on the membrane, and the original status is recovered. This phase is called repolarization.

The Hodgkin–Huxley model showed that virtually all membrane current models could be defined by writing the total membrane current, which is a sum of the individual currents carried by different ions through specific channels in the cell's membrane. The calculation is based on sodium ion flow, potassium ion flow, and the leakage ion flow. This behavior can be defined as

$$I = m^3\, h\, G_{\mathrm{Na}}\,(E - E_{\mathrm{Na}}) + n^4\, G_{\mathrm{K}}\,(E - E_{\mathrm{K}}) + G_L\,(E - E_L) \tag{8.3}$$

where

I = the total ionic current across the membrane;
m = the probability that one particle contributed to activate the sodium gate;
h = the probability that one inactivation particle has not caused the sodium gate to close;
G_{Na} = the maximum sodium conductance;
E = the total membrane potential;
E_{Na} = the sodium membrane potential;
n = the probability that one of four particles influenced the potassium gate;
G_{K} = the maximum possible potassium conductance;
E_{K} = the potassium membrane potential;
G_L = the maximum leakage conductance; and
E_L = the leakage membrane potential.

Hodgkin and Huxley computed empirical formulas for the sodium gate activation (m), sodium particle activation probability (h), and potassium gate activation probability (n). By applying the Hodgkin–Huxley equations, we can obtain the action potential function for the cells in different regions of the heart tissue, which depends on the variation in conductivity, length of the fibers, etc. For instance, Figure 8.22 shows the results obtained when using the Hodgkin–Huxley equations using parameters corresponding to cells of the atria [8].

The Hodgkin–Huxley model has been extensively used in different studies because it has been shown that it reproduces the electrical properties in the myocardium cells with fidelity. Nevertheless, the use of this model in a realistic reproduction of the heart tissue (probably consisting of millions of cells) can be computationally expensive. Consequently, different authors have tried to simplify the complexity of the equations, and various studies have attempted to solve this problem using cellular automata (CA) (see, for instance, [8], [12], and [13]). Most of these models are based on simple CA for excitable media, which discretize the Hodgkin–Huxley results. Figure 8.23 shows a complete specification of this model.

This Cell-DEVS coupled model uses 5 × 5 cells, Moore's neighborhood transport delays, and nonwrapped borders (special rules were defined for the borders). The heart-rules section represents the local computing function; the first rule represents the initiation of electrical activity in a resting

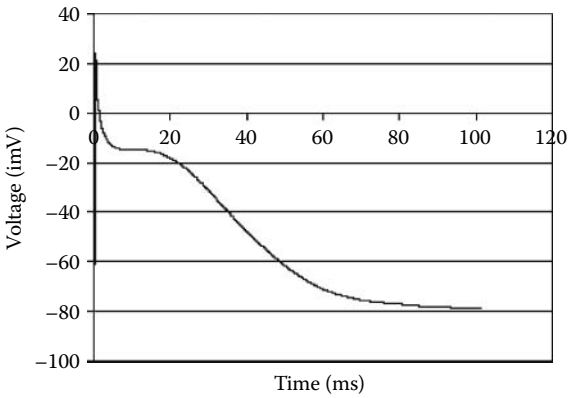


FIGURE 8.22 Action potential in the atria cells using Hodgkin–Huxley equations. (From Giambiasi, N., and G. Wainer. 2005. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:137–151.)

```
[Heart]
type : cell
dim : (5,5)
delay : transport
border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)
neighbors : (0,1) (1,-1) (1,0) (1,1) (0,0)
localtransition : Heart-rules

[Heart-rules]
rule : 2              0.48    {(0,0)=0 and statecount(2)>0 }
rule : 1              1.48    { (0,0) = 2 }
rule : 0              17.5    { (0,0) = 1 }
rule : { (0,0) }      0       { t }
```

FIGURE 8.23 Cell-DEVS definition of a simple heart tissue model.

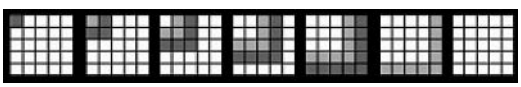


FIGURE 8.24 Heart tissue model execution.

cell (with value 0). In that case, we check to see if any of the neighbors is excited (value 2) and, in such a case, the cell becomes excited. The second and third rules define the cells changing to the recovering and resting states. Figure 8.24 shows the results obtained when this model executes. It shows the evolution of this considering an SA cell in (0,0).

This model represents three different delays at different scales, and each rule is triggered by an event that is executed asynchronously in each of the cells at randomly chosen instants. Representing this problem using these simple rules poses a problem in a model's precision. We have discretized the continuous function shown in Figure 8.22 with only three different discrete states. If a partial differential equation (PDE) is included on each cell, it will be able to react to each possible modification of the parameters adequately.

Figure 8.25 shows how to implement this model as Cell-DEVS running the Hodgkin–Huxley model in each of the cells. We implemented a model of the Action Potential (AP) function for the cells in the heart atria [8]. This Cell-DEVS model simulates the electrical behavior of the cells following the Hodgkin–Huxley model, as described in Section 8.4, discretizing time in each of the

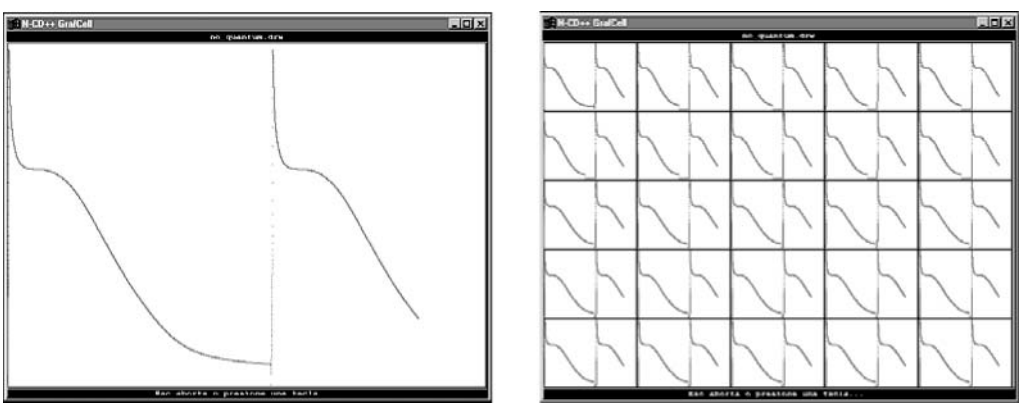
```
[heart]
type : cell
dim : (5,5,2)
delay : transport
border : nowrapped
neighbors : (-1,-1,0) (-1,0,0) (-1,1,0)
neighbors : (0,-1,0)  (0,0,0)  (0,1,0)
neighbors : (1,-1,0)  (1,0,0)  (1,1,0)  (0,0,1)
localtransition : heart-rule-AP

[heart-rule-AP]
rule : { AP(cellpos(0) } 1 { cellpos(2)=0 and (
                           (-1,0,0) > 0 or (0,-1,0) > 0 or (-1,-1,0)>0) and (0,0,0) = -83.0) }

rule : { AP(cellpos(0) } 1 { cellpos(2)=0 }

rule : { if( (0,0,0) = 1.0 or (0,0,0) = -83.0, 0.0, 1.0) } 1 { cellpos(2)=1 }
```

FIGURE 8.25 Cell-DEVS definition of the action potential function for a heart tissue model. (From Giambiasi, N., and G. Wainer. 2005. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:137–151.)

cells under execution. Figure 8.25 shows the model definition using CD++, which can be found in *./APAfun.zip.*

In this case, we use a three-dimensional model (5 × 5 × 2 cells) with transport delays. The neighborhood uses the adjacent cells in plane 0 and the cell above, which will be used to decide if the current cell should be computed. The local computing function, *heart-rule-AP,* is defined by two rules. The first one will be evaluated only by the cells in the first plane in the model (*cellpos(2)* = 0) and only if the cell is resting and a positive voltage is detected in the cell's neighborhood. This rule will trigger the update of the cell state using the Hodgkin–Huxley equation (Equation 8.3) (*AP* function). The second rule will be used in the subsequent activations. The third rule is evaluated only by the second plane (*cellpos(2)* = 1), and it is used to trigger time-based actions for the first plane. This plane changes its state in each time step, triggering the execution of the rules of the action potential function because Cell-DEVS considers only activation of a cell under asynchronous events. If no event occurs, the cells will become quiescent and the simulation ends.

The AP function in this model receives the coordinates of the current cell and its current state. Using these values, it recovers the previous state of the current cell and computes the next voltage using Equation (8.3). Figure 8.26 shows the execution results of this model. As we can see, the results obtained are the same as we obtained earlier by solving the Hodgkin–Huxley equation (in fact, most of the source code originally developed to build the AP function was reused in this Cell-DEVS model).



(a) (b)

FIGURE 8.26 Model execution using Hodgkin–Huxley equations: (a) individual cell; (b) cell space.

This model can also be defined using Cell-DEVS/GDEVS and QDEVS, as shown in [14]. The first step in such a study is to find a polynomial approximation to the original PDE defining the cell's behavior. The simplest way of doing so is to approximate the initial equation's experimental data using eight polynomials of degree one [15] (a higher level of accuracy can be obtained using GDEVS of a higher level with the same number of states and events). The identification of the parameters in each of the polynomials was obtained minimizing a quadratic criterion using minimum squares. The polynomials are defined by

$$P_i(t) = a_i\, t + b_i \qquad \forall i \in [1, 8] \tag{8.4}$$

using the coefficients in Table 8.1.

Figure 8.27 shows the result of this approximation function. Between 0 and 2 ms, we approximate the action potential using four different polynomials (because, when the cell is triggered, the signal generated by the Hodgkin–Huxley model is highly nonlinear). We also need a polynomial ending in the first positive value, which will trigger activity in the neighboring cells in this example (polynomial P_2 is in charge of this).

TABLE 8.1
Polynomial Coefficients for the Action Potential Model

I	A_i	b_i	Time (ms)
1	1.0250	–83.1478	[0, 0.35)
2	6.4555	–275.5886	[0.35, 0.43)
3	–0.2765	37.4703	[0.48, 1.48)
4	–0.0661	8.7840	[1.48, 2.48)
5	–0.0073	–8.6492	[2.48, 9.98)
6	–0.0022	–12.1344	[9.98, 17.48)
7	–0.0143	10.6898	[17.48, 60)
8	–0.0016	–64.0617	[60, +∞)

The coefficients in the polynomials are then converted into discrete event signals. Each cell uses polynomial coefficients to compute the current state and to inform the neighbors of the cell's state. As seen in Figure 8.28, the local computing function included in each of the cells receives the current coefficient ($(N)a_i$ and $(N)b_i$) from the neighboring cells. The cell's outputs are the current cell states specified as polynomial coefficients ($a_i(i,j)$ and $b_i(i,j)$). Timing of activation for each polynomial can be easily defined using the model delay functions.

Figure 8.29 shows the model implementation in CD++, as found in *./heartGDEVS.zip*. The cell is inactive until it receives external stimuli from a neighboring cell. In that case, the cell is activated and produces internal state changes (represented by the coefficient in the polynomials, which are

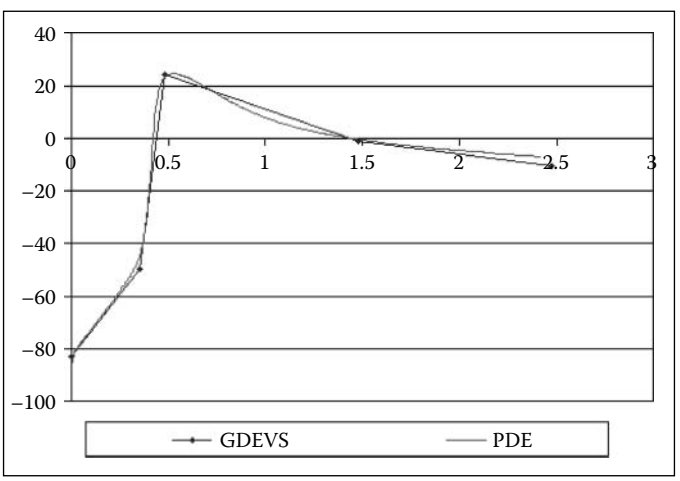


FIGURE 8.27 Linear approximation of the action potential function. (From Giambiasi, N., and G. Wainer. 2005. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:137–151.)

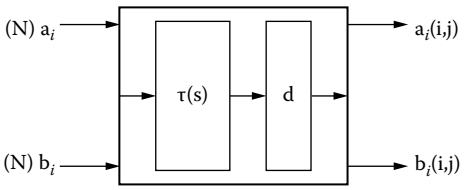


FIGURE 8.28 GDEVS cell. (From Giambiasi, N., and G. Wainer. 2005. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:137–151.)

```
[heart-GDEVS]
type : cell
dim : (6,6)
delay : transport
border : nowrapped
neighbors : (0,-1) (0,0) (-1,0) (-1,-1)
neighbors : (0,1)  (1,0) (-1,1) (1,1) (1,-1)
localtransition : heart-rule-GDEVS

[heart-rule-GDEVS]
rule : { S0 }  0 {(0,0)=-83 and voltage(0,-1) > 0 or voltage(-1,-1) > 0 or voltage(-

1,0)>0 }

rule : { S1, send(1.0250,-83.1478)    } 0.35  { (0,0) = S0 }
rule : { S2, send(6.4555,275.5886)    } 0.08  { (0,0) = S1 }
rule : { S3, send(-0.2765,37.47)      } 0.05  { (0,0) = S2 }
rule : { S4, send(-0.0661,8.784)      } 1     { (0,0) = S3 }
rule : { S5, send(-0.0073,-8.6492)    } 1     { (0,0) = S4 }
rule : { S6, send(-0.0022,-12.1344)   } 7.50  { (0,0) = S5 }
rule : { S7, send(-0.0143,10.6898)    } 7.50  { (0,0) = S6 }
rule : { S8, send(-0.0016,-64.0617)   } 4.25  { (0,0) = S7 }
rule : { S0, send(-0.0016,-64.0617)   } 4.15  { (0,0) = S8 }
rule : { (0,0) }                        0           { t }

[voltage-function]
voltage(cellpos) = cell.ai * time + cell.bi
```

FIGURE 8.29 Cell-DEVS/GDEVS implementation of the heart tissue model.

transmitted to the neighboring cells after the delay). The model flows through eight different states represented by each of the polynomials, plus an extra state to put the model into resting state.

We use a 6 × 6 cell space, transport delays, a nonwrapped model, and Moore's neighborhood. Then we define the local computing function, *heart-rule-GDEVS*. If a stimulus is received when the cell is inactive ((0,0) = –83), it will check the voltage received from the cells in the neighborhood (which is received through ports a_i and b_i and computed by the voltage function), reacting to positive voltage in any of them. It will change to the corresponding state (S_i, to the left of the specification) and will send the current a_i, b_i coefficients to the neighboring cells after the consumption of the delay. Each of the rules represents a cell's state change and the spread of the coefficients to the neighbors. Each of the cells will repeat the behavior defined here while storing the voltage value for display, which is shown in Figure 8.30.

As we can see, we obtained an output trajectory more precise than the one obtained with CA. This gain of precision involved a low extra cost in terms of computing time. Likewise, the complexity added to the cellular model developed in Cell-DEVS is reduced when compared with the solution using PDEs (which required implementing the Hodgkin–Huxley equations). As reported in Giambiasi and Wainer [8] and Wainer [14], this results in performance gains (at the cost of limited error).

```
Line : 83 - Time: 00:00:00:000
              0           1           2           3           4           5
    +-----------------------------------------------------------------------+
   0|     1.97000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   1|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   2|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   3|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   4|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   5|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
    +-----------------------------------------------------------------------+
Line : 115 - Time: 00:00:00:043
              0           1           2           3           4           5
    +-----------------------------------------------------------------------+
   0|     1.97000     1.99791   -83.00000   -83.00000   -83.00000   -83.00000|
   1|     1.99791     1.99791   -83.00000   -83.00000   -83.00000   -83.00000|
   2|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   3|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   4|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   5|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
    +-----------------------------------------------------------------------+
Line : 199 - Time: 00:00:00:086
              0           1           2           3           4           5
    +-----------------------------------------------------------------------+
   0|    24.19800    24.19800     1.99791   -83.00000   -83.00000   -83.00000|
   1|    24.19800    24.19800     1.99791   -83.00000   -83.00000   -83.00000|
   2|     1.99791     1.99791     1.99791   -83.00000   -83.00000   -83.00000|
   3|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   4|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
   5|   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000   -83.00000|
    +-----------------------------------------------------------------------+
```

FIGURE 8.30 Cell-DEVS/GDEVS execution of the heart tissue model. (From Giambiasi, N., and G. Wainer. 2005. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:137–151.)

Exercise 8.8

Study the error of the GDEVS approximation versus the original AP data.

Exercise 8.9

Study the performance of the implementation of the model using Cell-DEVS, comparing the results with GDEVS and QDEVS.

8.7 ENERGY PATHWAYS IN MITOCHONDRIA

We will now present a model introduced in Djafarzadeh, Mussivand, and Wainer [16] and found in *./glyco10.zip,* which is focused on the detailed analysis of the behavior of the mitochondrion, which fulfills different important roles in cellular metabolism [17,18]. The model in this section includes two biological pathways, putting emphasis on cellular metabolism and energy production aspects.

Mitochondria are small double-membrane organelles found in the cytoplasm of eukaryotic cells. Mitochondria are responsible for converting nutrients into the energy-yielding molecule adenosine triphosphate (ATP) to fuel the cells' activities [19]. Mitochondria can be divided into four components: outer membrane, intermembrane space, inner membrane, and the matrix (see Figure 8.31).

The chief function of the mitochondria is to create energy for cellular activity by the process of aerobic respiration. In this process, glucose is broken down in the cell's cytoplasm, via the **glycolysis** process, to form pyruvic acid. In a series of reactions, part of which is called the Krebs cycle, the pyruvic acid reacts with water to produce carbon dioxide and hydrogen. Energy is released as the electrons flow from the coenzymes down the electron transport chain to the oxygen atoms. The enzyme ATPase, which is embedded in the inner membrane, adds a phosphate group to adenosine diphosphate (ADP) in the matrix to form ATP. Aerobic respiration is an ongoing process, and

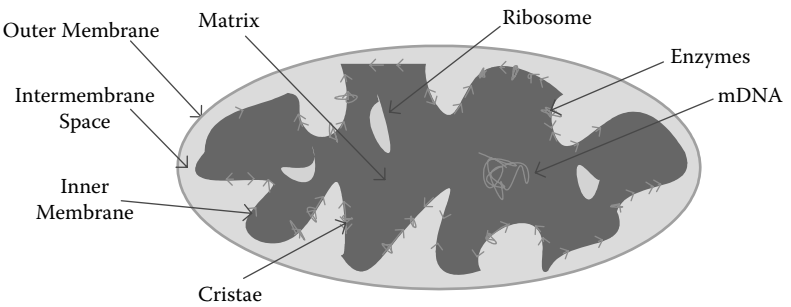


FIGURE 8.31 Scheme of the mitochondrion.

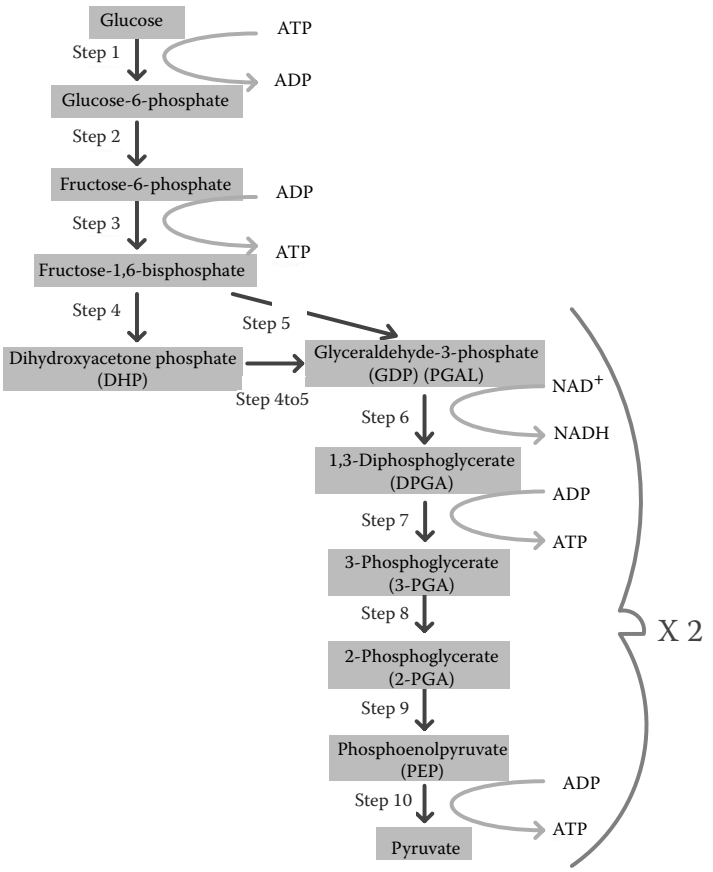


FIGURE 8.32 Glycolysis pathway. (From Curtis, H., and N. Barnes. 1989. *Biology,* 5th ed. New York: W. H. Freeman.)

mitochondria can produce hundreds of thousands of ATP molecules per minute. ATP is transported to the cytoplasm, where it is used for virtually all energy-requiring reactions. As ATP is used, it is converted into ADP, which is returned by the cell to the mitochondrion and is used to build more ATP [19]. Specific enzymes control each of the different reactions, as shown in Figure 8.32.

The glycolysis pathway was defined as a DEVS coupled model and it was implemented using CD++. For example, *Step 1* (in which glucose is phosphorylated by ATP to form glucose 6-phosphate and ADP [17]) can be defined as the atomic model

$$Step1 = < S, X, Y, \delta_{int}, \delta_{ext}, ta, \lambda > \tag{8.5}$$

where

S = {atpc, glucosec, ifhex, counter, phase, sigma};
X = {glucose, ATPi, hexokinase};
Y = {glucose_6_phosphate, ADP, H}; and

δ_{int}, δ_{ext}, ta, and λ are presented in Figures 8.33–8.35 (using CD++ implementation).

The external transition function presented in Figure 8.33 is invoked every time *glucose, ATPi,* or *hexokinase* is received by the model; as a result, the model simulates the reactions previously described in Figure 8.32. As we can see in the figure, whenever a substance is present, its value is added to a counter representing the number of molecules available (*glucosec*, *atpc*). In the case of the *hexokinase* (which is an enzyme), only presence is considered (whenever the enzyme is present, a value is set to *true*). The internal transition function schedules an internal event after a preparation time describing the timing for the transfer. If glucose, ATPi, and hexokinase are in the system, then the reaction will take place. When the time interval expires, the *output function* is invoked and the first value in *Step1* is sent through the corresponding output port.

In Figure 8.34, the output function is activated when all the conditions of the external function have been satisfied; that is, all three input events are in and the reaction can happen. As a result, *ADP, glycose_6_phosphate,* and *H* will be sent out through the corresponding output ports.

```
Model &Step1::externalFunction ( const ExternalMessage &msg ) {
        if( msg.port() == glucose ) {
                glucosec = glucosec + msg.value() ;
                if ( (atpc > 0 ) && (ifhex == true))
                        holdIn( active, Prep_Gluc );
        }
        else if( msg.port() == ATPi ) {
                atpc = atpc + msg.value() ;
                if ( (glucosec>0) && (ifhex==true) )
                        holdIn( active, Prep_ATPi );
        }
        else if ( msg.port() == hexokinase ) {
                ifhex = true ;
                if ( (glucosec > 0 ) && (atpc > 0) )
                        holdIn( active, Prep_Hexo );
        }
}
```

FIGURE 8.33 External transition function (δ_{ext}) for *Step1*.

```
Model &Step1::outputFunction(InternalMessage &msg) {
        if ( counter != 0 )    {
          sendOutput( msg.time(), ADP, counter );
    sendOutput( msg.time(), glucose_6_phosphate, counter );
          sendOutput( msg.time(), H, counter );    }
}
```

FIGURE 8.34 Output function (λ) for *Step1*.

```
Model &Step1::internalFunction(const InternalMessage &)  {
            counter = 0;
                 if ( (atpc >= 1) && (glucosec >= 1 )
                 && (ifhex == true ) )  {
                           if (atpc > glucosec) {
                             atpc = atpc - glucosec;
                             counter=glucosec; glucosec=0;
                            }
                            else if (atpc < glucosec) {
                               glucosec = glucosec-atpc;
                                counter = atpc; atpc = 0;
                            }
                            else if (atpc == glucosec) {
                                counter = atpc;
                                atpc = glucosec = 0;
                            }
                  }
             passivate();
```

FIGURE 8.35 Internal transition function (δ_{int}) for *Step1*.

```
[top]
components   :   step1@Step1   step2@Step2   step3@Step3   step4@Step4   step4to5@Step4to5
     step5@Step5 step6@Step6 step7@Step7 step8@Step8 step9@Step9 step10@Step10
out : H ADP NADH H2O pyruvate ATPo
in : glucose ATPi hexokinase phosphoglucoisomerase PFK isomerase aldolase G3PD NAD P PGK
     PGM enolase pyruvate_kinase
Link : glucose glucose@step1
Link : ATPi ATPi@step1
Link : hexokinase hexokinase@step1
...
Link : aldolase aldolase@step4
Link : isomerase isomerase@step4to5
...
Link : ATPo@step7 ATPo
Link : H2O@step9 H2O
Link : pyruvate@step10 pyruvate
Link : ATPo@step10 ATPo
```

FIGURE 8.36 Glycolysis coupled model.

After calling the output function, the internal transition function (shown in Figure 8.35) is invoked. This function will produce an internal state change according to the substances available in the mitochondria. The function updates the number of substances available according to the reaction, and it then *passivates*.

The remaining steps explained in Figure 8.32 (steps 2–10) were developed using a similar approach [16]. Following the description for the glycolysis, we built a DEVS coupled model including all the steps previously defined as atomic models, as seen in Figure 8.36.

When we execute this model in CD++, we can study the model's behavior by analyzing its outputs. One simulation scenario we created validating the glycolysis model is presented in Table 8.2.

These simulation results accurately describe the reactions that occurred during glycolysis [10], following the ideas shown in Figure 8.32. Tables 8.2 and 8.3 show the input/output trajectories for the model and *Step1,* respectively. As we can see in Table 8.3, by time 30:00, we have the three inputs required to produce a reaction. At time 30:00, two *glucose*s and six *ATPi* enter the system, generating two *ADP,* two *glucose_6_phosphate,* and two *H* molecules.

Exercise 8.10

Introduce a nonexpected behavior (erroneous transition functions, different substances present, different reactions on each step) in the glycolysis model. Analyze the results obtained.

TABLE 8.2
Inputs/Outputs for Glycolysis Model

Inputs	Outputs
10:00 glucose 2	50:000 h 2
18:00 ATPi 3	72:000 nadh 2
50:00 hexokinase 1	72:000 h 2
51:00 phosphGlucoiSom 1	72:000 atpo 2
52:00 PFK 2	72:000 h2o 2
53:00 isomerase 1	72:000 atpo 2
55:00 aldolase 1	72:000 pyruvate 2
62:00 G3PD 1	
63:00 PGK 1	
64:00 PGM 1	
65:00 enolase 1	
67:00 pyruvKinase 1	
70:00 NAD 3	
72:00 P 2	

TABLE 8.3
Inputs and Outputs for Step 1

Inputs	Outputs
15:00 hexokinase 1	30:000 adp 2
30:00 glucose 2	30:000 glucose_6_phosph 2
30:00 ATPi 6	30:000 h 2
40:00 glucose 4	40:000 adp 4
40:00 ATPi 1	40:000 glucose_6_phosph 4
55:00 glucose 1	40:000 h 4
55:00 ATPi 1	55:000 adp 1
65:00 glucose 1	55:000 glucose_6_phosph 1
	55:000 h 1
	65:000 adp 1
	65:000 glucose_6_phosph 1
	65:000 h 1

Figure 8.37 shows the execution results for *Step1* using CD++ Modeler. We see that at time 40:00, four *glucose* molecules enter the system generating four more outputs of each of the *ADP, glucose_6_phosphate,* and *H* molecules. We used CD++/Maya [20] to create three-dimensional graphics and animation for the glycolysis model (CD++/Maya will be described in detail in Chapter 16). Figure 8.38 shows a snapshot of this animation.

The Krebs cycle, also called the tri-carboxylic acid (TCA) cycle and the citric acid cycle (CAC), oxidizes pyruvate formed during the glycolysis pathway into CO_2 and H_2O. This cycle is a series of chemical reactions of central importance in all living cells that utilize oxygen. The citric acid cycle takes place within the mitochondria in eukaryotes and within the cytoplasm in prokaryotes. For each turn of the cycle, 12 ATP molecules are produced—one directly from the cycle and 11 from the oxidation of the three NADH and one $FADH_2$ molecules produced by the cycle by oxidative phosphorylation [10]. Glucose is converted by glycolysis into pyruvate. Pyruvate enters the mitochondria, linking glycolysis to the Krebs cycle. This step (step A) is also called the bridging step.

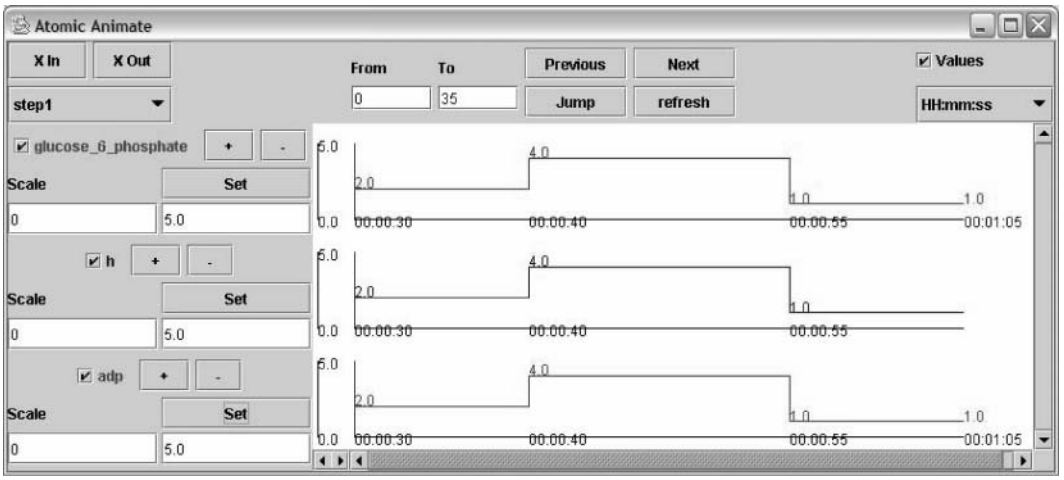


FIGURE 8.37 Atomic animation of step 1 of glycolysis.

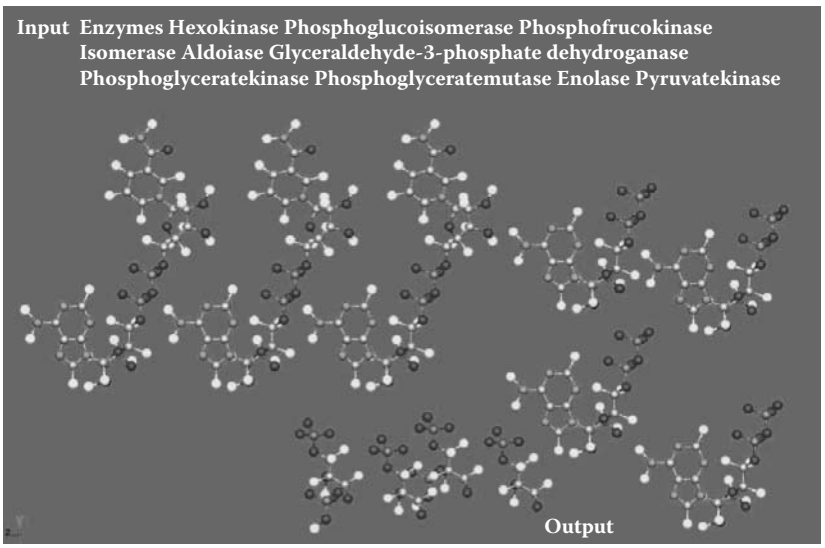


FIGURE 8.38 Visualization of the glycolysis execution in CD++/Maya: step 6.

Pyruvate dehydrogenase—a complex of three enzymes and five coenzymes—oxidizes pyruvate using NAD^+ to form acetyl CoA, NADH, and CO_2.

We defined a model of the Krebs cycle (depicted in Figure 8.39 using identical principles to the ones used for the glycolysis model). In *StepA* of Figure 8.39, pyruvate is degraded and combined with coenzyme A to form acetyl coenzyme A. NADH and CO_2 are released during this process (in fact, *StepA* is the link between glycolysis and the Krebs cycle). Figure 8.40 shows the definition of these coupled models using CD++Modeler.

Figure 8.41 shows snapshots of reactions in the Krebs cycle animation done in CD++/Maya. Figure 8.41(a) shows the beginning of the reaction, in which one pyruvate and four NAD^+ appear. Figure 8.41(b) shows the formation of acetyl CoA and the production of carbon dioxide and NADH as by-products.

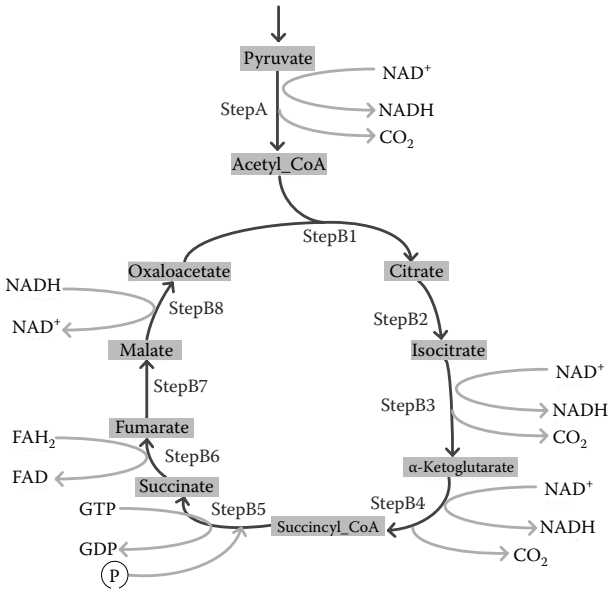


FIGURE 8.39 Krebs cycle reactions.

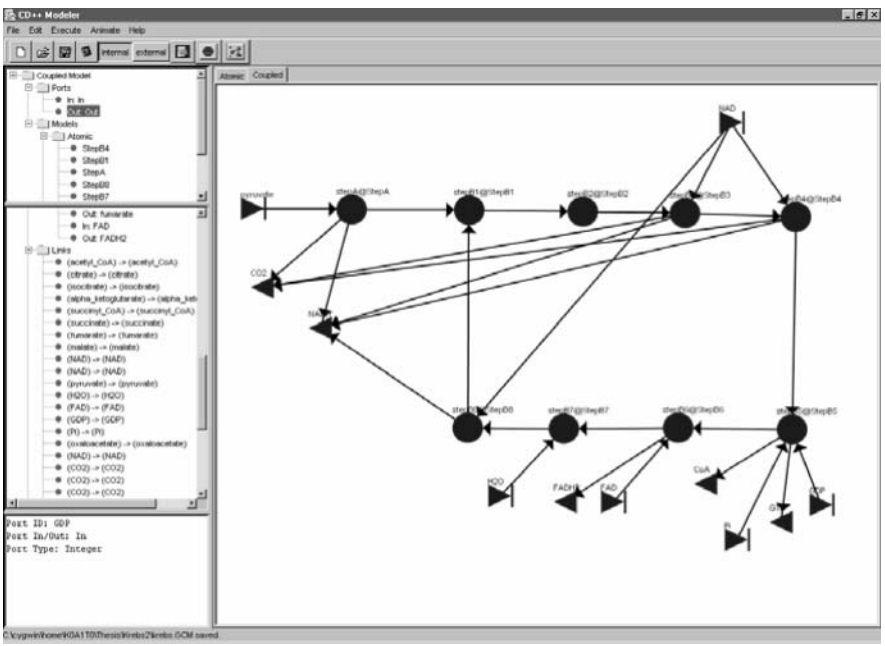


FIGURE 8.40 Defining the Krebs coupled model using CD++ Modeler.

8.8 SUMMARY

In the last few decades, computer simulation has become an integral part in the basic and applied fields of biological research. In this chapter, we showed how to model these kinds of systems using DEVS.

We focused on different examples. Initially, we presented a cellular model showing how synapsin and vesicles interact in nerve cells. This model allows us to study molecular interaction at the

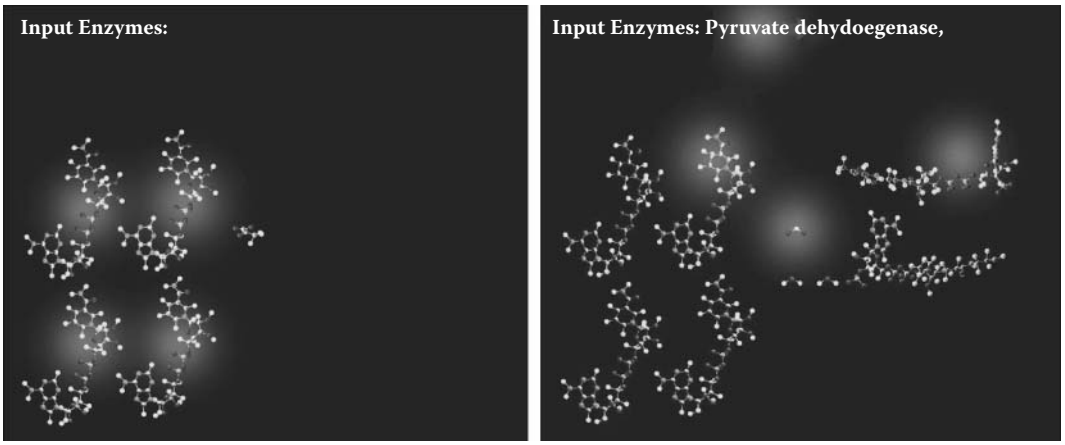


FIGURE 8.41 (a) The Krebs cycle begins; (b) acetyl CoA is formed.

level of the intramembrane of the cell. We then defined a model of the lobules in the human liver and showed their interaction. The model of spreading bacteria can be used to analyze their spread, which causes varied sicknesses. A cellular model presented allows us to analyze how a plague can spread within a population. The Hodgkin–Huxley model we included describes electrical behavior of the heart tissue. We compared the results obtained against those originally built with PDEs and cellular automata. We showed that we could provide adequate levels of precision at a fraction of the computing cost of differential equations. Finally, we included a detailed model of energy pathways in mitochondria (namely, glycolysis and Krebs cycle), which allows analyzing the molecular interactions at the level of the organelle. Other models in the field can be found in the central repository, including models on the formation of cancer (found in *./cancer.zip*) and on the relation between tumors and the immune system (found in *./Tumor-Immune_System.zip*). We also included other models on the spreading of disease (found in *./HIV.zip* and *./Epidemics.zip*), on enzyme kinetics (*./EnzymeKinetics.zip*), on the cell's membrane behavior in neurons (*./Nerve_Cell_Membrane.pdf*), and on the behavior of spiking neurons (*./SpikingNeuronTerminal.zip*).

We have shown that these models can be built and validated incrementally by using simple subcomponents. The approach also enables reuse of simulation components and allows seamless integration of these components into more complex simulation models.

The hierarchical and discrete-event capabilities of DEVS make it a good choice for modeling biological events. As illustrated here, CD++ can be used to model and simulate biological models using a systematic method with models that consist of sets of lower-level interactions.

REFERENCES

1. Benfenati, F., F. Valtorta, and P. Greengard. 1991. Computer modeling of synapsin I binding to synaptic vesicles and F-actin: Implications for regulation of neurotransmitter release. *Proceedings of the National Academy of Sciences USA* 88:575–579.
2. Bain, R., S. Jafer, M. Dumontier, G. Wainer, and J. Cheetham. 2006. Vesicle, synapsin and actin concentration time series modelling at the presynaptic nerve terminal (poster). *Proceedings of Symposium on Progress in Systems Biology 2006,* Ottawa, ON, Canada.
3. Wainer, G., B. Al-aubidy, A. Dias, R. Bain, S. Jafer, M. Dumontier, and J. Cheetham. 2007. Advanced DEVS models with applications to biomedicine. *Proceedings of AIS'2007 Artificial Intelligence, Simulation and Planning,* Buenos Aires, Argentina.

4. Hunt, C. A., G. Ropella, M. Roberts, and L. Yan. 2005. Biomimetic in silico devices. *Proceedings of Computational Methods in Systems Biology 2004; Lecture Notes in Bioinformatics 3082,* 35–43.
5. Ameghino, J., E. Glinsky, and G. Wainer. 2003. Applying cell-DEVS models of complex systems. *Proceedings of Summer Computer Simulation Conference,* Montreal, QC, Canada.
6. Shang, H., and G. Wainer. 2005. A model of virus spreading in CD++. *Proceedings of the International Conference on Computational Science,* Atlanta, GA.
7. Dzwinel, W. 2004. A cellular automata model of population infected by periodic plague. *Proceedings of ACRI 2004, LNCS 3305,* 464–473.
8. Giambiasi, N., and G. Wainer. 2005. Using G-DEVS and cell-DEVS to model complex continuous systems. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:137–151.
9. Goldschlager, N., and M. Goldman. 1989. *Principles of clinical electrocardiography.* Norwalk, CT: Appleton and Lange.
10. Alberts, B., D. Bray, L. Lewis, M. Raff, K. Roberts, and D. Watson. 1983. *Molecular biology of the cell,* 1st ed. New York: Garland Publishing, Inc.
11. Hodgkin, A., and A. Huxley. 1952. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology* 117:500–544.
12. Saxberg, B., and R. Cohen. 1991. Cellular automata models of cardiac conduction. In *Theory of heart,* edited by L. Glass, P. Hunter, and A. McCulloch. New York: Springer–Verlag.
13. Fenton, F. 2000. Numerical simulations of cardiac dynamics. What can we learn from simple and complex models? *IEEE Computers in Cardiology* 27:251–254.
14. Wainer, G. 2004. Performance analysis of continuous cell-DEVS models. *Proceedings of High Performance Computing & Simulation (HPC&S) Conference, 18th European Simulation Multiconference*, Magdeburg, Germany.
15. Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1986. *Numerical recipes.* Cambridge: Cambridge University Press.
16. Djafarzadeh, R., T. Mussivand, and G. Wainer. 2005. Modeling energy pathways in cells. *Proceedings of 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference,* San Diego, CA.
17. Krauss, S. 2001. Mitochondria: Structure and role in respiration. In *Nature encyclopedia of life sciences.* New York: Nature Publishing Group.
18. Poulton, J., and L. Bindoff. 2000. Mitochondrial respiratory chain disorders. In *Nature encyclopedia of life sciences.* New York: Nature Publishing Group.
19. Curtis, H., and N. Barnes. 1989. *Biology,* 5th ed. New York: W. H. Freeman.
20. Khan, A., G. Wainer, W. Venhola, and M. Jemtrud. 2005. On the use of CD++/Maya for visualization of discrete-event models. *Proceedings of IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation,* Paris.

APPENDIX

In order to program the AP function introduced in Figure 8.25 (and other similar functions to be incorporated in the model execution), the user must:

1. Write the function in C++.
2. Copy the function into the real functions source file (*realfunc.cpp* in the internal folder). Replace the type of the arguments with the CD++ types. For example, `double` must be replaced by `Real`. For instance,

```
#include <math.h>
#include <stdio.h>
Real MyFunc(const Real &r1, const Real &r2) {
     double var;
     var=r1.value()*r2.value() - 450.5;
  return Real(var);
 }
```

3. Add the prototype of the function in the real functions header file (*realfunc.h*).

4. Overload the *operator( )* of the Z class for the structure of the function with the correct kind (unary, binary, etc.) on the same header file (*realfunc.h*). For instance,

```
template <class T, class Z>
    struct r_MyFunc : public binary_function< T, T, Z>  {
    Z operator()(const T& t1, const T& t2) const  {
           if (EvalDebug().Active())EvalDebug().Stream() << " (myfunc) ";
           return MyFunc(t1,t2);
         }
      string type(){ return "MYFUNC";}
 };
```

5. Create the type of the function (for instance, *r_MyFunc*) on the same header file (*realfunc.h*):

```
typedef r_MyFunc< Real, Real > REAL_MYFUNC;
```

6. Define the respective operator for the class to manage the type value for the new function on the *synnode.h* header file. For instance,

```
typedef BinaryOpNode< REAL_MYFUNC, RealType, RealType > FuncMyFunc ;
```

7. Add the name and type of the new function on the dictionary of parser method (*parser.cpp* source file):

```
dict[ "myfunc" ] = ValuePair( BINARY_FUNC, new FuncMyFunc() ) ;
```

After recompiling CD++, the new function is available to use from the model. It is activated as follows:

```
rule : { Myfunc(cellpos(0)*1000+cellpos(1)*10+ if( (-1,0) > 0, 1.0,
0.0) } 5 {t}
```

9 Models in Defense and Emergency Planning

9.1 INTRODUCTION

In recent years, a wide range of novel modeling and simulation (M&S) techniques have become popular in the fields of defense and emergency planning (and some of these techniques have also been applied in computer games). As discussed in Palmore [1], there are obvious reasons for using simulation in this area: although warfare and emergencies are common, we cannot generate conflicts or catastrophic situations to study the results of different strategies, equipment, or advanced technologies. In addition, in these scenarios, obtaining real data and making accurate observations are complex, and making deliberate changes (e.g., in the face of combat) is extremely difficult.

In this chapter we will focus on how to create DEVS models with application in this area. We will first introduce a simple collision detection model using Cell-DEVS. Then we present a DEVS model for the synchronization of radar transmitters and receivers. We then introduce a Cell-DEVS model of the behavior of a target seeker and a model of land battlefields. Finally, we show a basic model on evacuation of buildings and how to describe flocking behavior of people being evacuated.

9.2 A SIMPLE MODEL OF AN UNMANNED VEHICLE

The first model to be introduced considers a simple unmanned aerial vehicle (UAV) using Cell-DEVS (presented in Madhoun and Wainer [2] and found in *./Collision_AvoidanceUAV.zip*). The UAV traverses a specific area searching for a target and avoiding static and moving obstacles in its way. The model deals with multiple UAVs moving and avoiding multiple obstacles. In order to model this behavior, each entity is assigned a state value, as shown in Figure 9.1.

The model in CD++ specification language is shown in Figure 9.2. The first portion of the coupled model defines the cell-space geometry, size, and neighborhood shape. Then we define the rules that govern model execution. As shown in the figure, the cell space is composed of 20×20 cells with transport delays. We show part of the rule definition of the static obstacles, UAVs, and moving obstacles. The *uav-rule* implements the UAV movement avoiding the static and moving obstacles, and the *move target rule* implements a moving obstacle from south to north.

Figure 9.3 shows a snapshot of the execution of this model with the allocations of UAV and obstacles. The UAVs (shown in dark gray) try to move from north to south facing static obstacles (shown in black) as well as moving obstacles (shown in light gray).

9.3 RADAR TRANSMITTER–RECEIVER

We will show how to integrate components of a radar system, as discussed in MacSween and Wainer [3]. The model examines the synchronization effects between radar receivers and transmitters. Radar transmitters use a particular frequency, with a given pulse rate, azimuth (i.e., the horizontal direction angle from north toward east), and beam width. Radar scanning receivers work on a tuned frequency (for a specified duration), with a particular azimuth and beam width, and have a tuning time associated with the change from one listening frequency to another.

	Empty Cell	UAV	Moving Obstacle	Static Obstacle
Color				
Movement	None			None
State	0	1	5	9

FIGURE 9.1 UAV state values.

```
[top]
type : cell
width : 20
height : 20
delay : transport
border : nowrapped
neighbors : (-2,-2) (-1,-2) (0,-2) (1,-2) (2,-2) (-2,-1) (-1,-1) (0,-1) (1,-1) (2,-1)
neighbors : (-2,0) (-1,0) (0,0) (1,0) (2,0) (-2,1) (-1,1) (0,1) (1,1) (2,1)
neighbors : (-2,2) (-1,2) (0,2) (1,2) (2,2) (3,-2) (3,-1) (3,0) (3,1) (3,2)

localtransition : uav-rule

[uav-rule]
...
%up
rule : 1 100 { (0,0)=0 and (1,0)=1 and (2,1)=5 and (1,-1)!=0 and (2,0)!=0 }
rule : 0 100 { (0,0)=1 and (-1,0) =1 }
...
%***************************************************************
%moving target rule
rule : 5 100 { (1,0) = 5 }
rule : 0 100 { (-1,0) = 5 }
```

FIGURE 9.2 UAV coupled model specification. (From Madhoun, R., and G. Wainer. 2005. *Journal of Defense Modeling and Simulation* 2:121–143.)

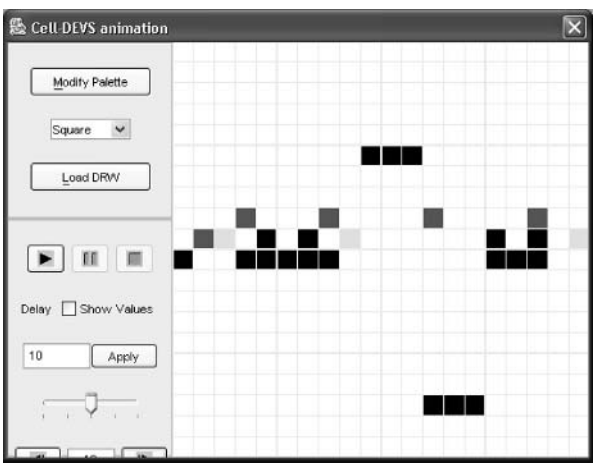


FIGURE 9.3 Initial allocations of UAVs and obstacles.

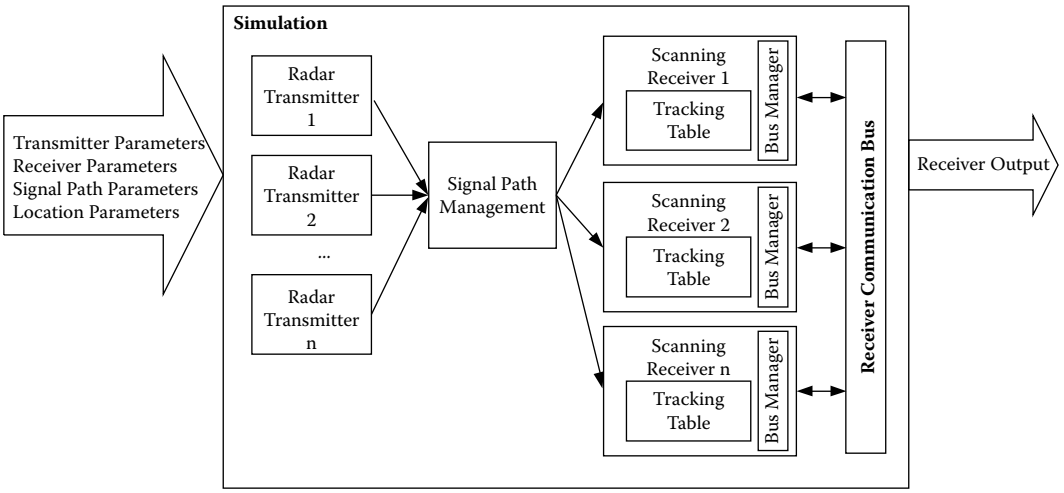


FIGURE 9.4 Structure of the radar transmitter/receiver model.

The interception of radar signals by a radar scanning receiver can be severely limited if the scan rate becomes synchronized with the one on its own transmitter. This synchronization effect occurs when a transmitter sends out radar pulses periodically but the receiver is scheduled to scan with a period such that the receiver is never listening to the transmitter. Therefore the transmitter can never be detected by the receiver, even though it may be transmitting. Every effort must be made to generate a receiver scan pattern that limits this effect because it seriously degrades the probability of intercept for the receiver. Our model generates simulation data to study this effect.

The scenario can be even more complex when we consider that multiple receivers can be available and communicate with each other (with each receiver notifying the others about radar transmitters that have been detected). Each receiver is connected to a bus, and it maintains a tracking table with information about the currently known transmitters. Figure 9.4 shows the structure of a DEVS model that we used to analyze the behavior of this system.

The *scanning receiver* atomic model (Figure 9.4) can be defined as follows:

$$\text{Scanning receiver} = < X, Y, S, \delta_{ext}, \delta_{int}, ta, \lambda > \tag{9.1}$$

where

X = { ext_signal }
Y = { notify, detected_signal_properties }
S = {Scan, Signal_Detected, Process_Signal, Notify}
δ_{ext} = { δ_{ext}(Scan, ext_signal) = Signal_Detected }
δ_{int} = { δ_{int}(Signal_Detected) = Process_Signal,
δ_{int}(Process_Signal) = Notify
δ_{int}(Notify) = Scan
}
ta = { ta(Signal_Detected) = DETECTION_TIME
ta(Process_Signal) = PROCESS_TIME
ta(Notify) = NOTIFY_TIME
ta(Scan) = INFINITY
}
λ (S) = { λ (Signal_Detected) = notify,
λ (Process_Signal) = detected_signal_properties
}

The model's states (S) represent a receiver that is scanning for a new signal, has detected one, is processing it, or is notifying the reception of the signal through the bus, respectively. As seen in the external transition funciton definition, the scanning receiver (*scan* state) is waiting to receive a new external signal (through the *ext_signal* input port). When this occurs, the model changes to the *Signal_Detected* state, which (according to the definition of the function *ta*) will be maintained during *DETECTION_TIME* units—the time the circuit takes to react to a signal detected. When *DETECTION_TIME* is consumed, the output function is executed, and the *notify* output port is used to inform about the state change. Then the internal transition function executes, changing the state to *Process_Signal*, which represents the fact that the new signal must be processed. This state lasts *PROCESS_TIME* units. When this time is consumed, the second line of the output function is executed, which informs that the signal property has been detected (by sending a signal through the *detected_signal_properties* output port). We need to model the time that the circuit takes for notification. Therefore, we change to the *notify* state, which lasts *NOTIFY_TIME* units. This state change does not generate outputs, but when the time is consumed, the model will be in *scan* state again (awaiting a new input forever).

This model was subsequently built in CD++ (and it can be found in *./RadarFreq.zip*). We used both C++ and DEVS graphs to model the radar's behavior (whose specification is presented in Figure 9.5).

As discussed in Chapter 5, CD++Modeler shows two views of the state machine: the left panel contains a sorted tree diagram and the right side contains a visual representation of the model. The four states of the scanning receiver are immediately apparent. When this model is exported to CD++ textual notation, the specification seen in Figure 9.6 is created.

As we can see in Figure 9.6, the text specification in CD++ is a direct mapping from the formal specification previously presented. A few syntactic variations can be found (mostly, the definition

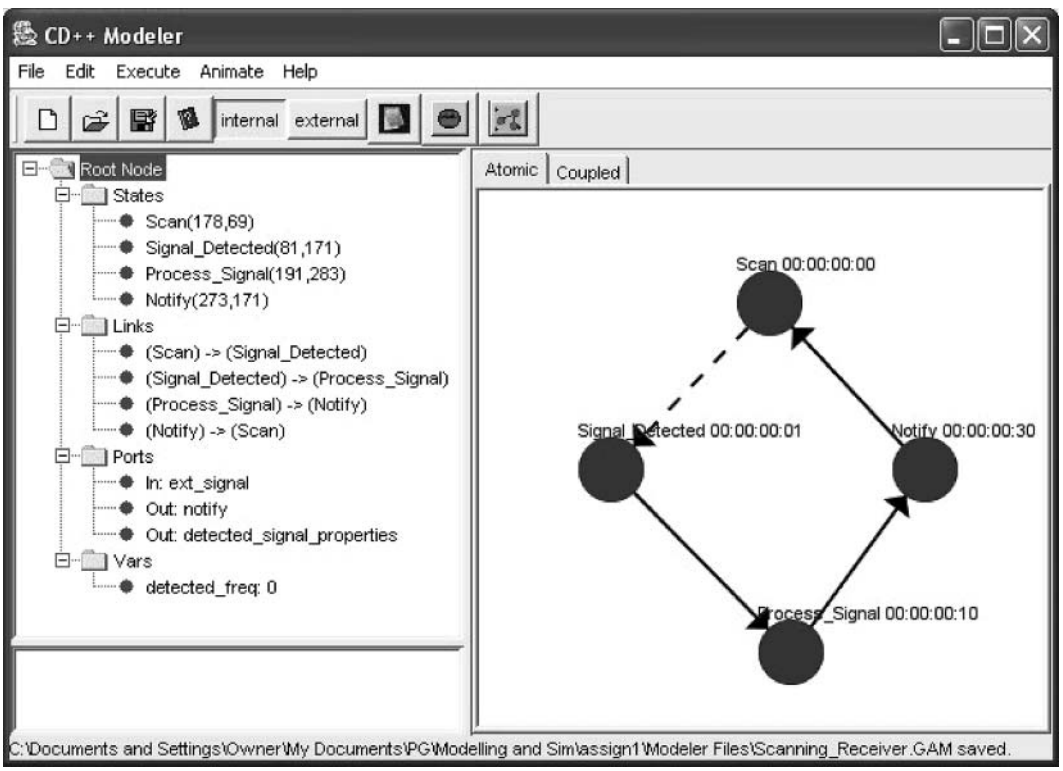


FIGURE 9.5 Graphical definition of the scanning receiver.

```
[Scanning_receiver]
in: ext_signal
out: notify detected_signal_properties
var: detected_freq
state: Scan Signal_Detected Process_Signal Notify
initial : Scan
ext: Scan?ext_signal Signal_Detected
int: Signal_Detected Process_Signal notify!1
int: Process_Signal Notify detected_signal_properties!value()
int: Notify Scan notify!0 detected_signal_properties!0
Scan:00:00:00:00
Signal_Detected:00:00:00:01
Process_Signal:00:00:00:10
Notify:00:00:00:30
detected_freq:0
```

FIGURE 9.6 Text specification of the scanning receiver model.

of an initial state for the state machine defined by the DEVS graph, and the inclusion of the output functions on the lines corresponding to the state changes, which makes writing the model easier).

Each of the models presented in Figure 9.4 was defined in a similar way to the scanning receiver [3]. Once this stage was completed, a coupled model was built, integrating all of the systems' components. Three *Transmitter* atomic models are defined, combined with two network receivers, as in Figure 9.4, and is formally defined as:

$$\text{Net\_Of Network\_Receivers} = < X, Y, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select > \tag{9.2}$$

where

X = { ∅ }
Y = { notify1, notify2, notify3 }
M_i = { Transmitter1, Transmitter2, Transmitter3, Network_Receiver1, Network_Receiver2 }
I_i = { I(Transmitter1) = Network_Receiver1, I(Transmitter1) = Network_Receiver2, I(Transmitter2) = Network_Receiver1, I(Transmitter2) = Network_Receiver2, I(Transmitter3) = Network_Receiver1, I(Transmitter3) = Network_Receiver2 }
Z_{ij} = { (Transmitter1.pulse_out, Network_Receiver1.ext_signal), (Transmitter1.pulse_out, Network_Receiver2.ext_signal), (Transmitter2.pulse_out, Network_Receiver1.ext_signal), (Transmitter2.pulse_out, Network_Receiver2.ext_signal), (Transmitter3.pulse_out, Network_Receiver1.ext_signal), (Transmitter3.pulse_out, Network_Receiver2.ext_signal) }
select = { (Transmitter*i*, Network_Receiver*j*) = Network_Receiver*j* i ∈ [1, 3], j ∈ [1, 2]}

The network receiver is a coupled model composed of a scanning receiver and tracking table, as follows:

$$\text{Network\_Receiver} = < X, Y, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select > \tag{9.3}$$

where

X = { test_signal, bus_receive_freq, bus_receive_id }
Y = { notify, bus_send_id, bus_send_freq }
M_i = { Tracking_Table, Scanning_Receiver }
I_i = { I(Tracking_Table) = Scanning_Receiver, I(Scanning_Receiver) = Tracking_Table }
Z_{ij} = { (Scanning_Receiver.detected_signal_properties, Tracking_Table.signal_props), (Tracking_Table.new_freq, Scanning_Receiver.ext_signal) }
select = { (Tracking_Table, Scanning_Receiver) = Scanning_Receiver }

This coupled model can be defined using CD++Modeler graphical notation as shown in Figure 9.7.

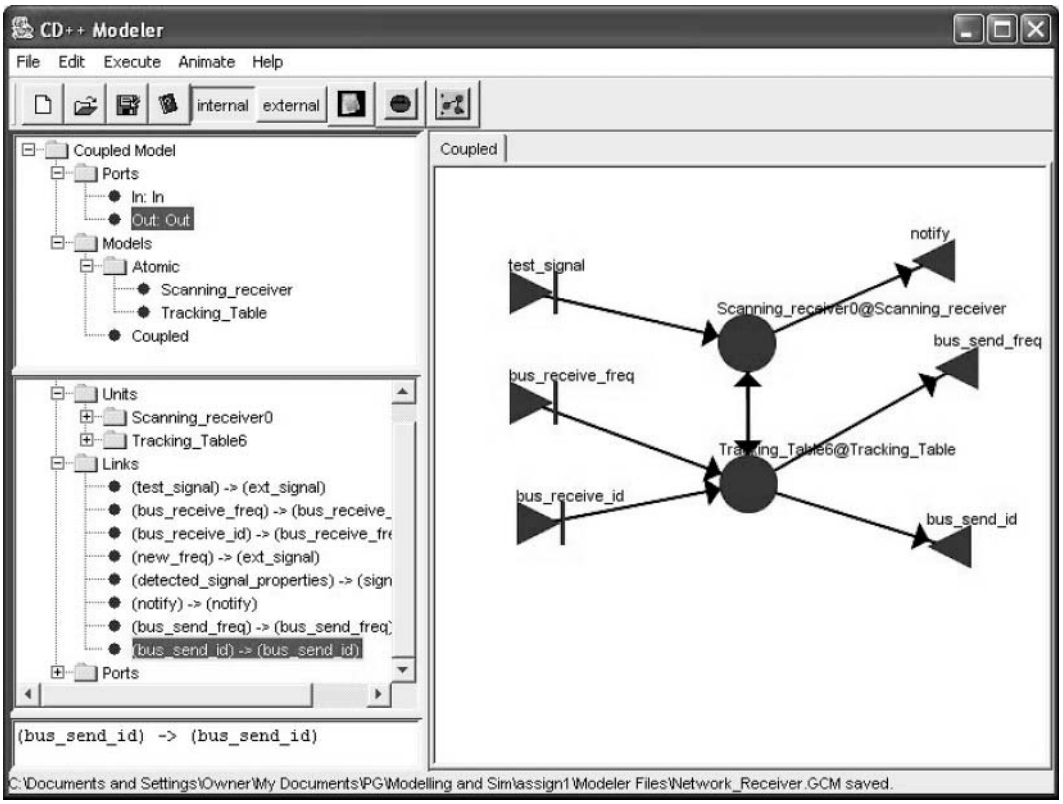


FIGURE 9.7 Network receiver coupled model: graphical representation.

When we export the top-level coupled model to CD++ textual notation, the specification shown in Figure 9.8 is generated. As we can see, there are three *transmitter* atomic models (*tr1, tr2,* and *tr3*), each configured with different frequencies and pulse characteristics. The two network receivers (*netrx1, netrx2*) are also configured to listening to different frequency bands. For each of the coupled models, we have defined the internal and external couplings, including initial values for each of the atomic components. Based on this simulation model, we carried out a variety of tests using different scenarios, including the following:

- Transmitter: pulse is sent at 22 kHz; pulse width = 3 ms; pulse interval = 30 ms.
- Scanning receiver is set to listen for pulses between 18 and 25 kHz.
- The tracking table tests that a signal is recorded and a bus message is sent and then tests that bus messages are received correctly.
- The network receiver tests the reaction to a signal and to bus messages.
- Different initial configurations for the *Net_Of_Network_Receivers* using:
  - Transmitters sending on frequencies not being scanned.
  - Transmitters sending on frequencies scanned by network receiver #1 only.
  - Transmitters sending on frequencies scanned by network receiver #2 only.
  - Transmitters sending on frequencies scanned by both.

Table 9.1 shows the results of testing the scanning receiver (when the model is set to listen for pulses between 18 and 25 kHz). As we can see in the table, the receiver initially gets a pulse (*ext_signal* = 10,000); however, it is not processed, as it is not within the specified range. The same occurs in the second test case (the test signal is 11 kHz). When we receive the external signal on

```
[top]
components: tr1@Transmitter tr2@Transmitter tr3@Transmitter netrx1 netrx2
out: notify1 notify2 notify3
Link: pulse_out@tr1 ext_signal@netrx1
Link: pulse_out@tr1 ext_signal@netrx2
...
Link: notify@netrx1 notify1
Link: notify@netrx2 notify2
...

[netrx2]
components:tt2@Tracking_Table rx2@Scanning_Receiver
in: ext_signal brf brid
out: notify bs_id bs_freq
Link: ext_signal ext_signal@rx2
Link: brf bus_receive_freq@tt2
Link: brid bus_receive_id@tt2
Link: detected_signal_properties@rx2 signal_props@tt2
Link: new_freq@tt2 ext_signal@rx2
Link: notify@rx2 notify
Link: bus_send_id@tt2 bs_id
Link: bus_send_freq@tt2 bs_freq

[rx2]
freq_lower_bound : 18000
freq_upper_bound : 25000

[tt2]
table_id : 2

[tr1]
frequency : 19000
pulseDuration : 00:00:00:5
pulsePeriod : 00:00:00:40
...
```

FIGURE 9.8 Coupled model definition: radar Tx/Rx.

TABLE 9.1
Testing the Scanning Receiver

Event File	Output File
00:00:10:000 ext_signal 10000	
00:00:30:000 ext_signal 11000	
00:01:00:000 ext_signal 22000	00:01:00:001 notify 1
	00:01:00:011 detected_signal_properties 22000
	00:01:00:041 notify 0
	00:01:00:041 detected_signal_properties 0
00:02:20:000 ext_signal 23000	00:02:20:001 notify 1
	00:02:20:011 detected_signal_properties 23000
	00:02:20:041 notify 0
	00:02:20:041 detected_signal_properties 0

the third line (22 kHz), the model changes to the *Signal_Detected* state and waits 1 ms (the delay associated with the state, as shown in the specification in Figure 9.6), and then the output function is activated (which sends the value *1* through the *notify* output port). Then the internal transition function puts the model in *Process_Signal* state for 10 ms. When this time is consumed, the

detected_signal_properties outputs the frequency. We can see a similar scenario in the next test case, where the right frequency is output. Finally, we can see how the outputs are reset when the model goes back to the *scan* mode.

Exercise 9.1

Take the previous model and change the frequency of pulse, width, and interval and then repeat the test cases, trying to detect different synchronization problems.

Exercise 9.2

Change the structure of the model and include a new receiver. Repeat all the test cases previously discussed.

9.4 A TARGET-SEEKING DEVICE

We present a Cell-DEVS model describing the behavior of a simple, target-seeking device introduced in MacSween and Wainer [3] and based on the original model presented in Reynolds [4]. As shown in Figure 9.9, the seeker acts to steer a device toward a specified position in global space. This behavior adjusts the device so that its velocity is radially aligned toward the target, as discussed in Reynolds [4].

As we can see in Figure 9.9, the model acts to steer the seeking device toward a given position. The seeker must adjust the current direction and speed in order to make its velocity align toward the target. Reynolds [4] suggested defining an *Action Selection* for the seeker, which is specified by dictating the destination location. In order to model this seeking behavior using Cell-DEVS, it was necessary to create discrete states to represent the current state of the simple device. The state defines a device with no velocity or motion in one of nine directions: moving northwest (NW; value = 1), north (N; value = 2), northeast (NE; value = 3), west (W; value = 4), stationary (value = 5), east (E; value = 6), southwest (SW; value = 7), south (S; value = 8), and southeast (SE; value = 9). The model uses two planes (one for collision detection and the second for the steering behavior), using the following neighborhood on each plane:

N = { (–2,–2) (–2, –1) (–2, 0) (–2,1) (–2,2) (–1, –2) (–1, –1) (–1,0) (–1,1) (–1,2) (0,–2) (0,–1) (0,0) (0,1) (0,2) (1,–2) (1,–1) (1,0) (1,1) (1,2) (2,–2) (2,–1) (2,0) (2,1) (2,2) }

The specification describes the discrete motion that was implemented to simulate the effect of a desired velocity on a device. To do so, we used the following attributes to define the movement

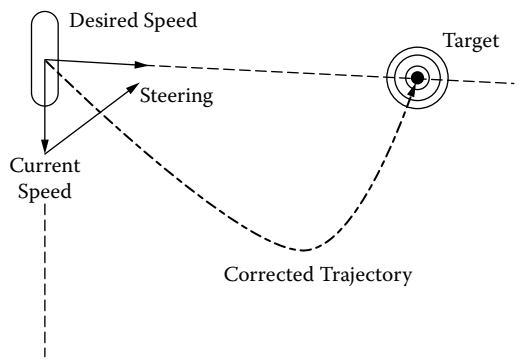


FIGURE 9.9 Informal behavior of the seek model. (From Reynolds, C. W. 1999. *Proceedings of Game Developers Conference,* San Jose, CA.)

of the device: {*mass, position vector, velocity vector, max_force, max_speed, orientation, N basis vectors*}, where $N = 2$. The motion of this device is defined by the following:

steering_force = truncate (steering_direction, max_force)
acceleration = steering_force/mass
velocity = truncate (velocity + acceleration, max_speed)
position = position + velocity
and the new basis vectors are defined by:
new_forward = normalize (velocity)
approximate_up = normalize (approximate_up) // if needed
new_side = cross (new_forward, approximate_up)
new_up = cross (new_forward, new_side)

The seek behavior motion is defined by the following:

desired_velocity = normalize (position-target)*max_speed
steering = desired_velocity – velocity

This basic behavior can be summarized in Figure 9.10.

The multiple combinations of actual and desired velocity could result in the same destination cell for a device. In order to avoid collisions, a simple priority scheme is used when multiple cells want to move into the same cell: stationary devices have the highest priority, cells to the NW have the lowest, and those in the SE have the second highest. Figure 9.11 shows a collision scenario. In Figure 9.11(a), the cells to the N and W yield to those in the S and E (in this case, the cell in gray is the one moving). In Figure 9.11(b), the cell to the S has higher priority than those to the N, so the gray cell to the W is the one to move.

Figure 9.12 shows an excerpt of the model definition in CD++. The model rules, found in *./Target-seeker.zip,* define the discrete motion implemented to simulate the effect of a desired velocity on a device. The first set of rules is in charge of detecting collisions (as shown in Figure 9.11), and the second set of rules shows the model's steering behavior. An input was provided to each cell to specify the desired velocity of the device.

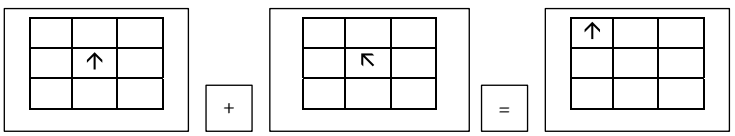


FIGURE 9.10 Definition of update rules.

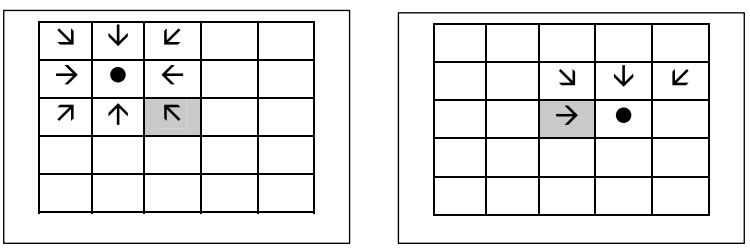


FIGURE 9.11 Collision avoidance examples. (a) cells to N and W yield to those in S and E; (b) cells to the S has higher priority.

```
[seek]
% Layer 0 - Current Position and Direction
% Layer 1 - Desired Velocity
type : cell
dim : (20,30,2)
delay : transport
defaultDelayTime : 100
border : wrapped

neighbors : (-2,-2,0) (-2,-1,0) (-2,0,0) (-2,1,0) (-2,2,0) (-1,-2,0) (-1,-1,0) (-1,0,0)
(-1,1,0) (-1,2,0) (0,-2,0) (0,-1,0) (0,0,0) (0,1,0) (0,2,0) (1,-2,0) (1,-1,0) (1,0,0)
(1,1,0) (1,2,0) (2,-2,0) (2,-1,0) (2,0,0) (2,1,0) (2,2,0)   (-2,-2,1) (-2,-1,1) (-2,0,1)
(-2,1,1) (-2,2,1) (-1,-2,1) (-1,-1,1)  (-1,0,1) (-1,1,1) (-1,2,1) (0,-2,1) (0,-1,1)
(0,0,1) (0,1,1) (0,2,1) (1,-2,1) (1,-1,1) (1,0,1) (1,1,1) (1,2,1) (2,-2,1) (2,-1,1)
(2,0,1) (2,1,1) (2,2,1)
% Desired Velocity layer stays the same
zone : constant { (0,0,1)..(19,29,1) }
localtransition : move-rule

[move-rule]
% Current Position and Direction Layer (0,*,*)

% Collision when moving up and left
rule : 5 100 {
  % We are actually moving up and left (11,51,21,24,27,41,42,43 combos):
  ( ( ((0,0,0)=1) and ((0,0,1)=1) ) or ( ((0,0,0)=5) and ((0,0,1)=1) ) or
        ( ((0,0,0)=2) and ((0,0,1)=1) ) or ( ((0,0,0)=2) and ((0,0,1)=4) ) or
        ( ((0,0,0)=2) and ((0,0,1)=7) ) or ( ((0,0,0)=4) and ((0,0,1)=1) ) or
        ( ((0,0,0)=4) and ((0,0,1)=2) ) or ( ((0,0,0)=4) and ((0,0,1)=3) )
  )   and
  % There is someone else who wants the same cell
  ( ( ((-2,-2,0)=9) and ((-2,-2,1)=9)) or (((-2,-2,0)=5) and ((-2,-2,1)=9)) or
  (((-2,-2,0)=8) and ((-2,-2,1)=9) ) or (((-2,-2,0)=8) and ((-2,-2,1)=6) ) or
  (((-2,-2,0)=8) and ((-2,-2,1)=3) ) or (((-2,-2,0)=6) and ((-2,-2,1)=9) ) or
  (((-2,-2,0)=6) and ((-2,-2,1)=8) ) or (((-2,-2,0)=6) and ((-2,-2,1)=7) ) or
...
  (((0,-1,0)=2) and ((0,-1,1)=2) ) or (((0,-1,0)=5) and ((0,-1,1)=2) ) or
  (((0,-1,0)=1) and ((0,-1,1)=2) ) or (((0,-1,0)=1) and ((0,-1,1)=3) ) or
  (((0,-1,0)=1) and ((0,-1,1)=6) ) or (((0,-1,0)=3) and ((0,-1,1)=2) ) or
  (((0,-1,0)=3) and ((0,-1,1)=1) ) or (((0,-1,0)=3) and ((0,-1,1)=4) )

  )
}

% Up and Left movement
rule : 1 100 { % Collisions accounted for previously
  % We are actually moving up and left (11,51,21,24,27,41,42,43 combos):
  ( ( ((1,1,0)=1) and ((1,1,1)=1) ) or ( ((1,1,0)=5) and ((1,1,1)=1) ) or
        ( ((1,1,0)=2) and ((1,1,1)=1) ) or ( ((1,1,0)=2) and ((1,1,1)=4) ) or
        ( ((1,1,0)=2) and ((1,1,1)=7) ) or ( ((1,1,0)=4) and ((1,1,1)=1) ) or
        ( ((1,1,0)=4) and ((1,1,1)=2) ) or ( ((1,1,0)=4) and ((1,1,1)=3) )   )
}
```

FIGURE 9.12 Model definition in CD++.

Figure 9.13 displays the two state variables employed in the definition of the Cell-DEVS model (displayed side by side). The left-hand plane shows the current location and velocity of three devices. The right-hand plane describes the "desired velocity vector field" of the devices. The "desired location" for all three devices is the center of the plane, and the "desired velocity vectors" steer them to that point.

The three devices enter from the top-right corner, and they stop at the desired location. The devices enter (at times 0, 500, and 900 ms) with a velocity different from the desired velocity, and each acts in accordance with the state transitions to turn to the desired velocity. At 1.2 s, the first device enters a region with a different desired velocity. The first device reaches the target cell at 1.5 s and stops. The other devices follow the same path.

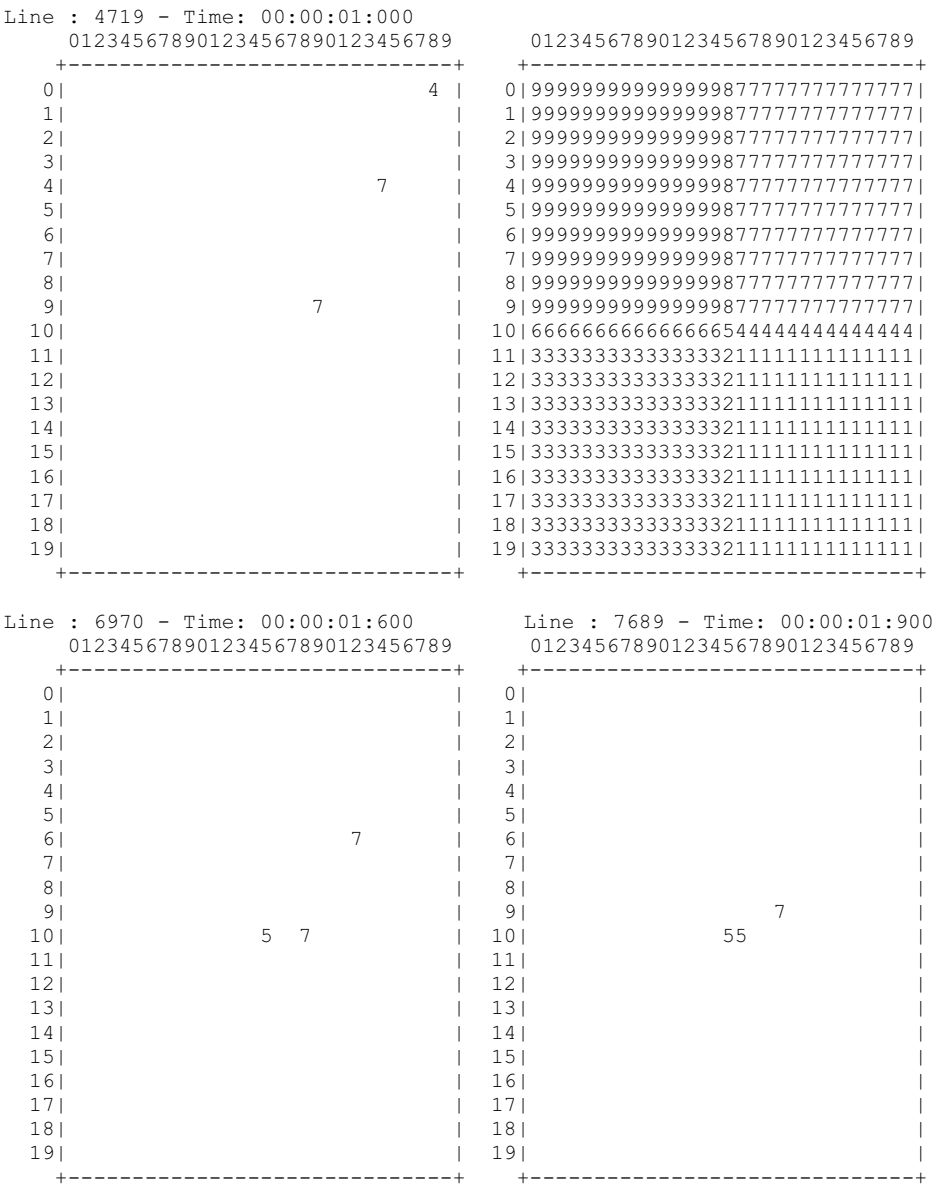


FIGURE 9.13 Three devices seeking the desired location.

Figure 9.14 displays a snapshot of a test where devices enter the plane from various locations and with different velocities (each must respond to the desired velocity in accordance with each current velocity). Initially (Figure 9.14(a)), the devices move toward the center. Then a second set of devices (with different initial velocity) enters the plane. The final step represents the congestion of devices toward the desired location.

Exercise 9.3

The radar transmitter/receiver model presented in Section 9.3 can be integrated with the seeker model we just introduced. Build a new model (called radar) that will scan a region of the cell space at a given frequency, as seen in Figure 9.15. The information given by the radar transmitter/receiver model must

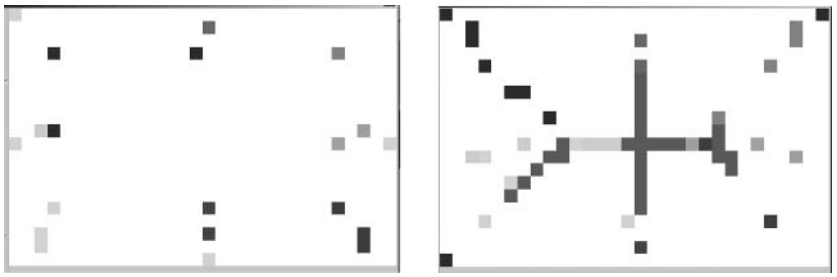


FIGURE 9.14 Seekers with collision avoidance. (a) Sashing target; (b) Target found.

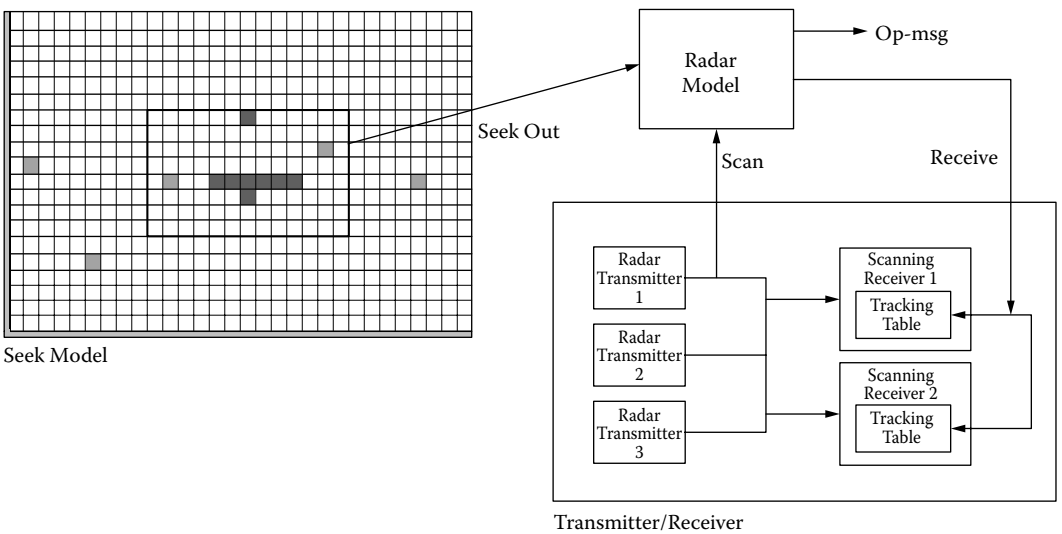


FIGURE 9.15 Integrating seek and transmitter models.

be used to start the radar model scanning activity. Upon activation, the radar will scan the field defined by the seek Cell-DEVS model and will generate two outputs: a reception signal for the transmitter/receiver and a number of operator messages, according to the values received in the field. Write the radar atomic model and the coupled model integrating the three components.

Exercise 9.4

Integrate the UAV model presented in Section 9.2 with the coupled model defined in Exercise 9.3. Replace the seeker model by the UAV model and repeat the tests.

9.5 LAND BATTLEFIELD

We present a land battlefield model using Cell-DEVS (introduced in Madhoun and Wainer [2] and found in *./Battlefield.zip*), in which two armies engage in battle. Each army is composed of a number of soldiers defending a flag. The goal of each army is to capture the enemy's flag or to defend its own, as depicted in Figure 9.16.

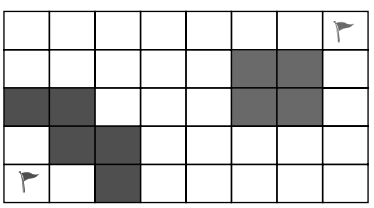


FIGURE 9.16 Possible troop allocations.

Different authors have used cellular automata to model these systems (for instance, references 5–8). The model represented here is based on some of them, and the main characteristics can be summarized as follows:

- The battlefield is two-dimensional (no airplanes or missiles).
- Each soldier can be in one of three states: *alive, injured,* or *dead.*
- The situation awareness of the soldier is limited to the neighborhood (no telecommunication equipment is used).
- If an *alive* soldier is attacked, the state changes to *injured.*
- If an *injured* soldier is attacked by an enemy soldier, he becomes *dead.*
- The soldier's ability to fight is dependent on a randomly assigned factor: fighting ability (FA). In addition, an *injured* soldier will have a lower FA than the one who is *alive.*
- Injured soldiers recover to the *alive* state if they are not surrounded by the enemy.
- Unless engaged in a fight, a soldier moves toward the enemy's flag.
- If a soldier is surrounded by the enemy, she engages in a fight. The outcome of this fight depends on the FA of the soldiers fighting.
- The flag is acquired once an enemy soldier moves to its neighborhood.

The status of the soldier is represented by a signed integer to distinguish between the two armies. One of the armies has positive values (army A) and the other has negative values (army B). Table 9.2 describes this representation.

TABLE 9.2
Battlefield Model State Values

Status	Description
2	Fighter of army A alive
1	Fighter of army A injured
0	Fighter is dead and cell is empty
–1	Fighter of army B injured
–2	Fighter of army B alive
5	Flag of army A
–5	Flag of army B

The FA of each soldier is represented by a random real number ranging from 0 to 1. Zero represents no FA at all (used for the flag and for dead soldiers), while 1 represents a very high FA. In addition, the soldier will have an effect on the enemy only if the FA is greater than 0.5. The assignment is done using a random function with uniform distribution, and it is executed at two points: at the beginning of the battle and after engaging in a fight with an enemy soldier. Table 9.3 describes the FA factor.

When two or more soldiers engage in a fight, the outcome depends on the difference between their FAs, as seen in Figure 9.17. In this case, two soldiers in army A (light gray) engage with two in army B (dark gray). Initially, the four soldiers are alive. When they engage in a fight, the B soldier in the middle dies, the FA of the surrounding soldiers is reduced, and the middle soldier in army A is injured.

Because each soldier aims to acquire the enemy's flag, he needs to know about the flag position. This information is represented as a real number having the integer part representing the flag row number and the fractional part representing the flag column number; that is, row + column/100

TABLE 9.3
Fighting Ability States

Status	Fighting Ability
2	Uniformly distributed number in the range [0.45, 1]
1	Uniformly distributed number in the range [0,0.55]
0	Fighter is dead and cell is empty 0.0
–1	Uniformly distributed number in the range [0,0.55]
–2	Uniformly distributed number in the range [0.45,1]
5	Does not engage in fights 0.0
–5	Does not engage in fights 0.0

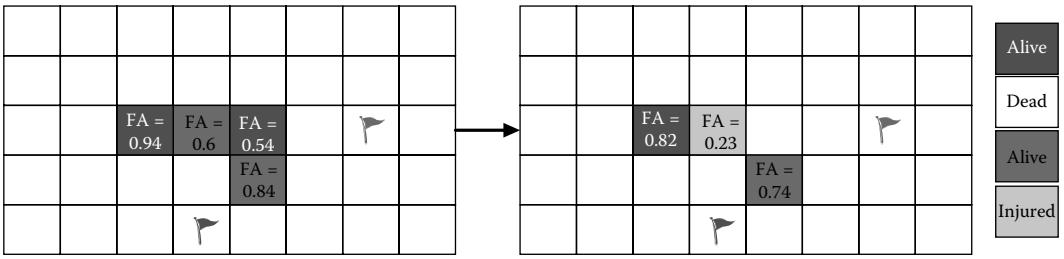


FIGURE 9.17 The effect of different fighting abilities.

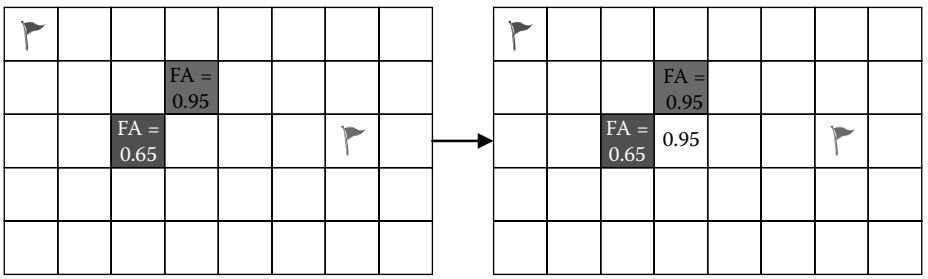


FIGURE 9.18 Free-cell move-in factor evaluation.

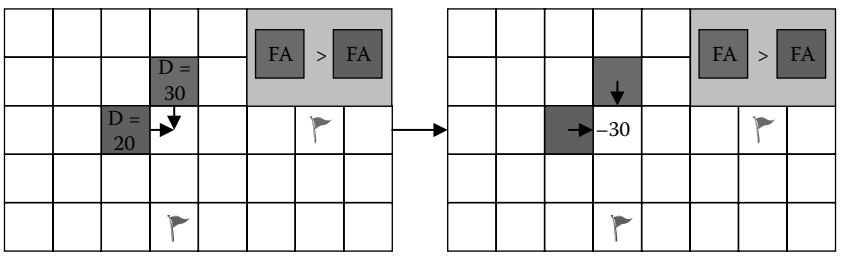


FIGURE 9.19 Free-cell move-in factor with intention.

(e.g., row = 2, column = 4 → 2.04). If a soldier is not surrounded by the enemy, she tends to move toward the enemy's flag. To do so, the soldier needs to calculate the direction of the next step in order to move closer to the target. This is done by comparing the current cell position of the soldier with the enemy's flag position.

The *free-cell move-in factor* is an integer number that is calculated for every free cell to resolve any conflict if two or more soldiers want to move to the same free cell. This factor is evaluated as the maximum FA of the soldiers surrounding the free cell. Figure 9.18 illustrates this point: the two soldiers want to move to the same cell, but the one with higher FA is the one moving.

A different implementation of the model (found in *./BattleField2.zip*) computes the free-cell move-in factor as the maximum FA of the soldiers in the neighborhood who intend to move to the cell. Only the one with the maximum FA will be allowed to move to the free cell. In this scenario, the free-cell move-in factor will be the direction of the soldier with maximum FA, with an opposite sign to indicate that the cell will be occupied by the soldier coming from that direction. Figure 9.19 illustrates this point.

In CD++, each of these behaviors was included on a different layer of a three-dimensional cell space (Figure 9.20). The layers used to implement the model are as follows:

- layer 0: soldier's status and allocation in the battlefield;
- layer 1: FA factor;
- layer 2: flag position of army B;
- layer 3: flag position of army A;
- layer 4: movement directions of each soldier; and
- layer 5: move-in factor associated with each free cell.

The model was executed with different test scenarios. The first one we present here is devoted to analyzing only the movement rules of the fighters toward the enemy's flag. Figure 9.21 shows the initial and final configurations of the army (one fighter of each army was killed in the battle; both armies eventually reached the flags).

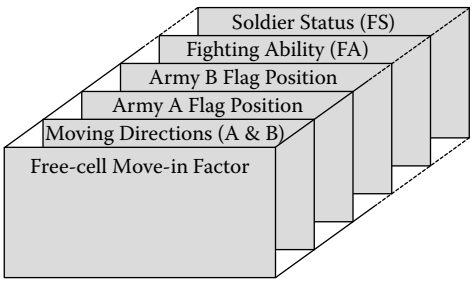


FIGURE 9.20 Cell space layers.

The battlefield model was extended using the facilities available in parallel CD++. As shown in Chapter 8 and introduced in López and Wainer [9], this includes the ability to define multiple I/O ports for each cell in the cell space or multiple state variables per cell. Because the ports connect each cell with all of its neighbors, they can be used to represent information to be transferred between cells. Instead, state variables are local to the cell and can be used to represent internal data that cannot be referenced from outside the cell. Both features were used to implement the battlefield model, dispensing with the need to define extra layers of cells to represent each piece of information. Instead, the following input/output ports were used:

- *fs* represents the soldier status (i.e., alive, injured, dead).
- *fa* represents the FA of the soldier.
- *enemy_flag* is location of the enemy flag, using the same format explained earlier.
- *direction* represents the direction of the next move of the soldier.

Different rules were defined to mimic the behavior of soldiers in a battlefield, including:

- fighting rules: behavior of soldiers engaged in a fight;
- flags-under-attack rules: behavior of the flag when attacked by an enemy soldier;
- flags-not-attacked rules: behavior of the flag when not attacked;
- movement-direction rules: direction of the next step for each soldier to come closer to the enemy flag; and
- movement: behavior of the soldiers when moving in the battlefield.

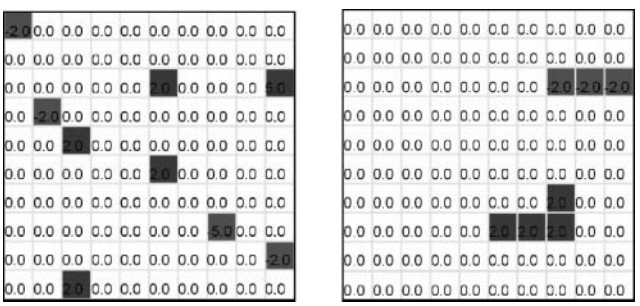


FIGURE 9.21 Testing movement rules.

```
#BeginMacro(fight_rule_1)  (
if( ((-1,0)~fs=-1 or (-1,0)~fs=-2) and (-1,0)~fa>0.5 and ((-1,0)~fa>(0,0)~fa) , -1, 0) +
if( ((0,-1)~fs=-1 or (0,-1)~fs=-2) and (0,-1)~fa>0.5 and ((0,-1)~fa>(0,0)~fa), -1, 0) +
if( ((0,1)~fs=-1 or (0,1)~fs=-2) and (0,1)~fa>0.5 and ((0,1)~fa>(0,0)~fa), -1, 0) +
if( ((1,0)~fs=-1 or (1,0)~fs=-2) and (1,0)~fa>0.5 and ((1,0)~fa>(0,0)~fa) , -1, 0)
)
#EndMacro
```

FIGURE 9.22 Fighting rules macros.

```
%==============Army A fighter is surrounded by an enemy fighter

rule : { ~fs:= 1 ; ~fa:= uniform(0,0.55) ; ~direction := 0 ;  } 100
       { (0,0)~fs=1 and (statecount(-1,~fs)+statecount(-2,~fs))>0 and
       (#macro(fight_rule_1))=0  }
rule : { ~fs:= 0 ; ~fa:= 0 ; ~direction := 0 ; ~enemy_flag := -1 ; }  100
       { (0,0)~fs=1 and (statecount(-1,~fs)+statecount(-2,~fs))>0 and
       (#macro(fight_rule_1))<0  }
rule : { ~fs:= 2 ; ~fa:= uniform(0.45,0.99) ; ~direction := 0; } 100
       { (0,0)~fs=2 and (statecount(-1,~fs)+statecount(-2,~fs))>0 and
       (#macro(fight_rule_1))=0  }
rule : { ~fs:= 1 ; ~fa:= uniform(0,0.55) ; ~direction := 0 ;  } 100
       { (0,0)~fs=2 and (statecount(-1,~fs)+statecount(-2,~fs))>0 and
       (#macro(fight_rule_1))=-1 }
rule : { ~fs:= 0 ; ~fa:= 0 ; ~direction := 0 ; ~enemy_flag := -1; }  100
       { (0,0)~fs=2 and (statecount(-1,~fs)+statecount(-2,~fs))>0 and
       (#macro(fight_rule_1))<-1 }
```

FIGURE 9.23 Fighting rules.

As an example of these rules, we show the implementation of the fighting rules. The macro *fight_rule_1* in Figure 9.22 checks whether the soldier (from army A) is in the neighborhood of an enemy soldier (from army B). Then it checks whether the soldier has a higher FA and, in that case, adds (–1) to the overall value of the macro for each such soldier.

The number generated by *fight_rule_1* is used in the main body of the rule (presented in Figure 9.23) to evaluate the following conditions:

- If a soldier in army A is injured (*fs* = 1) and is surrounded by enemy soldiers whose FAs are lower than hers, the soldier's FA factor is reduced.
- If a soldier in army A is injured (*fs* = 1) and is surrounded by enemy soldiers whose FAs are higher than hers, the soldier dies (FA = 0).
- If a soldier in army A is alive (*FS* = 2) and is surrounded by enemy soldiers whose FAs are lower, she remains alive and is assigned a new FA factor.
- If a soldier in army A is alive (*FS* = 2) and is surrounded by enemy soldiers, but only one of them has a higher FA, the soldier is injured and assigned a new FA factor.
- If a soldier in army A is alive (*FS* = 2) and surrounded by enemy soldiers and more than one of them have a higher FA, the soldier dies (FA = 0).

The same rules are used for B soldiers when surrounded by army A soldiers. Figures 9.24, 9.25, and 9.26 show different scenarios. In Figure 9.24, only the movement rules are analyzed, and we can see how the soldiers of army A move toward and acquire the B flag. In Figure 9.25, we show a scenario where the fighting rules are used when soldiers of both armies engage in a fight. In Figure 9.26, all the rules are activated to test the overall behavior of the model, obtaining a similar result to the one presented in Figure 9.21.

Some extra features were added to the model to improve its behavior using parallel CD++, including:

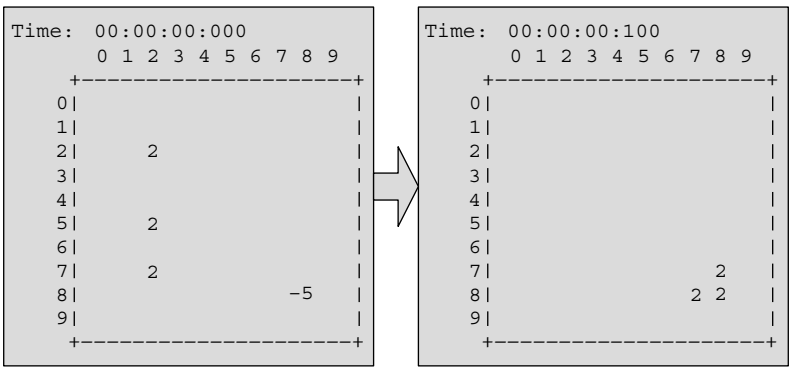


FIGURE 9.24 Testing movement rules.

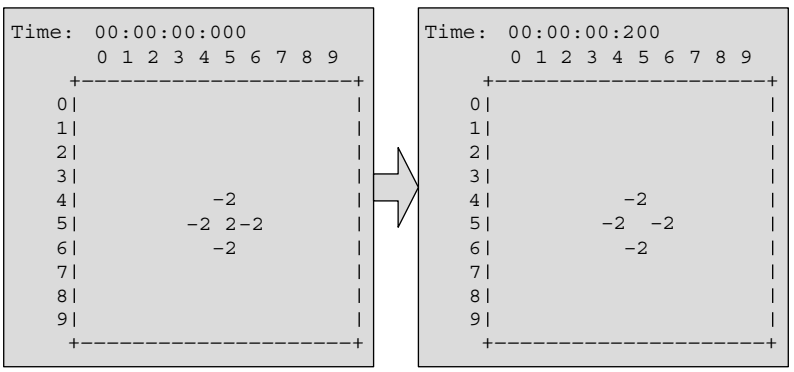


FIGURE 9.25 Testing fighting rules.

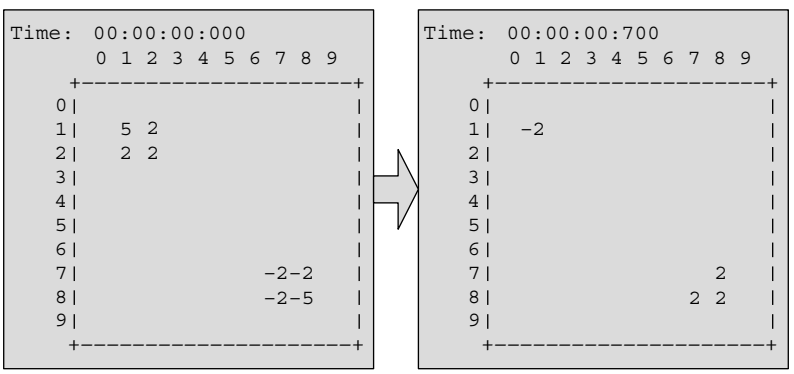


FIGURE 9.26 Overall test of the model.

- Extending the situation awareness of the soldier (neighborhood) to include the eight surrounding cells; hence, the soldier is able to attack and move diagonally as well as horizontally or vertically (Figure 9.27);
- Obstacle avoidance: the soldiers are able to avoid obstacles (FS = 50) while moving toward the enemy's flag, as seen in Figure 9.28;

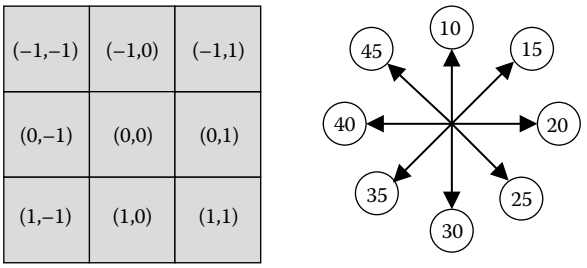


FIGURE 9.27 Extending the soldier's neighborhood to Moore's neighborhood.

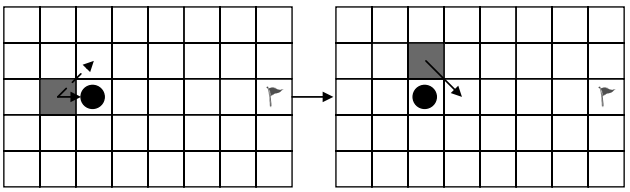


FIGURE 9.28 Obstacle avoidance example.

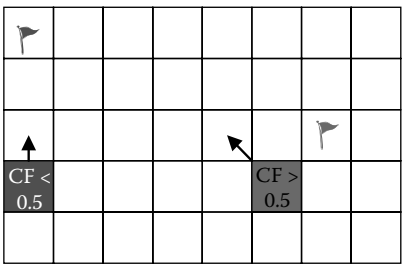


FIGURE 9.29 Effect of the courage factor fighting ability on the soldier's behavior.

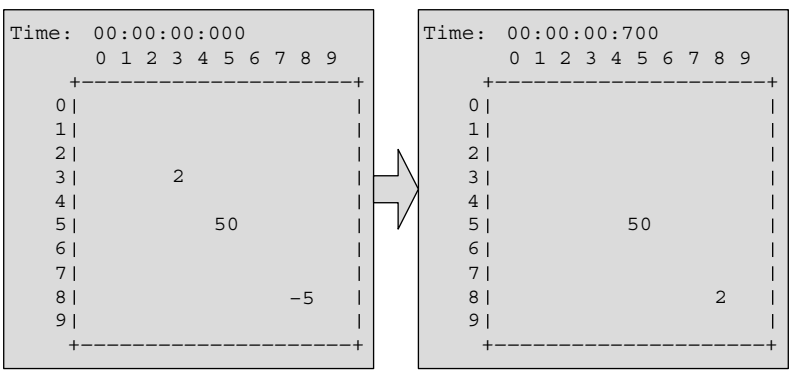


FIGURE 9.30 Testing the obstacle avoidance feature.

- Courage factor (CF): this factor is used to simulate that not all the soldiers on a battlefield will have the same courage to fight the enemy. Hence, this factor will determine if the soldier is going to attack the enemy or retreat toward his own base/flag (Figure 9.29).

The results of these tests are shown in Figures 9.30 and 9.31.

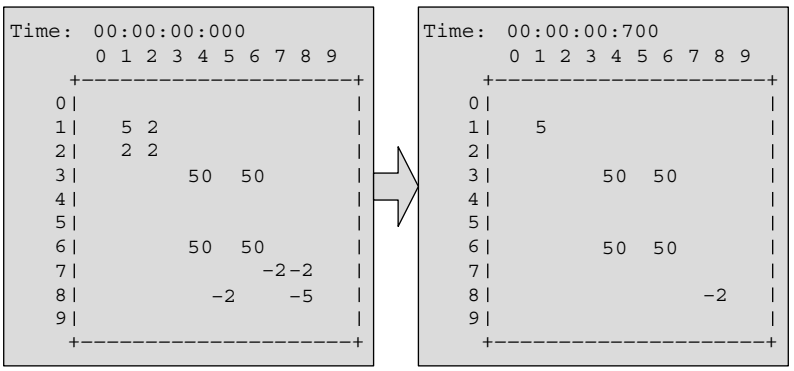


FIGURE 9.31 Testing the overall behavior of the model.

9.6 EVACUATION PROCESSES

The simulation of evacuation processes was originally applied to buildings or the aviation industry and ship evacuation [10]. The model presented here (originally introduced in Ameghino and Wainer [11] and found in *./shipevacuation_2.zip*) represents people moving along a ship's deck, trying to get out through an exit door. The goal is to understand where the bottlenecks can occur and which solutions are effective in preventing congestion.

The basic idea was to simulate the behavior and movement of every single person involved in the evacuation process. A Cell-DEVS model was chosen with a minimum set of rules to characterize a person's behavior:

- People try to move toward the closest exit.
- People move at different speeds.
- If the way is blocked, people can decide to move away and look for another way.

Table 9.4 describes the encoding of the cell state, in which each position of the state is represented by a natural number in which each digit represents a different state. We used two planes: one to represent the floor plan and people moving, and the other to include information on orientation to the closest emergency exit. We assigned a potential distance to an exit to every cell of this layer. The persons move in the room, trying to minimize the potential of the cell in which they are (see Figure 9.32).

Figure 9.33 shows the main rules of an evacuation model. We have two different planes to separate the rules that govern the people moving among walls or aisles from the orientation guide to an exit. The rules in Figure 9.33 define what path a person should follow using the orientation plane. The basic idea is to take the direction decreasing the potential of a cell, building a path following

TABLE 9.4
Encoding of States for the Evacuation Model

Digit	Meaning
6	Next movement direction. 1: W; 2: SW; 3: S; 4: SE; 5: E; 6: NE; 7: N; 8: NW
5	Speed (cells per second: one to five)
4	Last movement direction can vary from 1 to 8 (as digit #6)
3	Emotional state: the higher this value is, the lower is the probability that a person panics
2	Number of movement that increases the potential of a cell
1	Panic level represents the number of cells that a person will move, increasing the cell potential

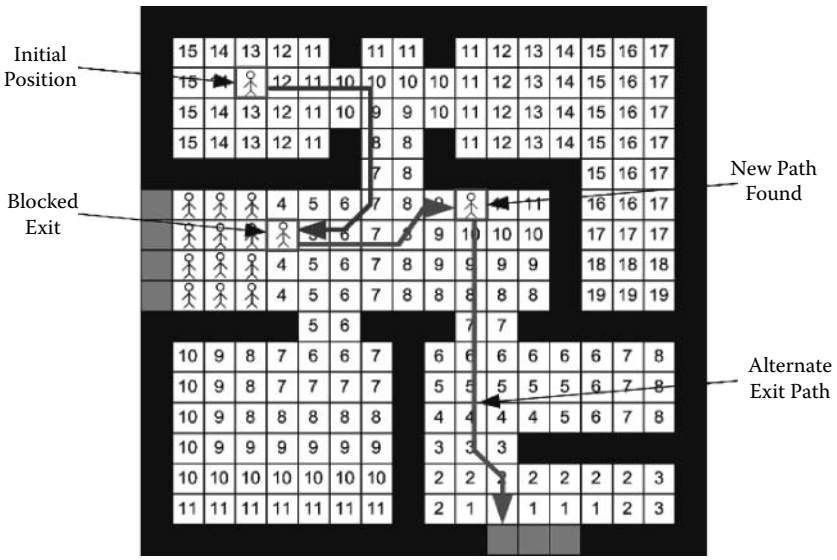
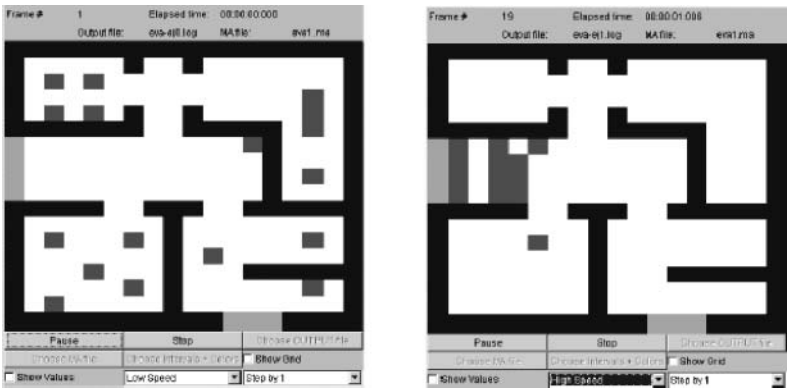


FIGURE 9.32 Orientation layer: potential value (people using this try to use a path decreasing the potential) [12]; see also [13].

```
[deck]
dim : (18,18,2)        delay : inertial       border : wrapped
localtransition : EvaRule
neighbors : (-1,-1,0) (-1,0,0) (-1,1,0) (0,-1,0) (0,0,0) (0,1,0)
(1,-1,0) (1,0,0) (1,1,0) (-1,-1,1) (-1,0,1) (-1,1,1) (0,-1,1) ...

[EvaRule]
% Rules to govern people movement
rule : { trunc((0,0,0)/10)*10+1}  {1000 / remainder(trunc((0,0,0) /10),10) }
       { (
         (0,0,0)>0 AND remainder(trunc((0,0,0)/1),10)=0 AND
remainder(trunc((0,0,0)/100000),10)=0
             AND ((0,-1,0)=0 OR (0,-1,0)=-2) AND cellPos(2)=0
          )
        AND
        (
        ( (0,-1,1) <= (1,-1,1)  OR (1,-1,0)>0  OR (1,-1,0)=-1  OR (randint(5)=0) ) AND
        ( (0,-1,1) <= (1,0,1)   OR (1,0,0)>0   OR (1,0,0)=-1   OR (randint(5)=0) ) AND
        ( (0,-1,1) <= (1,1,1)   OR (1,1,0)>0   OR (1,1,0)=-1   OR (randint(5)=0) ) AND
        ( (0,-1,1) <= (0,1,1)   OR (0,1,0)>0   OR (0,1,0)=-1   OR (randint(5)=0) ) AND
        ( (0,-1,1) <= (-1,1,1)  OR (-1,1,0)>0  OR (-1,1,0)=-1  OR (randint(5)=0) ) AND
        ( (0,-1,1) <= (-1,0,1)  OR (-1,0,0)>0  OR (-1,0,0)=-1  OR (randint(5)=0) ) AND
        ( (0,-1,1) <= (-1,-1,1) OR (-1,-1,0)>0 OR (-1,-1,0)=-1 OR (randint(5)=0 )
        )
        )}
...
```

FIGURE 9.33 Specification of evacuation model.

those whose neighbor value is lower, as shown in Figure 9.32. We use eight rules to control the people's movement (one per direction). In all cases, the rule analyzes the eight near neighbors to understand what direction the person should take; if all the eight near neighbors have the same value, we resolve it using a value at random. A person moves to decrease the movement potential by decreasing the distance to the exit. If there is no available move decreasing the potential, a person will try to move to a neighboring cell that has the same potential. In the worst-case scenario, the person will move further away in an attempt to find another route, as seen in Figure 9.32.



(a) (b)

FIGURE 9.34 (a) People seeking an exit; (b) after 15 s, people found the exit.

Figure 9.34 shows the simulation results for this model. The gray cells represent people who want to escape using the exit doors. The black cells represent walls. Note that the leftmost part in the figure shows people waiting in the exit doorway, due to congestion.

The next example describes the movement of people in a metro station. When a train arrives at the station, it is often the case that a person in the car wants to get out but finds individuals on the platform. In this example we focus on the problems derived from this situation. The model is restricted to only two groups of individuals and only one car with two doors. People can either get into or out of the train. We use a Cell-DEVS model to represent the station and the people moving on the platform and a DEVS generator to model people arriving at the station. We defined three classes of individuals: those who want to get out of the train and go to the platform exit, those who want to get into the train using door A, and those who want to get into the train using door B.

Figure 9.35 shows the implementation of the model (found in *./metro.zip*) using CD++ [12]. The cell space (*arriving*) and the DEVS component that generates individuals (*PeopleGenerator*) are defined at the beginning of the model. The generator uses an exponential distribution function. The rules of the Cell-DEVS model represent the movement of the people using a combination of the direction (second digit; 1: S; 2: E; 3: N; 4: W) and the door to be reached (first digit; 1: A; 2: B). The rules determine the behavior of each person considering these two values and the presence of individuals in the neighboring cells. Hence, a person moves toward an adjacent cell based on the group to which he or she belongs, its current location, direction, and state of the nearby cells.

We use an extended von Neumann neighborhood (a 5 × 5 rhombus centered on the origin cell). The *arriving* Cell-DEVS model has three input ports: *inputPeople* represents individuals arriving, and they are generated by the *peopleGenerator* model (which uses an exponential distribution of mean of 4 and initial value of 1, which represents the arrival of a new person). People arriving will be directed to the cell (15,9). Door A is in (9,0) and door B is in (19,0). The model's *rules* define how people move in different directions (including collision detection). The *createPeople* rule is executed every time a new input is created by the *PeopleGenerator* model; the initial direction is created at random, and the person arrives in cell (15, 9) in the model. The *DoorHandler* rules are activated when people arrive at the doors (the value 77 is used to define an open door and 99 defines closed doors).

Figure 9.36 shows in detail the conflicts between different individuals trying to get into the train. People are represented by dark gray cells. In this figure, the black cells represent individuals leaving the train (with high priority). The light gray cell on the left is the train's door (A), and the remaining gray cells represent people trying to get into the train. We can see the interactions between persons leaving and entering the car.

```
[top]
components : arriving PeopleGenerator@Generator
in :   doorA    doorB
link : out@PeopleGenerator inputPeople@arriving
link : doorA    inputDoorA@arriving
link : doorB    inputDoorB@arriving

[arriving]
type : cell    dim : (30,10)   delay : inertial
defaultDelayTime : 500         border : nowrapped
neighbors  :  (0,-2)(-1,-1)(0,-1)(1,-1)(-2,0)(-1,0)(0,0)(1,0)(2,0)(-1,1)(0,1)(1,1)(0,2)

in : inputPeople  inputDoorA   inputDoorB
link : inputPeople in@arriving(15,9)
link : inputDoorA  in@arriving(9,0)
link : inputDoorB  in@arriving(19,0)
portInTransition : in@arriving(15,9) createPeople
portInTransition : in@arriving(9,0) doorController
portInTransition : in@arriving(19,0) doorController
. . .
[rules]
rule : 0  500 { (0,0)=24 and (0,1)=0 }
rule : 0  500 { (0,0)=23 and (-1,0)=0 and (-1,-1)!=24 }
rule : 0  500 { (0,0)=22 and (1,0)=0 and (1,-1)!=24 and (2,0)!=23 }
. . .
rule : 6  700 { ( (0,0)=1 ) }
rule : 7  700 { ( (0,0)=2 ) }
rule : 8  700 { ( (0,0)=3 ) }
. . .
rule : 19 500 { ( (0,0)=0 ) and ( (0,-1)=14 ) and (1,0)!=23 and (-1,0)!=22 }
rule : 24 10 { (0,0)=0 and (0,-1)=77 }

[createPeople]
rule : { ( randint(1)*10 ) + 1 } 10 { (0,0)=0 }
rule : {(0,0)} 10 {t}

[PeopleGenerator]
distribution : exponential
mean : 4       initial : 1      increment : 0

[doorController]
rule : 77 1 { (0,0)!=77 and (0,0)!=99 }
rule : 99 1 { (0,0)=77 }
rule : 0  1 { t }
```

FIGURE 9.35 Specification of the metro station.

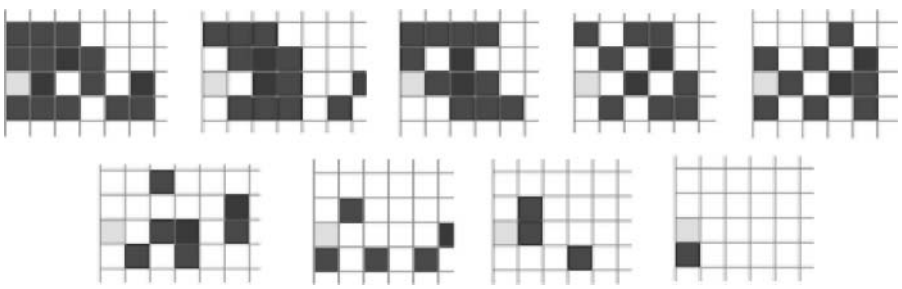


FIGURE 9.36 Execution results of people getting in and out using door A.

Figure 9.37 shows individuals arriving at the station and waiting for the train. Two light gray cells located on the right side of each slide represent the platform entrance. The gray cells represent people who want to get into the train using door A, placed in the upper part of the grid. The dark gray cells represent people who want to get into the train using door B, placed in the lower part of the grid. The rightmost slide in the figure shows two groups of individuals standing along the border of the platform waiting for the train doors to open.

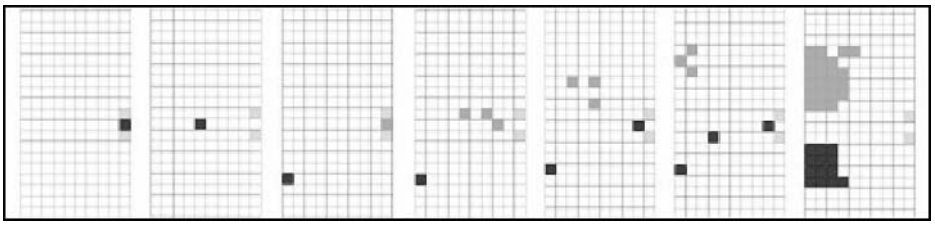


FIGURE 9.37 Execution results of metro station model.

The last model presented in this section, originally introduced in Reynolds [4], is based on the flocking behavior of birds during migration. The term *flocking* has been used in the field of evacuation studies because it also permits describing the behavior of people who move together under emergency situations, thus allowing description of the behavior of individuals following a leader and related individuals moving together. As discussed in Reynolds [4], the motion of a flock seems fluid because of the interaction between individuals. To do so, we focus on the behavior of an individual (or at least that portion of the individual's behavior that allows it to participate in the flock), based on the following behavior rules defined in Reynolds [4]:

- Collision Avoidance: avoids collisions with nearby flock mates.
- Velocity Matching: attempt to match velocity with nearby flock mates.
- Flock Centering: attempt to stay close to nearby flock mates.
- Individuals fly in certain direction at a certain speed.
- The field of vision of the individual is 300°, but only they have good sight forward (in a zone from 10° to 15°)

We defined a Cell-DEVS version of this model in Ameghino and Wainer [11] in which we changed the original simulation parameters. Each cell now represents a space of 4 m^2, which can fit a medium-size individual (~18–23 cm.). A second of simulation time represents a second of real time (hence, an individual who moves in the simulation with a speed of seven cells per second represents an individual flying at 50 km/h). The cell state encoding uses a natural number to represent both the direction of the individual (1: NW; 2: N; 3: NE; 4: W; 6: E; 7: SW; 8: S; 9: SE) and the speed. Individuals are represented using numbers greater than 100,000; the field of vision is represented using numbers below 10,000. For example, the cell value 100014 represents an individual moving in direction W (unit value equal to 4). The speed is computed by taking the second digit in this representation (in this case, 1), multiplying by 10, and adding 90 to the total (in this case, 100 ms); this value is used in the cell's delay. The first digit shows the original direction, the second shows whether the cell is used (1) or not (0), and the last two digits in this group are used to count the number of individuals in the neighborhood (e.g., if a vision cell is used by a neighboring bird to the left, the value will be 4100). In order to avoid collision, when two or more individuals want to move to the same place, they change direction using a random variable. Figure 9.38 describes the model specification as found in *./boids.zip*.

The *fly rule* implements the conditions previously explained:

- There must be an individual on the cell ((0,0) > 100,000).
- We check whether the vision cell is in the right direction (e.g., the vision cell to the S must have that direction ((–1,0) = 8100).
- In that case, we compute the delay for the cell.
- The cell state will change according to the current direction of the flock.

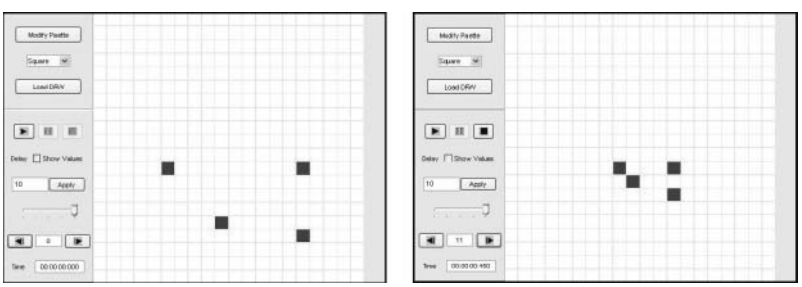
```
[boids]
type : cell        dim : (20,20)     delay :  transport
border : wrapped
neighbors : (-2,-2) (-2,-1) (-2,0) (-2,1) (-2,2) (-1,-2) (-1,-1) (-1,0) (-1,1) (-1,2) (0,-2)
(0,-1) (0,0) (0,1) (0,2) (1,-2) (1,-1) (1,0) (1,1) (1,2) (2,-2) (2,-1) (2,0) (2,1) (2,2)
...
[fly-rule]
     rule : { 1 + if(((-2,-2)>100000),1,0) + if(((-2,-1)>100000),1,0) +
if(((-2,0)>100000),1,0)+if(((-2,1)>100000),1,0)+if(((-2,2)>100000),1,0)+
     if(((-1,-2)>100000),1,0)+
if(((-1,-1)>100000),1,0)+if(((-1,0)>100000),1,0)+if(((-1,1)>100000),1,0)+
if(((-1,2)>100000),1,0)+
if(((0,-2)>100000),1,0)+if(((0,-1)>100000),1,0)+if(((0,1)>100000),1,0)+
if(((0,2)>100000),1,0)+if(((1,-2)>100000),1,0)+if(((1,-
       1)>100000),1,0)+if(((1,0)>100000),1,0)+if(((1,1)>100000),1,0)+
if(((1,2)>100000),1,0)+if(((2,-2)>100000),1,0)+
if(((2,-1)>100000),1,0)+if(((2,0)>100000),1,0)+if(((2,1)>100000),1,0)+
      if(((2,2)>100000),1,0)}{90+trunc((0,0)/10-10000)*10}
{(0,0)>100000 AND ((-1,-1)=9100 OR (-1,0)=8100 OR (-1,1)=7100 OR (0,-1)=6100 OR
       (0,1)=4100 OR (1,-1)=3100 OR (1,0)=2100 OR AND (1,1)=1100)}
...
```

FIGURE 9.38 Specification of the flocking model.

In addition to the fly rule, the following conditions are checked in the model:

- If the cell is taken,
  - if there is a neighboring individual or taken cell, invert the current direction;
  - if there is an individual with distance 1, modify the speed and direction to follow it;
  - if there is a potential collision with a taken cell, invert the current direction; and
  - if there are no individuals in the neighborhood, keep direction and speed.
- If the cell is not taken:
  - if an individual is coming, take the cell and record the direction of the individual;
  - otherwise, free the cell.

Figure 9.39 shows some execution results for this model. The rules introduced represent the basic flying behavior of birds. Individuals fly freely, but when an individual detects others, it tries to follow them. To do so, the individual changes direction and speed to avoid collision or losing the flock. Different time conditions can be used to simulate the change of an individual's velocity. The example presented in Figure 9.39 shows the individuals flying and how individuals find each other.



(a) (b)

FIGURE 9.39 Flocking behavior: (a) four individuals flying isolated; (b) individuals flying together.

9.7 SUMMARY

Defense simulations are some of the most advanced applications in simulation. We can see current advances in this field in the numerous annual worldwide conferences in the field. Likewise, diverse journals deal with this complex topic (including the *SCS Journal of Defense Modeling and Simulation,* the *Training and Simulation Journal,* the *C4ISR Journal,* and numerous others), which shows the importance of simulation tools and methodologies in this field.

In this chapter we have introduced some examples in using DEVS and Cell-DEVS in the field of defense and emergency planning. We first presented a simple collision detection model using Cell-DEVS. Then we introduced a DEVS model for the synchronization of radar transmitters and receivers. We then presented a Cell-DEVS model of the behavior of a target seeker and a model of land battlefields. Finally, we showed basic evacuation models, including crowding in a metro, evacuation of a ship, and flocking behavior that can be used for people being evacuated. Various other examples can be found in our model repository, including a Cell-DEVS model of radar (*./radar.zip*), a model on the load monitoring system for a CC-130 aircraft (*./Hercules.zip*), and a model of a flight deck simulation (*./HelicopterFlightDeckSimulation.zip*) to study the operations on deck.

REFERENCES

1. Palmore, J. 1997. Mini-symposium/workshop report. Warfare analysis and complexity. Military Operations Research Society. September 15–17, 1997. http://www.mors.org, JHU/APL. Laurel, MD.
2. Madhoun, R., and G. Wainer. 2005. Developing defense applications using DEVS/cell-DEVS. *Journal of Defense Modeling and Simulation* 2:121–143.
3. MacSween, P., and G. Wainer. 2005. On the construction of complex models using reusable components. *2004 Spring Simulation Interoperability Workshop*, Arlington, VA.
4. Reynolds, C. W. 1999. Steering behaviors for autonomous characters. *Proceedings of Game Developers Conference,* San Jose, CA.
5. Woodcock, A. E. R., L. Cobb, and J. Dockery. 1988. CA: A new method for battlefield simulation. *Signal* 42:41–50.
6. Gore, J. 1996. Chaos, complexity and the military. Technical report 96-E-61, National Defense University, National War College. Military Strategy and Operation Seminar D.
7. Ilachinski, A. 2004. *Artificial war: Multiagent-based simulation of combat.* Singapore: World Scientific Press.
8. Ilachinski, A. 2000. Irreducible semi-autonomous adaptive combat (ISAAC)—An artificial life approach to land combat. *Military Operation Research* 5:29–46.
9. López, A., and G. Wainer. 2004. Improved cell-DEVS model definition in CD++. In *ACRI 2004,* LNCS 3305, ed. P. M. A. Sloot, B. Chopard, and A. G. Hoekstra. New York: Springer–Verlag.
10. Kim, H., D. Lee, J. H. Park, J. G. Lee, B. J. Park, and S. H. Lee. 2001. Establishing the methodologies for human evacuation simulation in marine accidents. *Proceedings of 29th Conference on Computers and Industrial Engineering,* Montréal, QC, Canada.
11. Ameghino, J., and G. Wainer. 2004. Application of the cell-DEVS formalism for modeling cell spaces. In *Proceedings of Artificial Intelligence, Simulation and Planning,* Jeju Island, Korea, LNCS 3397.
12. Ameghino, J., E. Glinsky, and G. Wainer. 2003. Applying cell-DEVS models of complex systems. In *Proceedings of Summer Computer Simulation Conference,* Montreal, QC, Canada.
13. Brunstein, M., and J. Ameghino. 2003. Modeling evacuation processes using Cell-DEVS. Internal Report. Universidad de Buenos Aires, Computer Science Department.

10 Models in Architecture and Construction

10.1 INTRODUCTION

Simulation has been used for different kinds of applications in architecture and construction. One of the main uses is planning and resource allocation during the construction process because workspace conflicts can delay construction activities, reduce productivity, or cause accidents. Many existing tools are available to help construction managers— for instance, MicroCYCLONE [1], VitaScope [2], and Simphony [3]. Other applications include the management of heritage building, training of craftsmen, and better use of heating/air conditioning to reduce CO_2 emissions.

In this chapter we will show how to use DEVS and Cell-DEVS to create basic models with application to this area. Our first example is focused on analyzing a sand pile model. We then show a model of the redecking of the Cartier Bridge in Montreal, in which we analyze spatial issues in the construction site. Finally, we show a model of evacuation in buildings that can be applied to analyze better location for emergency exits in new buildings.

10.2 A SAND PILE MODEL

Malamud and Turcotte [4] presented a cellular model that can be applied for modeling landslides. This kind of sand pile model was originally created to analyze major earthquake phenomena because they often cause landslides [5]. But this kind of sand pile model can be also used for varied applications in construction (where different materials need to accumulate), as shown in Pla-Castells, García, and Martínez [6].

Saadawi and Wainer [7] introduced a Cell-DEVS model which represents a pile of sand on a table area organized as a grid. The model is initially loaded with a random number of particles, and sand particles are added to the pile continuously at the middle cell (as done in construction sites). Whenever it contains four or more sand particles, each cell redistributes its content to the four non-diagonally neighboring cells. In the case of four particles, the cell would be emptied after redistribution. For any number above four particles, the cell would distribute only four sand particles to its neighbors and keep the rest. When a cell reaches capacity, the redistribution operation starts. This operation can trigger more distributions among neighboring cells, which in turn can do the same for their neighbors. In the model, this would represent avalanches, whose severity can be measured either by the number of cells participating in a redistribution operation or by the number of particles lost from border cells [4].

Figure 10.1 shows the definition of this model (found in *./sandpile.zip*). We use a grid of 10 × 10 cells, with a von Neumann neighborhood. A non-negative value represents the number of sand particles in the cell. We use inertial delays because some rules that accumulate sand particles into a cell (for instance, rules 2 and 3) need to be executed only once on each redistribution (executing these rules more than once would increase a cell's value unnecessarily). The use of an inertial delay solves this problem: if a cell is notified several times in one time step due to changing neighbors, its value is evaluated and increased only once for all notifications because they all fall in the same time step. If, instead, we use transport delays, the evaluation of a cell's value would happen sequentially without preempting the previous value, thus accumulating a wrong value.

```
[top]
components : sandpile particleGenerator@Generator
link : out@particleGenerator in@sandpile
out :  out
link : out@particleGenerator out

[sandpile]
type : cell
dim : (10, 10)
delay : inertial       border : nowrapped
neighbors : (0,1)   (1,0)  (0,-1) (-1,0)  (0,0)
in : in
link : in in@sandpile(5,5)
localtransition : sandpile-rule
portInTransition : in@sandpile(5,5) NewParticle-rule

[sandpile-rule]
rule : { statecount(4)+statecount(5)+ statecount(6)+statecount(7)- 1 } 100 { (0,0) = 4}
rule : { statecount(4)+ statecount(5)+ statecount(6)+statecount(7)+(0,0)} 100 {(0,0)< 4}
rule : { (0,0)+statecount(4)+statecount(5)+ statecount(6)+statecount(7)-1-4} 100 { (0,0)
        > 4 }
rule : {(0,0)} 100 { t }

[NewParticle-rule]
rule : { statecount(4)+ statecount(5)+ statecount(6)+statecount(7)+(0,0) + 1} 2 {(0,0)<4}
rule : { statecount(4)+ statecount(5)+ statecount(6)+statecount(7)+(0,0)} 2 {(0,0)>=4}
rule : {(0,0) + 1 } 2 {t}

[particleGenerator]
distribution : exponential
mean : 2       initial : 1    increment : 0
```

FIGURE 10.1 CD++ tool model definition file.

The preceding Cell-DEVS model only notifies neighbors at some predefined threshold states (all intermediate states that are not important to neighboring cells are not notified, e.g., when a cell changes its value from 0 to 1, 2, or 3). When changes do not involve redistribution to other cells, we do not notify neighbors about the change. This also enhances execution speed.

The first rule shown in Figure 10.1 states that a cell with exactly four particles takes a new value, which is the sum of those neighboring cells that are distributing (i.e., the neighbors with four or more particles) minus one (because the cell itself is being counted by the *statecount(4)* statement). We have only four neighbors, so the new value assigned will be between zero (in case no neighboring cell is giving any particles) and four (in case the cell has distributed its particles to neighbors and taken one particle from each neighbor with four or more particles). This scenario is shown in Figure 10.2(a), where the cell distributes to four neighboring cells with zero particles.

The second rule in the model states that if the cell has less than four particles, it will not redistribute to its neighbors. The cell will receive only one particle from each overflowing neighbor that has four or more particles and will add the particles received to its current value. The following rule states that if the cell contains more than four particles, then it will redistribute only four particles

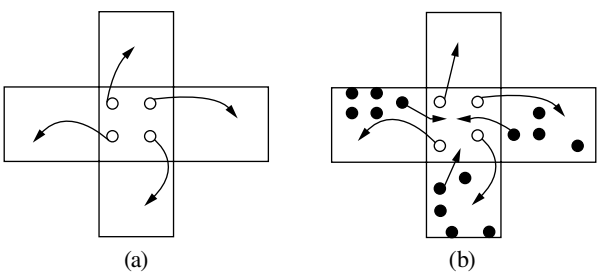


FIGURE 10.2 (a) (0,0): rule 1; (b) (0,0) and (1,0): rule 1; (0,1): rule 2; (–1,0) and (0,–1): rule 3.

to the neighbors and will retain the rest. We then add any particles received from neighbors with overflow. We subtract four to count for four particles distributed to neighbors.

Figure 10.2(b) shows the case of a cell distributing to neighbors; three of them have four or more particles. Hence, it will get one particle from each overflowing cell (with four or more particles in it).

The *NewParticle* rule in the model is used for new sand particles generated by the DEVS generator model. When the *particleGenerator* model creates a new particle, it is transmitted to the *sandpile* Cell-DEVS model (through the *in* input port). The particle is transmitted to the cell (5,5), which will activate the *NewParticle* rule upon reception of a new particle. When this happens, the rules check whether the cell's contents are fewer than four particles; in this case, the cell increases its contents by one particle, adding to this any particles coming from overflowing neighbors. Likewise, if the cell's contents are four or more particles, the cell will add one to its contents, plus any particles from overflowing neighbors. We subtract one from the final result to count for the self state.

Figures 10.3–10.6 show the simulation results for this model. The first test shows that the model handles distribution while conserving the number of sand particles. In the example presented in Figure 10.3, we initialized some cells with four or five sand particles, keeping the others empty, while keeping the sand generator disabled. We can see that all cells with an initial value of four distributed their contents to the neighboring cells (rule 1) and now contain zero particles. Only the cells that initially contained four or more particles and a neighbor with four or more particles (distributing cells) get a particle from each distributing neighbor (for instance, cells (1,2) and (1,3)). Cells like (2,6) get a particle from each distributing neighbor (in this example, two). In addition, all cells with fewer than four particles (rule 2) obtained one particle from each distributing neighbor. The total number of sand particles on the grid remained unchanged before and after redistributions (21 particles).

In Figure 10.4, we show the results of a test in which the generator is also disabled because we want to test rule 1 (for cells containing four particles) and rule 3 (for cells containing more than four particles). In this test, cell (1,3) takes the value of five particles along with other cells containing some other values. We see that all cells with four or more particles are distributing. Cell (1,3) has distributed four particles and kept one, plus four particles from distributing neighbors (rule 3). In the next time step, cell (1,3) redistributes all its four particles and stays empty (rule 1). The model starts with 42 particles and ends with the same number.

Figure 10.5 presents 11 cells filled with four or more particles. The cells distribute their content to neighbors until a steady state is reached. In this figure, the first three time steps are shown along with the final step in model execution. We begin and finish the test with 56 particles on the grid.

Finally, the example presented in Figure 10.6 shows the execution of the complete model, which uses the generator of sand particles in discrete time intervals and delivers them to the *SandPile* Cell-DEVS model. These time intervals for arriving sand particles are simulated by an exponential

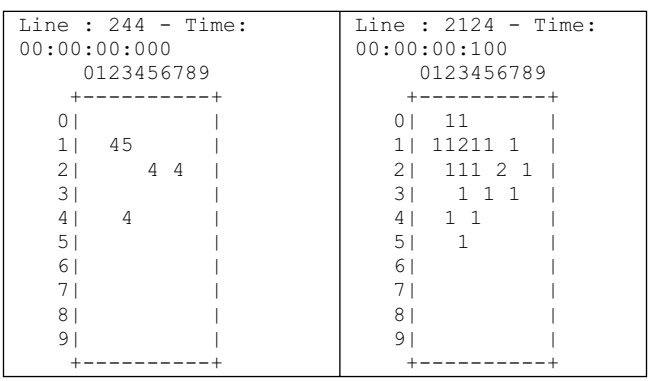


FIGURE 10.3 Distribution of cells with four particles.

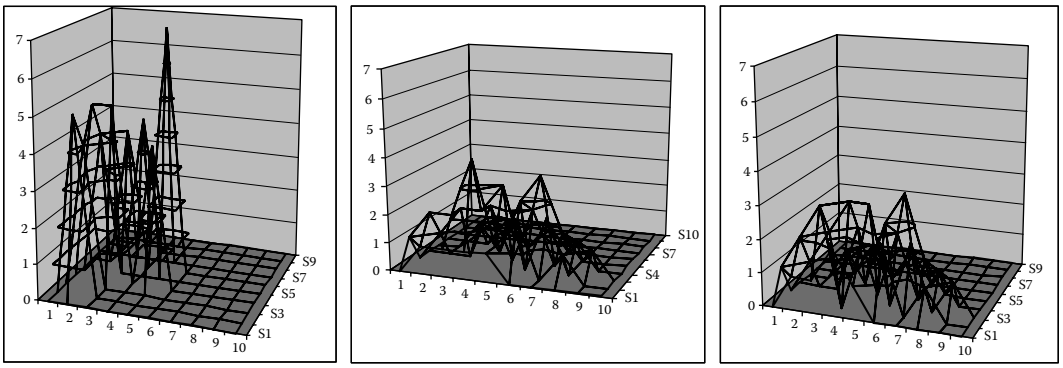


FIGURE 10.4 Distribution of cells with four or more particles.

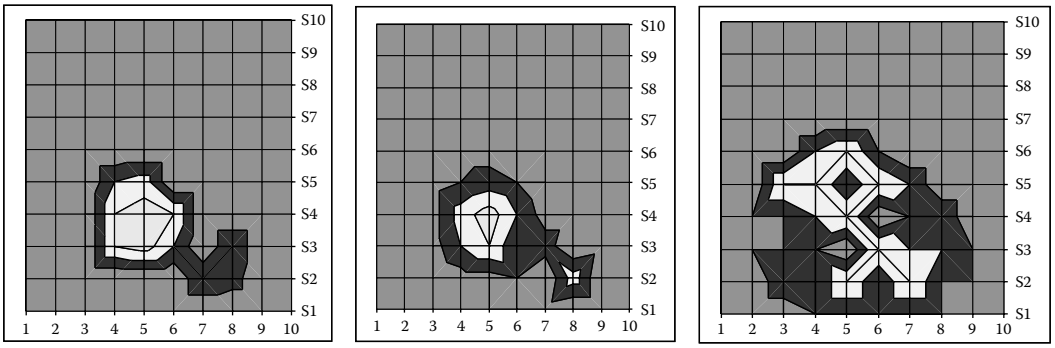


FIGURE 10.5 Eleven internal cells with four or more particles.

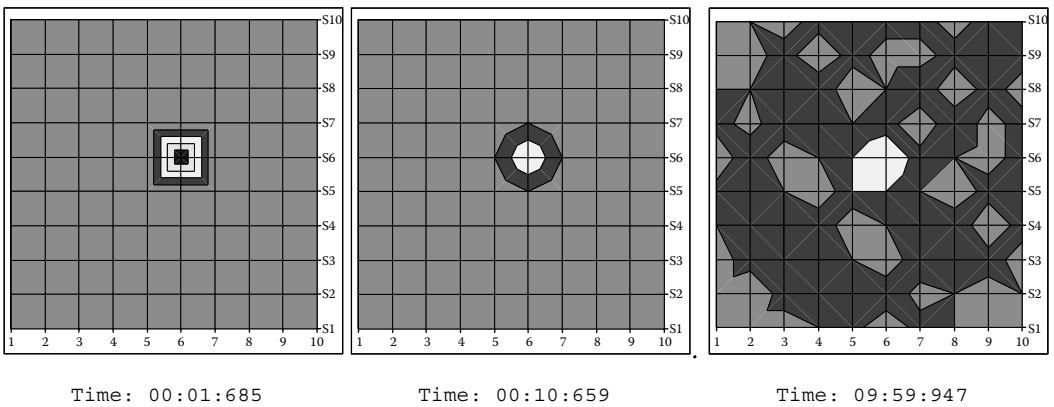


FIGURE 10.6 Executing the *SandPile* model.

distribution. Whenever a particle arrives from the generator, it appears in the middle cell (5,5) after a delay of two time units, after a delay of two time units and changes in their neighbors after 100 time units (as specified in our model). In each interval, one new sand particle is put into the center cell (5,5), which starts accumulating sand particles to the overflowing limit of four and then starts redistributing its contents to neighbors, producing avalanches.

Exercise 10.1

Create a mechanism to analyze the simulation results of this model. In order to do so, it is necessary to build a DEVS model based on the transducer example presented in Chapter 4. The transducer model should be connected to a few selected cells, in order to count whether the cells are distributing and, hence, an avalanche has started. This component will count all cells participating in an avalanche to estimate its severity.

Exercise 10.2

Create an alternate mechanism to analyze severity. In this case, it is necessary to build a zone in the border, with independent rules. The rules in these border cells must count sand particles escaping the board as a measure of avalanche severity.

10.3 SIMULATING THE REDECKING OF THE JACQUES CARTIER BRIDGE

The Jacques Cartier Bridge crosses the Saint Lawrence River in Montreal (Quebec, Canada), joining Montreal Island and the south shore of the river (information about the bridge can be found at http://www.pjcci.ca/English/jacques-cartier/PONT.HTM#bridge). In recent years, the bridge has been redecked. During the construction of these kinds of structures, space is a scarce resource that may cause crucial problems—particularly in bridges like this one, which is used by more than 100,000 vehicles a day. In this section, we show a method presented in Zhang et al. [8], where we represent space resources using Cell-DEVS, allowing the engineer to represent the spatial relationships between different activities.

In Figure 10.7, we identify the space resources represented in the construction, using abstract symbols for each of them. Two spaces are explicitly represented in this model: one for crews and one for trucks. Other spaces, like the moving path of the truck and waiting areas, are considered as available all the time and not explicitly represented. Conflict detection can be simplified by checking the state of each cell and avoiding an occupied cell being used by other objects. Based on this idea, a cell-based model was built using the site layout shown in Figure 10.7.

The model consists of multiple submodels interacting, as seen in Figure 10.8. The *bridge* is a three-dimensional Cell-DEVS model in which each layer represents *occupancy, control* (for mobility conditions), and *IDs* of the objects occupying the cells, respectively, as seen in Figure 10.9. The occupancy layer is used to define the type of equipment occupying a cell. The control layer is used to decide the moving state and direction, to detect conflicts, and to set the priority for moving, depending on the types of objects in the occupancy layer. The ID layer contains identification numbers for each piece of equipment. The combined information in the three layers gives a triplet of attributes for that location: <occupancy, mobility, ID>. Different rules are applied for simulating moving trucks, conflict detection, truck generation, direction changes, etc. The rules governing the interactions between layers guarantee the coupling of the attribute triplets.

Figure 10.10 shows a few examples of these rules. The first rule in the figure represents an empty truck moving to the west to an old section. The occupancy layer is used to make the truck move

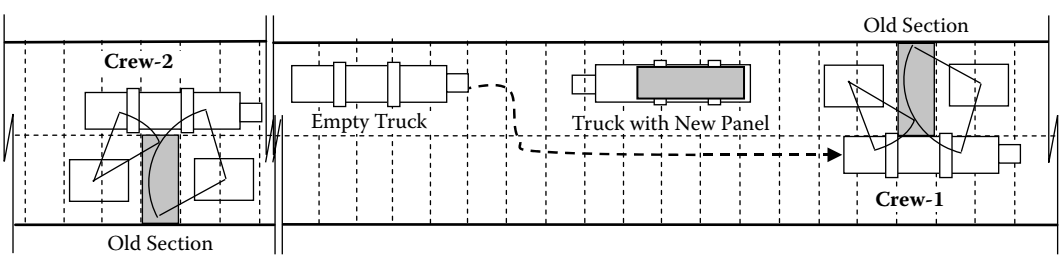


FIGURE 10.7 Worksite layout of the bridge. (Adapted from Zhang, C. 2005. *Proceedings of the 37th Conference on Winter Simulation*, 1541–1548.)

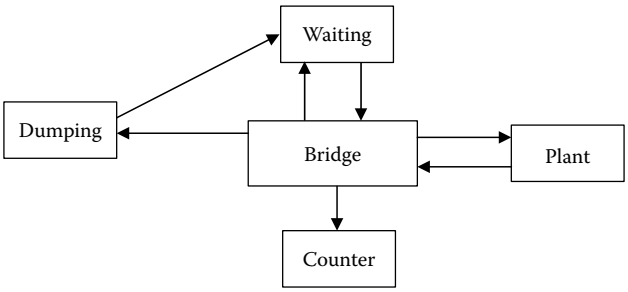


FIGURE 10.8 Interaction between models.

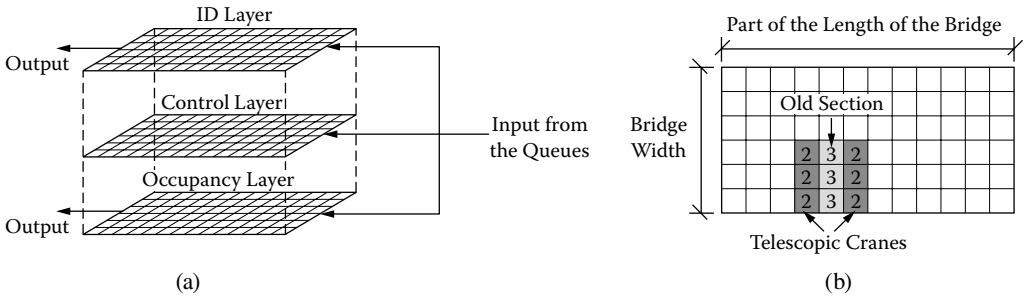


FIGURE 10.9 (a) The three layers for the bridge model; (b) cell representation of the occupancy layer. (Adapted from Zhang, C. 2005. *Proceedings of the 37th Conference on Winter Simulation*, 1541–1548.)

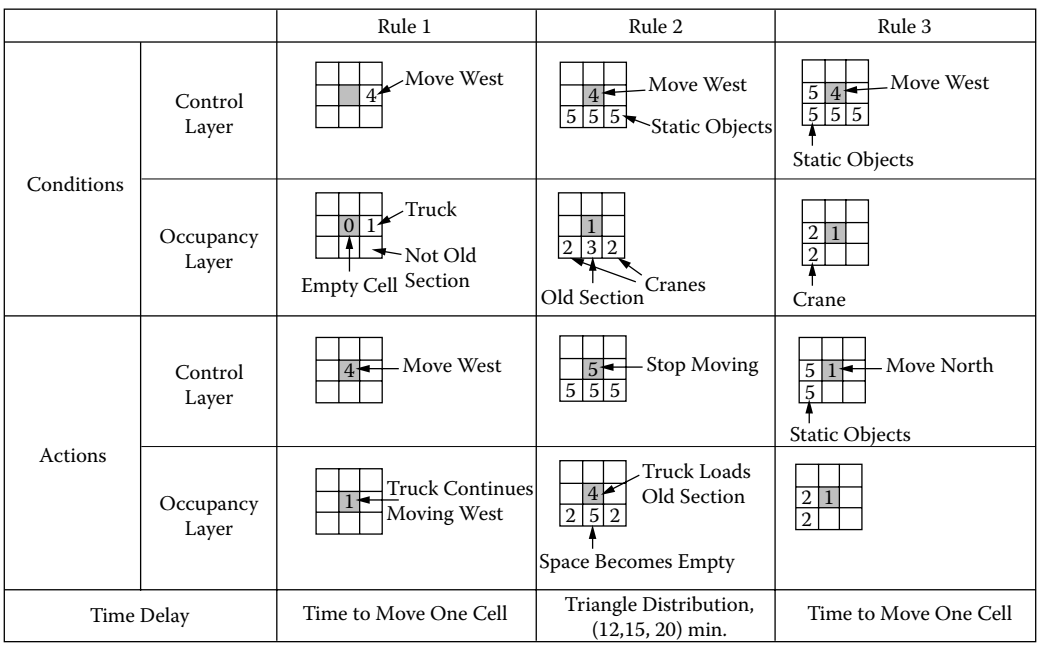


FIGURE 10.10 Bridge model rules. (Adapted from Zhang, C. 2005. *Proceedings of the 37th Conference on Winter Simulation*, 1541–1548.)

if it is not facing an old section. The control layer is used to decide that the truck is heading west (if a truck is moving in a certain direction, it will continue moving in the same direction). The timing delay for this action is calculated as the average speed of the truck. The second rule in the figure represents a truck arriving at the location of an old section. The occupancy layer is used to

determine that the truck should stop to load the old section. After the time delay needed for this task, the truck will change its occupancy state to four, which means the truck is now carrying the old section. In the control layer, the mobility state is changed to represent that the object is now static (5), while the truck is loading the old section. After the loading is done, the mobility state changes to (4) and the truck will move west again (assuming that this is the direction toward the dump area). The third rule deals with conflict detection: the cells representing static objects (such as old sections and cranes) are avoided by moving objects (e.g., trucks) as previously defined in Figure 10.8.

Figure 10.11 shows the four main areas in the model: the bridge, plant, dumping area, and waiting area for trucks transporting old sections, which are represented as separate cell models. To calculate productivity, a counter is also created in cells. Arrows show the input and output signals between different cell models. Each model is divided into cells, which can be occupied by different equipment and materials over time. Different numbers are used to represent different equipment states and the occupation of spaces. The cell dimensions are supposed to be 3×3 m. The total length of the main span is about 600 m, and the width of the bridge is about 20 m; therefore, the main span of the bridge can be represented as 200×6 cells.

The example presented in Figure 10.12 shows the bridge model with two cranes working around one old panel that needs to be replaced. An empty truck arrives for loading the old panel on the right-hand side. The truck moves to the west, and when it reaches the position of the panel, it load the old panel. At the same time, the panel changes to represent that the space is empty. The truck continues

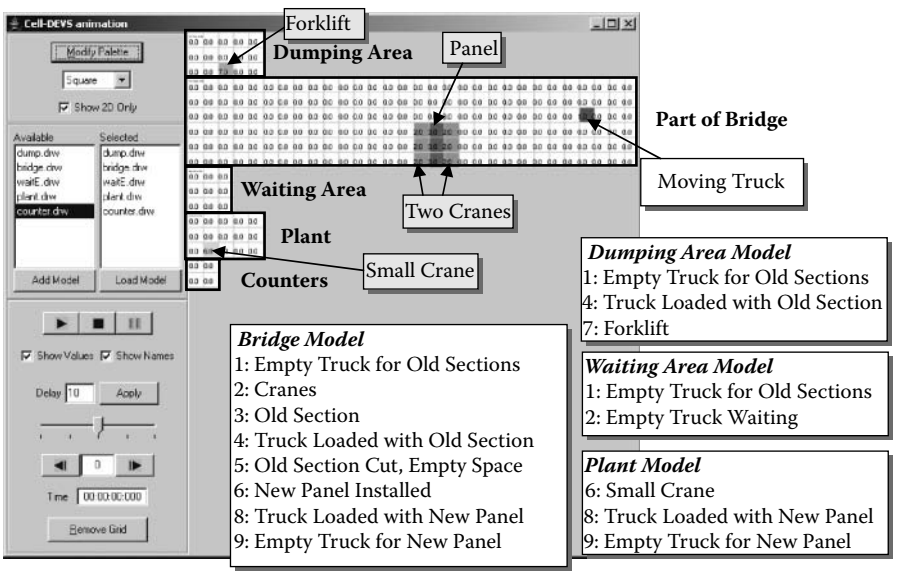


FIGURE 10.11 Graphical display of the Cell-DEVS model.

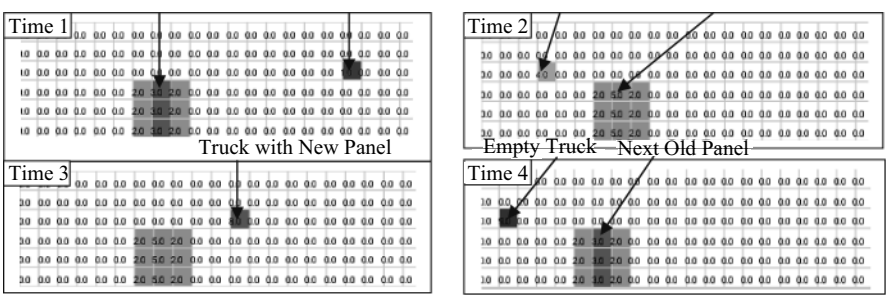


FIGURE 10.12 Part of bridge model showing the states of each cell.

to move to the left end of the bridge. The plant model uses a signal to inform that the old panel has already been cut. At this point, the plant model generates a truck for a new panel, which moves to the location of the small crane, where it changes to show that the truck has been loaded with a new panel. It then goes to the bridge and stops at the location of the empty space; the panel is installed and, after that, the crane moves to the left and the next old panel appears in the corresponding position. At the same time, the truck state is changed to "empty" and a signal is sent to the waiting area model. If there is a waiting truck, it is activated and it continuously moves to the bridge and begins a new cycle. In the dumping area model, the truck continuously moves to the forklift and unloads the old panel, and then it moves to the waiting area. When the waiting area model receives an external signal, it means that a truck has unloaded the old panel and is ready to go to the bridge for loading another old panel. The counter model counts the number of old and new panels.

10.4 ANALYSIS OF EVACUATION IN EMERGENCIES: CASE OF THE SAT BUILDING

Sophisticated evacuation models have been developed to assist rescue and emergency response crews with proper situation analysis and prompt reaction procedures. The ability to simulate and represent such situations increases training efficiency and creates the opportunity for better understanding of conditions. In Chapter 9, we presented a simple model used for evacuation processes on ships, based on references 9–11. In this section, we show an extended version of such a model, which can be used to study evacuation processes in buildings. This is a useful tool for civil engineers and building planners and can be used to study bottlenecks during emergency evacuations (e.g., fire or earthquakes), permitting them to analyze which solutions are effective to prevent congestion.

The model introduced in Chapter 9 was extended; in this case, we expand the basic architecture of the model and include advanced rules to define a person's behavior (e.g., a person in panic moves in the opposite direction of the exit, trying to find a different way to abandon the building, not paying attention to the flow of the rest of the evacuees). The example presented here has been applied to a three-dimensional floor plan of the Society for Arts and Technology (SAT), a building located on the Boulevard St. Laurent in downtown Montreal (a center devoted to the creation, development, and conservation of digital culture). This version of the model, presented in Poliakov, Wainer, and Jemtrud [12] and found in *./SatEvacuation.zip,* also uses two layers: one used to represent the building's architecture and the people moving within the building, and the other used for orientation purposes. The orientation layer contains information that serves to guide persons toward emergency exits, with a potential distance, as discussed in Chapter 9.

Figure 10.13 shows the initial configuration for these two layers for the SAT building. The figure on the left represents the walls, exits, and initial positions of the people. Walls are colored in black and exits in light gray. In the picture on the right, we can notice the second layer, which holds the distances to the exits. These are free cells where people can walk, and they contain information about the shortest path to the closest exit.

The cells containing people are set using the following notation [13] (Figure 10.14):

- *dn* represents the direction of movement of the people: 1: west (W); 2: southwest (SW); 3: south (S); 4: southeast (SE); 5: east (E); 6: northeast (NE); 7: north (N); 8: northwest (NW).
- *v* is the speed of the individual. This allows us to implement different people speeds, which makes for a more realistic evacuation (expressed in cells per second: one to five)
- *dp* is the last movement direction (from one to eight, as in *dn*).
- *p* represents the emotional state of the person: The higher this value is, the lower is the probability that a person becomes panicked.
- *m* represents a person's moving potential. As explained in Chapter 9, a person moves to decrease this potential by reducing the distance to the exit. If there is no available move in

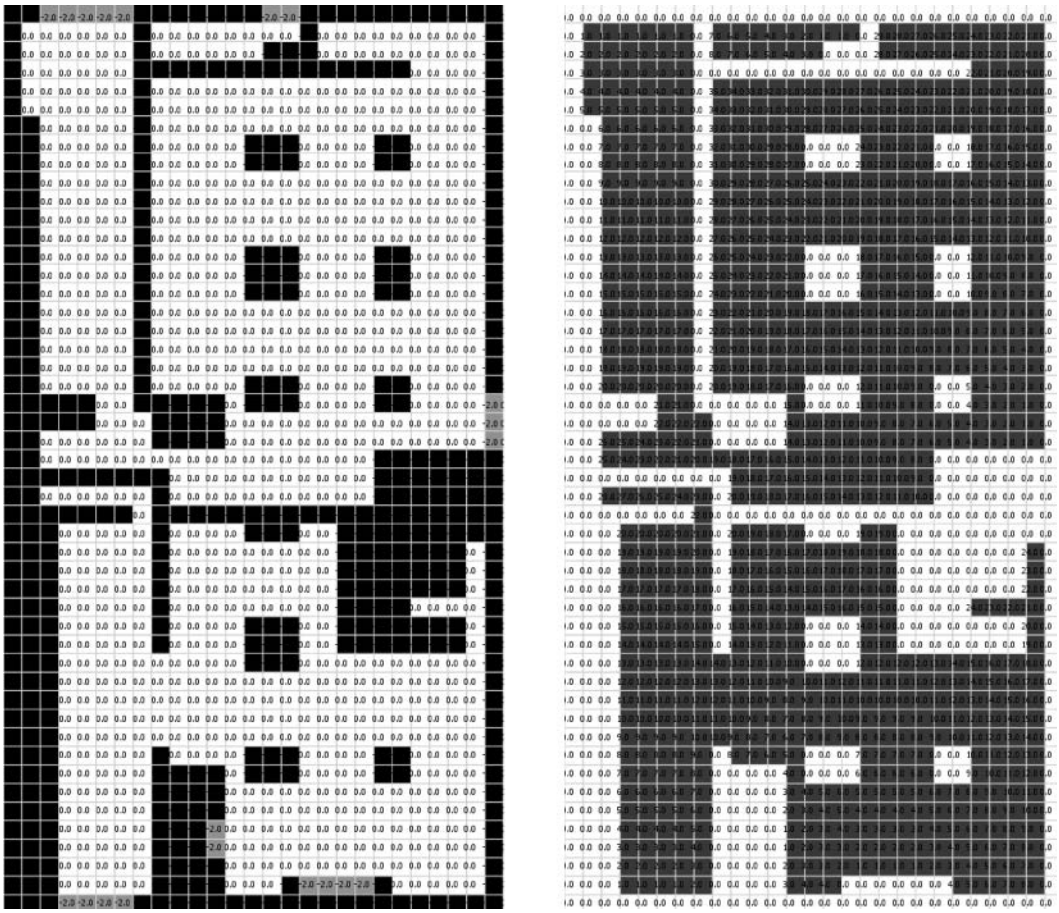


FIGURE 10.13 Initial definition for the SAT building layers.

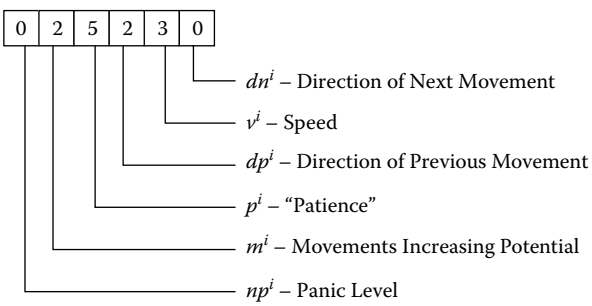


FIGURE 10.14 Representation of state variables on the SAT building cells.

this direction, a person will try to move to a neighboring cell that has the same potential. Otherwise, the person will move further away in an attempt to find another route.

- *np* defines the panic level (represents the number of cells that a person will move, increasing the cell potential). A situation where *p* is low and *np* is high will represent a high panic situation in which the person will very likely choose a wrong move.

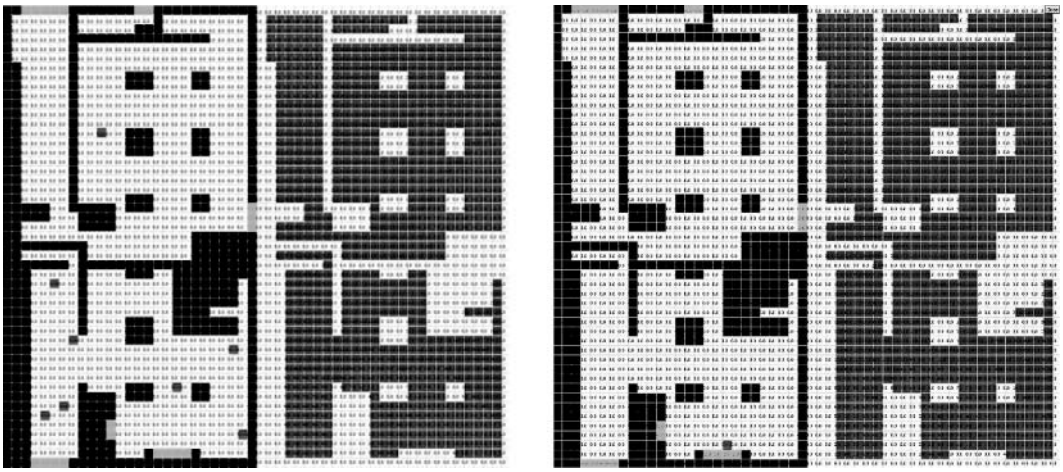
```
[SAT]
dim : (18,18,2)        delay : inertial       border : wrapped
localtransition : EvaRule
neighbors : (-1,-1,0) (-1,0,0) (-1,1,0) (0,-1,0) (0,0,0) (0,1,0)
(1,-1,0) (1,0,0) (1,1,0) (-1,-1,1) (-1,0,1) (-1,1,1) (0,-1,1) ...

[EvaRule]
% Rules to govern people movement: as on FIG xx, CHAPTER YY
...

% Rules to control panic behavior
rule : {trunc((0,0,0)/10)*10+1}  {1000/remainder(trunc((0,0,0)/10),10) }
{  (
    cellPos(2)=0 AND (0,0,0)>0 AND ( (0,-1,0)=0 OR (0,-1,0)=-2 ) AND
         remainder(trunc((0,0,0)/1),10)=0 AND remainder(trunc((0,0,0)/100000),10)>0
    )
AND
    (
    ( (0,-1,1) >= (1,-1,1) OR (1,-1,0)>0 OR (1,-1,0)= -1) AND
    ( (0,-1,1) >= (1,0,1)  OR (1,0,0) >0 OR (1,0,0) = -1) AND
    ( (0,-1,1) >= (1,1,1)  OR (1,1,0) >0 OR (1,1,0) = -1) AND
    ( (0,-1,1) >= (0,1,1)  OR (0,1,0) >0 OR (0,1,0) = -1) AND
    ( (0,-1,1) >= (-1,1,1) OR (-1,1,0)>0 OR (-1,1,0)= -1) AND
    ( (0,-1,1) >= (-1,0,1) OR (-1,0,0)>0 OR (-1,0,0)= -1) AND
    ( (0,-1,1) >= (-1,-1,1)OR ( -1,-1,0)>0 OR (-1,-1,0)=-1))}
```

FIGURE 10.15 Specification of evacuation model.

The rules as defined in the CD++ model files, shown in Figure 10.15, are responsible for recording and using each separate digit from the definition as specified previously. There are eight sets of rules like these, which determine the movement of the individual in each direction (in this case, this rule determines the movement from the cell to the west to the origin cell). These rules are applied on plane 0—that is, the plane with information about the individuals (*cellpos*(*2*) = *0*). To determine if we can move, we need an individual in the cell ((*0,0,0*) > *0*). This model does not allow collisions, so every time that a person moves, the destination to the west must be empty ((*0,-1,0*) = *0 OR* (*0,-1,0*) = *-2,* which represents the exits). These rules govern panic behavior, so we need to check the panic digit (e.g., if the first digit in Figure 10.14 is not 0); a person in panic will take a wrong path or will not follow the orientation path. In this case, the direction will be calculated taking a path where the cell's potential is increased. For instance, the cell to the west must have a higher distance value on plane 1 than the rest of the neighbors. In this case, we change the direction of the individual to go to the west (*trunc*((*0,0,0*)*/10*) × *10* + *1*), and we delay the movement according to the desired speed (*1000/remainder*(*trunc*((*0,0,0*)*/10*),*10*)).

Figures 10.16–10.21 show the results for different simulations of this model. They all use different cell values for human behavior—speed, panic, etc. Our first example considers eight people without panic, initially placed at random inside the building. The simulation results for this example can be found in Figure 10.16. As we can see, the building is evacuated without any complications in 13:015 s. Because the level of complexity is small, people follow the shortest path to exit the building. The building is almost empty (which is a normal condition for the SAT building); however, there are people in each sector of the floor. This evacuation is designed to give us a general idea of the exit directions people will follow. For instance, the cell (13,0) on the first plane takes the initial value 005040, which, considering that the definition in Figure 10.14 has no panic, the movement potential is zero and the level of patience is five (which results in a low probability for panic because the patience level is high). The two zero digits in the second and fourth positions represent the fact that the person has not moved yet. The digit 4 represents the initial speed.

The test depicted in Figure 10.17 includes more people close to specific exits to try to identify the bottlenecks in the building. The figure represents the placement of eight persons located in the lower left-hand corner of the plan (which is a café area). The panic level is still zero to follow an organized simulation and show us how people would evacuate under normal conditions. In this case, we can see a bottleneck situation, and we can visualize a pile-up around one specific exit. Although the



(a) (b)

FIGURE 10.16 (a) SAT at time 00:000—initial placement of people; (b) time 13:015—last person to leave the building.

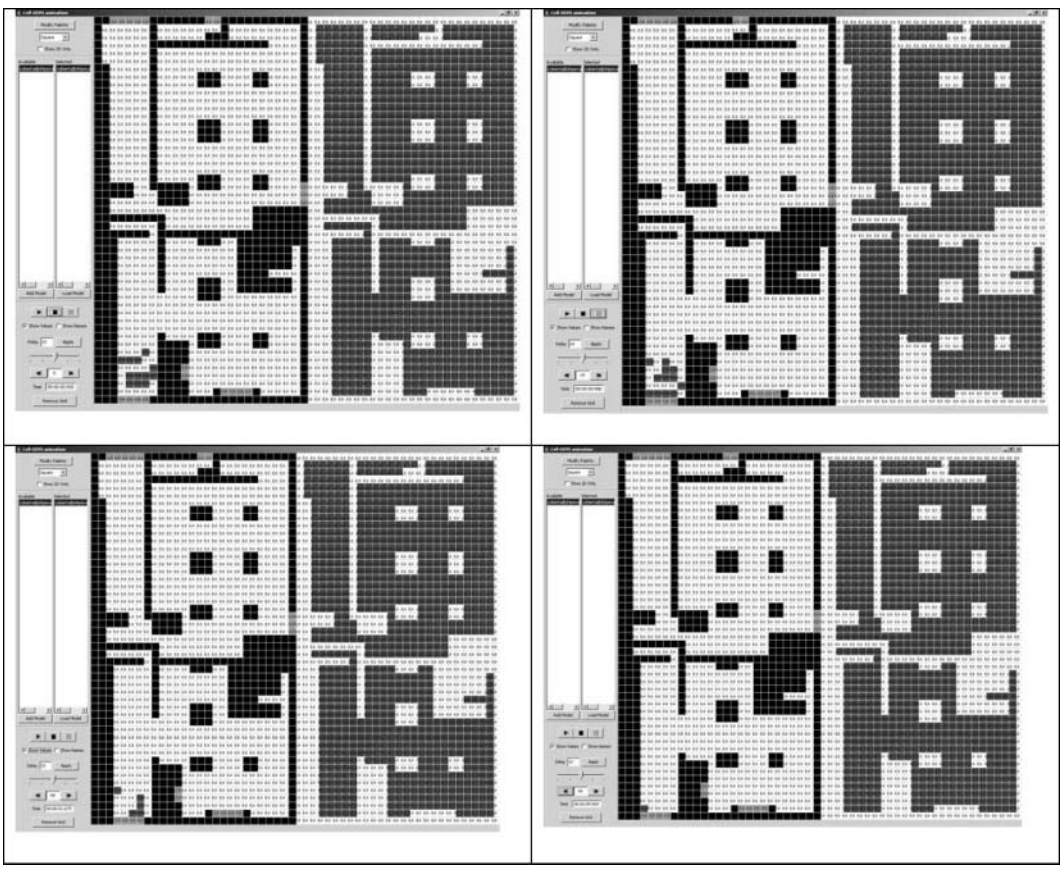


FIGURE 10.17 Time 00:000—initial placement of people; time 00:500—first movements of the individuals; time 01:005—people move toward the exit at the set speeds; time 04:005—last person to leave the building.

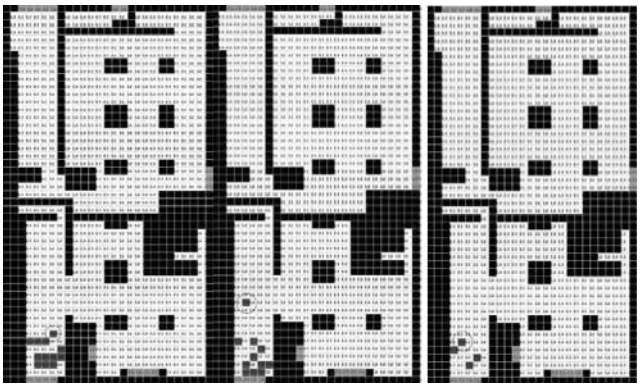


FIGURE 10.18 Evacuation with panic (one person): 05:004 s.

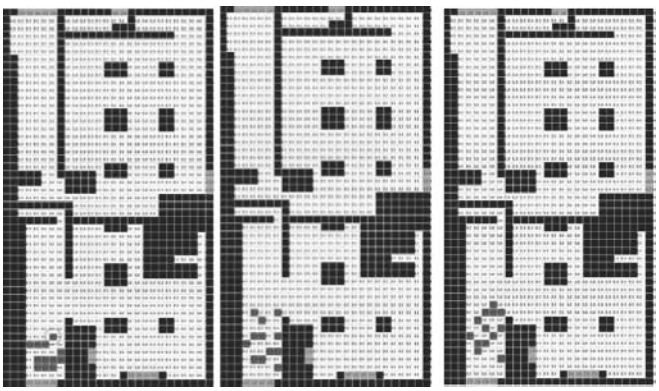


FIGURE 10.19 Evacuation with panic: 15:519 s.

total evacuation time is 04:005 s, this occurs because of the proximity of the individuals to the exit. As we can see, the building is also evacuated in an orderly fashion.

We then used the same model and included a panic effect in one of the persons. In order to observe the effect of panic on the simulation time, we used the exact same number of people and their positions as specified earlier. If we analyze the execution results in Figure 10.18, we notice that a person moves away from the exit due to a blocked exit, while the rest of the people leave the building normally. The total evacuation time is 05:004 s because the person in panic takes longer to abandon the building.

Figure 10.19 illustrates the case where the initial values are the same as specified before, but introducing panic for every person. We notice an increase in evacuation time up to three times larger than that observed in the previous example (which shows the influence of panic level).

We then increased the number of people and added more people to the other two exits on the right (Figure 10.20), which offers an interesting evacuation situation: two totally separate exits in great proximity to each other. This would allow us to follow people's behavior and proper choice of closest exit. In the case of no panic, the people would follow the second layer and decide where the closest exit is. However, if panic is introduced, the chaotic movements result in longer traveling times.

Figure 10.21 introduces a panic factor into all the individuals. Although the starting positions are maintained and the only difference between the two models is the panic level, we notice a difference

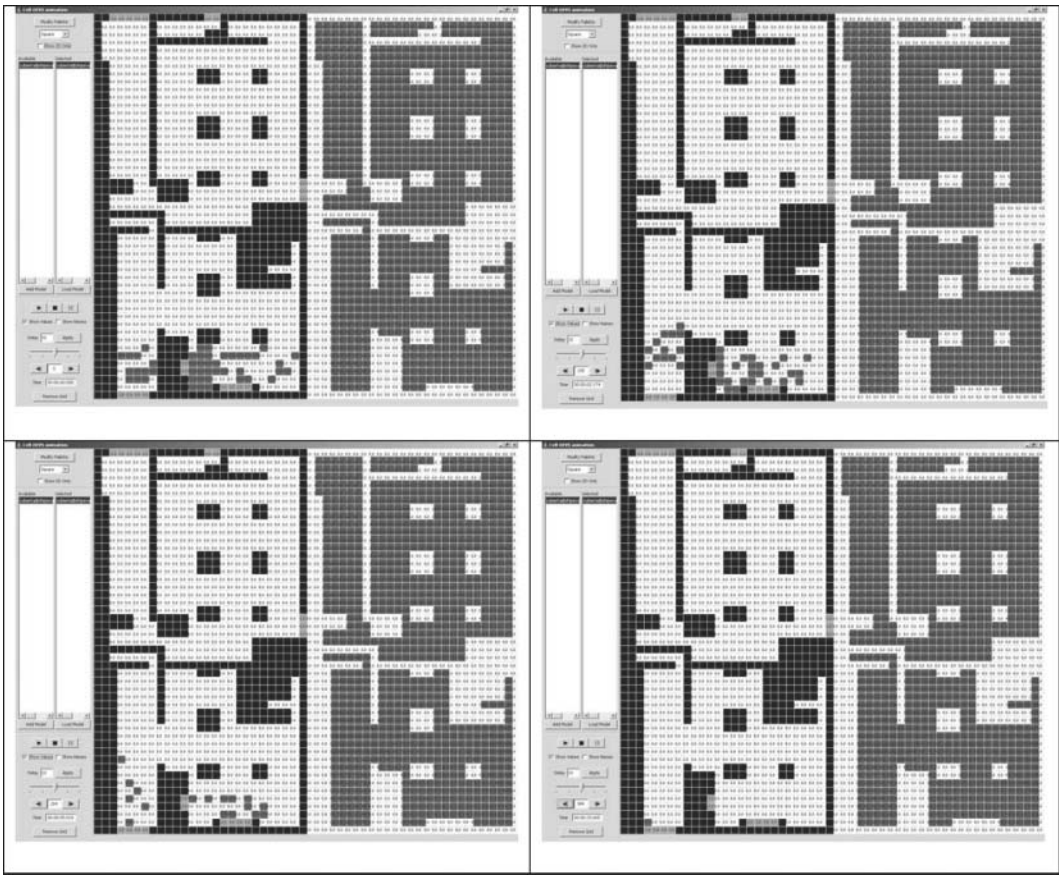


FIGURE 10.20 Low level of panic: 15:605 s.

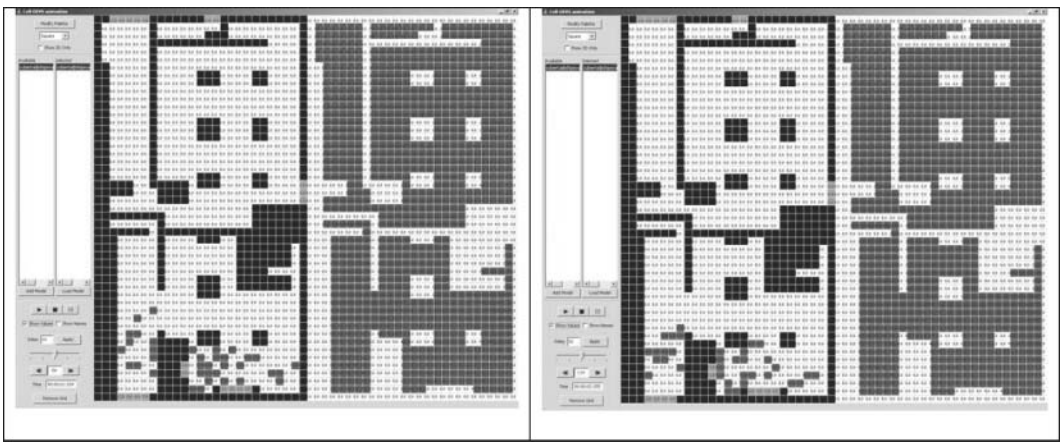


FIGURE 10.21 High level of panic: total evacuation time: 25:029 s.

of about 40% extra execution time between them: the second evacuation is slower due to the panic, which causes confusion and chaotic movements. This occurs because the exits get blocked, and people start moving in directions increasing their movement potential.

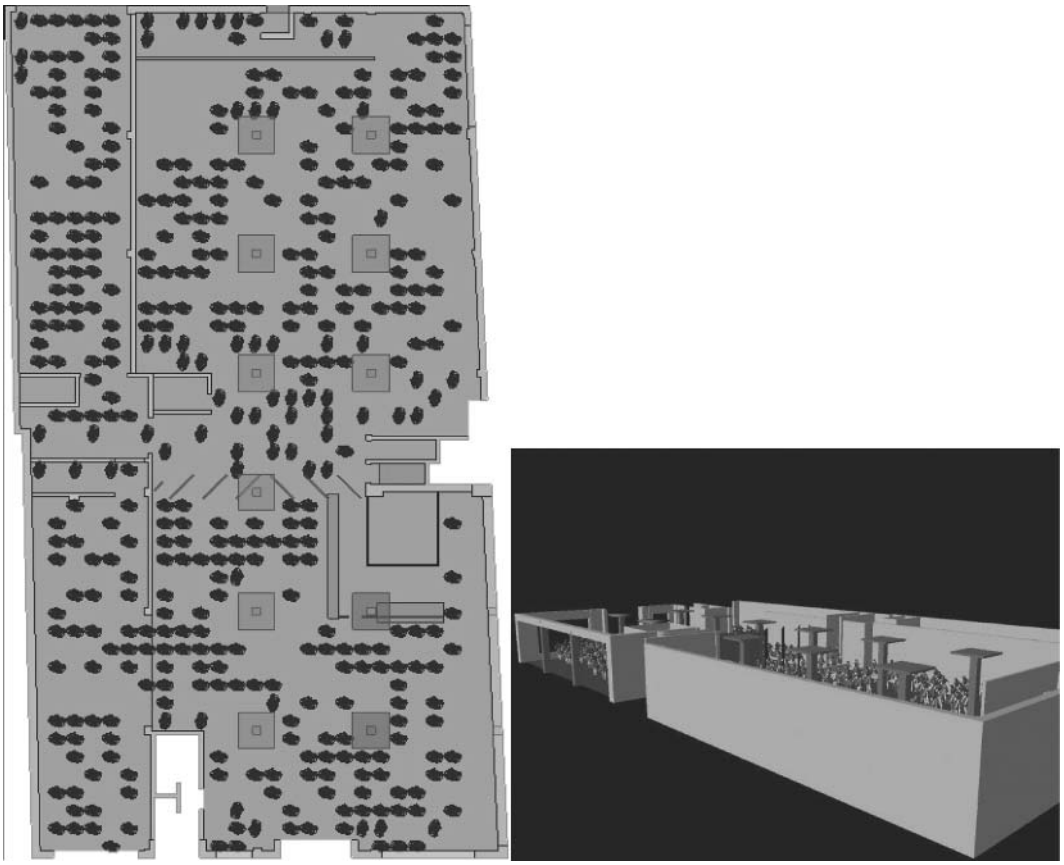


FIGURE 10.22 Three-dimensional visualization of the SAT evacuation model.

Figure 10.22 shows a three-dimensional version of this simulation, which is executed using CD++/Maya (whose details will be introduced in Chapter 16). The figure contains images rendered of separate frames that demonstrate the progressive motion of the human figures toward the dedicated building exits, using the SAT simulation results just introduced.

10.5 SUMMARY

This chapter introduced varied models with application in construction and architecture. We first introduced a Cell-DEVS model to simulate landslides. Using this simple model, we showed how to model systems that involve material accumulation and flow. Material flow and accumulation need special modeling consideration with Cell-DEVS to preserve mass conservation laws. In modeling such systems, not only updating model rules but also choosing the model characteristics would be important for correct behavior, because type of delay would affect that behavior.

We showed how to use spatial models to analyze issues on construction sites using simulation. Different simulation models were built to investigate space representations and conflicts during construction. Dividing space into cells can be used as a general way to represent workspaces and facilitate a simple method to analyze workspace conflicts.

Finally, we showed a model of building evacuation using specialized rules considering panic, shortest distance to the exits, collision detection, and different speeds of the individuals. A three-dimensional visualization graphical user interface enables sophisticated visualization to better understand the results, as will be discussed in Chapter 16.

REFERENCES

1. Halpin, D. W., and L. S. Riggs. 1992. *Planning and analysis of construction operations.* New York: Wiley Interscience.
2. Kamat, V. R. 2001. Visualizing simulated construction operations in 3D. *Journal of Computing in Civil Engineering* 15(4):329–337.
3. Hajjar, D., and S. AbouRizk. 1999. Simphony: An environment for building special purpose construction simulation tools. *Proceedings of 1999 Winter Simulation Conference,* Phoenix, AZ.
4. Malamud, B., and D. Turcotte. 2000. Cellular-automata models applied to natural hazards. *Earth System Science* May/June: 42–51.
5. Christensen, K. 1991. Dynamical and spatial aspects of sandpile cellular automata. *Journal of Statistical Physics* 63:653.
6. Pla-Castells, M., I. García, and R. J. Martínez. 2004. Granular system models for real-time simulation. Industrial Simulation Conference, Malaga, Spain, 88–93.
7. Saadawi, H., and G. Wainer. 2003. Modeling a sand pile application using cell-DEVS. *Proceedings of 2003 SCS Summer Computer Simulation Conference,* Montreal, QC, Canada.
8. Zhang, C., T. M. Zayed, A. Hammad, and G. Wainer. 2005. Representation and analysis of spatial resources in construction simulation. *Proceedings of WSC '05: Proceedings of the 37th Conference on Winter Simulation*, Orlando, FL, 1541–1548.
9. Klüpfel, H., T. Meyer-König, J. Wahle, and M. Schreckenberg. 2000. Microscopic simulation of evacuation processes on passenger ships. *Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry*, Karlsruhe, Germany, 63–71.
10. Kim, H., D. Lee, J. H. Park, J. G. Lee, B. J. Park, and S. H. Lee. 2001. Establishing the methodologies for human evacuation simulation in marine accidents. *Proceedings of 29th Conference on Computers and Industrial Engineering,* Montreal, QC, Canada.
11. Ameghino, J., and G. Wainer. 2004. Application of the cell-DEVS formalism for modeling cell spaces. *Proceedings of Artificial Intelligence, Simulation and Planning,* Jeju Island, Korea, LNCS 3397.
12. Poliakov, E., G. Wainer, and M. Jemtrud. 2007. A busy day at the SAT building. *Proceedings of AIS 2007, Artificial Intelligence, Simulation and Planning,* Buenos Aires, Argentina.
13. Meyer-König, T., H. Klüpfel, and M. Schreckenberg. 2001. A microscopic model for simulating mustering and evacuation processes onboard passenger ships. *Proceedings of TIEMS, the International Emergency Management Society's Eighth Annual Conference,* Oslo, Norway.

11 Models in Environmental Sciences

11.1 INTRODUCTION

Modeling and simulation have been widely used for studying behavior in the environmental sciences. In recent years, some of the studies have been based on the use of cellular models. We will show how CD++ simplifies the construction of such cellular models by allowing an intuitive rule-based model specification. We will present the definition of different models, including pollution in a basin, vegetation growth, watershed formation, and fire spreading, and will focus on how to define such rule-based applications using the Cell-DEVS methodology and how to implement the model in CD++.

11.2 VIABILITY OF POPULATION ON A FIELD

The first model we will present in this chapter permits modeling the viability of population on a field, based on the work presented in Darwen and Green [1]. The population can include vegetal or animal life, and the goal is to study the connection between the area occupied initially by the population and its chances of survival. The population is not limited to an area, and the members can roam freely. In order to study viability of the population, different parameters are used; this could result in indefinite expansion (viability), growth to a steady state, or extinction. The model considers two types of dynamics: the local, governed by parameters of fertility and maximum population per cell, and migration, which considers that the population of a cell can increase by immigration from neighboring cells. As discussed in Darwen and Green [1], each cell on the field contains a part of the population, and the dynamics on each cell are defined by

$$N(t+1) = r \cdot N(t) \cdot \left(1 - \frac{N(t)}{K}\right) \tag{11.1}$$

where

N represents the size of the population on a cell;
t represents the current time;
$t + 1$ represents the next time unit;
$r \geq 0$ represents the reproduction rate; and
K represents the maximum local population on each cell.

The reproduction rate considers the fertility and the population mortality, as follows:

$$\begin{aligned} &N(t+1) = r \cdot N(t) + \left(\frac{-r}{K}\right) \cdot N(t)^2 \Leftrightarrow N(t+1) = N(t) + r \cdot N(t) - N(t) + \left(\frac{-r}{K}\right) \cdot N(t)^2 \Leftrightarrow \\ &N(t+1) = N(t) + \left(r-1\right) \cdot N(t) + \left(\frac{-r}{K}\right) \cdot N(t)^2 \end{aligned} \tag{11.2}$$

Let $\alpha = (r - 1)$ be the fertility rate and $\beta = (-r/K)$ be the mortality rate.

The model also considers migration among the four adjacent cells—north (N), south (S), east (E), and west (W)—as follows:

$$\delta^2 N_{x,y} = N_{x,y-1} + N_{x,y+1} + N_{x+1,y} + N_{x-1,y} - \left(4 \cdot N_{x,y}\right) \tag{11.3}$$

Consequently, if we add this term to Equation (11.2), we can obtain the behavior of the cell (x,y) at time t:

$$N_{x,y}(t+1) = N_{x,y}(t) + \alpha \cdot N_{x,y}(t) + \beta \cdot N_{x,y}(t)^2 + \gamma \cdot \delta^2 N_{x,y}(t) \tag{11.4}$$

where $\gamma < 1$ is the proportion of the population on a cell that is ready to migrate.

Finally, considering that population is never negative and Equation (11.4) can violate this condition for some parameters, we use the heavyside operator $H(z)$, which satisfies $H(z) = z \; \forall \; z > 0$ and $H(z) = 0$ otherwise. The result is

$$N_{x,y}(t+1) = H\left(N_{x,y}(t) + \alpha \cdot N_{x,y}(t) + \beta \cdot N_{x,y}(t)^2 + \gamma \cdot \delta^2 N_{x,y}(t)\right) \tag{11.5}$$

If the population expands up to the borders of the field of study, the population is viable. The population is not affected by external factors, including external immigration. If the population reaches the value

$$N_{eq} = \frac{r-1}{r} \cdot K,$$

the system is in steady state, and the population does not change in successive generations. A Cell-DEVS implementation of the model presented in Wainer [2] and found in *./viability.zip* is shown in Figure 11.1.

As we can see, the model uses a 40×40 cell space using the N, S, E, and W neighbors. The *viability* rules define the local computing and delay functions using Equation (11.5). In this case, we use $r = 0.18$, $K = 200$ (maximum population of the cell), and $\gamma = 0.22$. The simulation results for some basic cases are presented in Figure 11.2; three basic behaviors can be established for different parameters using the same initial conditions. Figure 11.2(a) shows the results for viable populations that expand to the borders of the grid in 200 transitions, as a consequence of using a large α (fertility rate) and

```
[viability]
type : cell     delay : transport   dim : (40,40)
border : nowrapped
neighbors : (-1,0) (0,-1) (0,0) (0,1) (1,0)
localtransition : viability-rules

[viability-rules]
rule : {if( ( (0,0)+(0.2)*(0,0)+(-0.006)*power((0,0),2)+(0.18)*

            ((0,-1)+(0,1)+(1,0)+(-1,0)+(-1)*(4*(0,0) ) ) ) < 0, 0,
                if(((0,0)+(0.2)*(0,0)+(-0.006)*power((0,0),2)+(0.18)*((0,-1)+(0,1)+(1,0)+
                    (-1,0)+(-1)*(4*(0,0))))>200, 200, trunc((0,0)+(0.2)*(0,0)+
                       (-0.006)*power((0,0),2)+(0.18)*((0,-1)+(0,1)+(1,0)+(-1,0)+(-1)*
                        (4*(0,0))))
        ) )  } 100 {cellpos(0)>0 and cellpos(1)>0 and cellpos(0)<63 and cellpos(1)<63  }
```

FIGURE 11.1 Cell-DEVS specification of a portion of the model using CD++. (From Wainer, G. 2006. *Simulation: Transactions of the Society for Modeling and Simulation International* 82:635–660.)

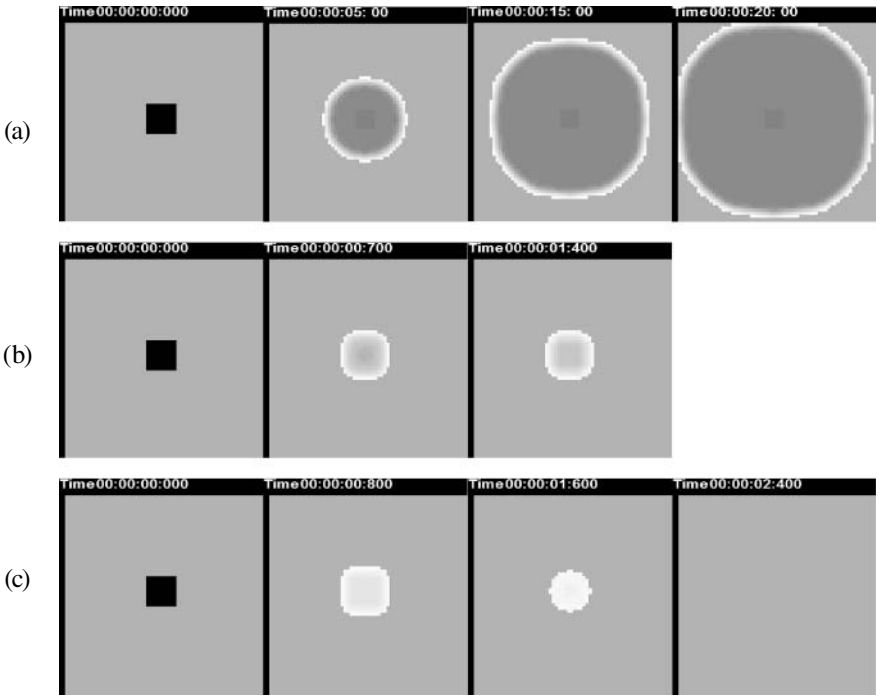


FIGURE 11.2 Viability rules—basic behavior: (a) K = 200; r = 1.2; $\alpha = 0.2$; $\beta = -0.006$; $\gamma = 0.18$; and N = 166. (b) K = 200; r = 1.1; $\alpha = 0.1$; $\beta = -0.0055$; $\gamma = 0.18$; and N = 166. (c) r = 1.05; $\alpha = 0.05$; $\beta = -0.00525$; $\gamma = 0.18$; and N = 190.

a small γ (mortality). Figure 11.2(b) shows a population that becomes steady after some expansion. This is due to the use of an intermediate-size α combined with a small γ. In the last example, we can see that the population expands up to a certain point and then diminishes until extinction.

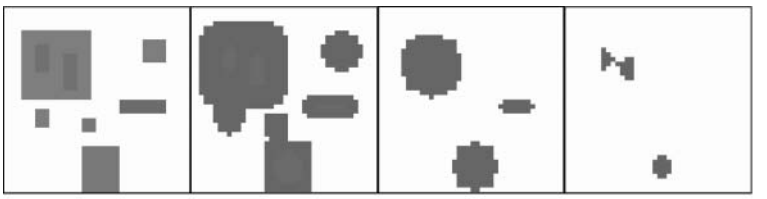
Figure 11.3 shows the execution results for a more complex scenario, found in *./population.zip*. In the first example in the figure, we can see that the population initially expands but later disappears, due to a small rate of fertility combined with low mobility. In the second case, we see population expanding and a viable case due to a higher fertility rate (while keeping the mobility rate). In the last example, population expands further than in the first case, but it does not cover the whole region. Although mobility has increased, the low reproduction rate makes growth slow while producing a steady-state condition in which population stops growing.

Exercise 11.1

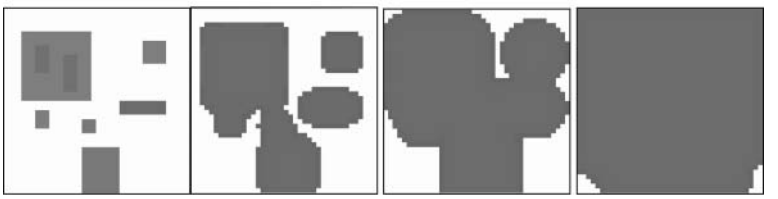
Build a program that allows the user to analyze the viability of the population in time. To do so, count the number of cells' values at every time step (searching in the draw files and plotting the results in a timeline). The program should graph the average value and standard deviation at every time step. Test the software using different initial scenarios (using the scenario generation tool included in the *./population.zip* model).

11.3 ANT FORAGING MODELS

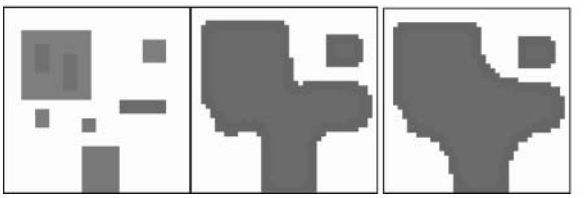
In this section, we present two different mechanisms for modeling ants moving on the ground while searching for food. In the first version, which is based on the Vants model by Langton [3], the ants move freely and, when they find vegetation, they follow a two-step process: they first cut the leaves and then the root. When ants find vegetation, they eat the leaves and rotate 90° to the right. When



r = 1.1, K = 100, γ = 0.12



r = 1.5, K = 100, γ = 0.12



r = 1.25, K = 100, γ = 0.22

FIGURE 11.3 Viability analysis scenarios.

the leaves have been eaten, the ants get the root and rotate 90° to the left. If there is no vegetation, the ants move forward. Figure 11.4 shows CD++ implementation of the model, which was presented in Ameghino and Wainer [4] and can be found in *./Vants.zip*.

As can be seen, we used a three-dimensional model. The first layer represents the field, and the second layer is used for collision detection by determining different steps to evaluate the movement of the ants. We use four states to represent the present direction, combined with the field's state (1 when there is vegetation in the cell, 2 when the leaves have been eaten, and 3 or 4 for empty cells). For instance, a value of 24 means that the leaves have been eaten (2) and the ant is moving west (1: S; 2: E; 3: N; 4: W). In order to avoid collisions, we define auxiliary ant directions (5: S; 6: E; 7: N; 8: W).

In the first rule presented here, if an ant is moving south and it finds vegetation, it will potentially rotate to the east and eat the leaves. In this rule `(0,0,1) = 1` defines the potential movement for the ant. When a cell in the second layer is 2, we must analyze only rules to compute the next position of the ants, thus avoiding collisions. For instance, the second rule shown in the figure takes an ant moving to the east `(cellpos(2) = 0 and (0,0,1) = 2 and ((0,0,0) = 26 or (0,0,0) = 36 or (0,0,0) = 46)` and checks whether it is facing an ant moving to the west `((fractional((0,1,0)/10) * 10) = 8)`. In that case, it changes according to the current cell's state. If the cell is empty (46), the current ant changes direction (34 = moving to the west and empty). Otherwise (i.e., the cell's value = 26 or 36), we add 4 to the direction (auxiliary direction) to avoid collisions. We also analyze the seven possibilities for collision for an ant, making those in conflict stop while only one moves. We then check whether the ant has the space to move. The term `(fractional((-1,1,0)/10) * 10) = 7` or `(fractional((1,1,0)/10) * 10) = 5` or `(fractional((0,1,0)/10) * 10)! = 0))` in the third rule used to check whether the ant conflicts with the one at the northeast or the southeast. In that case, we move the ant and eliminate it from the origin cell, as seen in the following two rules. Then, show the rules used for the growth of

```
[vants]
type : cell              dim : (10,10,2)
delay : transport        border : wrapped
neighbors: (-2,0,0) (-1,-1,0) (-1,0,0) (-1,1,0) (0,2,0) (0,0,1) (0,-2,0) (0,-1,0) (0,0,0) (0,1,0)
neighbors: (1,-1,0) (1,0,0) (1,1,0) (2,0,0)
localtransition : calculus

[calculus]
...
rule : 26 100 { cellpos(2)=0 and (0,0,1)=1 and (0,0,0)=11 }
...
rule : { if((0,0,0)=46, 34, trunc(((0,0,0)/10))*10)+4 } 100 { cellpos(2)=0 and (0,0,1)=2 and

            ((0,0,0)=26 or (0,0,0)=36 or (0,0,0)=46) and (fractional((0,1,0)/10)*10)=8 }
...
rule : { trunc(((0,0,0)/10))*10)+2 } 100 {cellpos(2)=0 and (0,0,1)=3 and ((0,0,0)=26 or
           (0,0,0)=36 or (0,0,0)=46) and ((fractional((-1,1,0)/10)*10)=7 or
                (fractional((1,1,0)/10)*10)=5 or (fractional((0,1,0)/10)*10)!=0)}
...
rule : { trunc(((0,0,0)/10))*10+4 } 100 {cellpos(2)=0 and (0,0,1)=4 and ((0,1,0)=28 or
          (0,1,0)=38 or (0,1,0)=48)}
...
rule : 30 100 {cellpos(2)=0 and (0,0,1)=5 and (0,0,0)>=45 and (0,0,0)<=48}

%vegetation growth
rule : 20 2973 {cellpos(2)=0 and (0,0,0)=30 and statecount(20)>=4 and (statecount(10) +
                        statecount(20) + statecount(30)>12)}
rule : 10 2677 {cellpos(2)=0 and (0,0,0)=20 and statecount(10)>=3 and (statecount(10)+
                        statecount(20)+statecount(30)>12)}

%2nd plane counter
rule : { (0,0,0) + 1 } 100 {cellpos(2)=1 and (0,0,0)<7}
rule : 1 100 {cellpos(2)=1 and (0,0,0)>=7}
```

FIGURE 11.4 Specification of the Vants model.

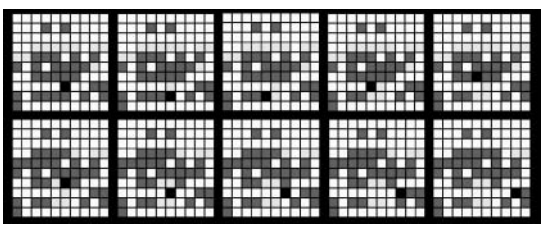


FIGURE 11.5 Execution results of an ant foraging model.

the vegetation. We finally show the rules used for incrementing a counter in the second layer (triggering each phase that allows the corresponding rules to execute).

Figure 11.5 shows the execution of the model using CD++. The dark cells contain vegetation, and an ant moving in the lower rows is eating the leaves. The ant behaves using the rules recently explained. We also can see the growth of the vegetation, represented as a change of state for the lighter cells, where a long delay is used.

Exercise 11.2

Modify the cellular model and change the amount of vegetation available and the number of ants in the field.

The next example is an ecological model based on work by Nishidate, Baba, and Gaylord [5], in which ants follow a path from an anthill to a source of food. When an ant finds food, it returns to the anthill, leaving a hormone (pheromone) in its path. The other ants use this as a signal that leads to the source of food. The model was implemented in CD++ and presented in Ameghino, Glinsky, and Wainer [6] (the source for the model can be found in *./ants2.zip*). Each cell in the Cell-DEVS space

represents a section of the field. The cell can contain vegetation or pheromone, an ant seeking food, an ant following pheromone, soil, or an ant following pheromone and returning to the anthill with food [2,5].

To avoid collisions, two or more ants in conflict change direction at random until one ant can actually move. When an ant takes food from the ground, it changes its course to the opposite direction and follows the pheromone path to return to the anthill. In a case in which there is no pheromone, the ant moves at random, seeking the anthill or another pheromone path but leaving its own pheromone trace. Table 11.1 describes the cell state coding.

TABLE 11.1
Cell State Encoding of the Ants Model

0	0	F	F	D
1	0	0	C	C
2	Q	0	0	D
3	0	0	0	0

Notes: *F*: Pheromone concentration; can vary from 1 to 99. *D*: Ant direction: 0, north; 1, east; 2, south; 3, west. *C*: Quantity of food; can vary from 1 to 99. *Q*: Flag indicating if an ant is carrying food; 1: carrying food, 0: seeks food.

Figure 11.6 describes the model specification. We define the dimensions of the cell space, neighborhood, initial values, and, finally, the rules that define the behavior of an ant. We used different macro definitions to avoid large statements in the specification of rules. In this case, macros provide an easy mechanism for frequent statements, such as checking the existence of an ant, food, or pheromone in the neighboring cells. Hence, the rules specify the behavior of an ant based on its direction, current location, and the value of the adjacent cells. For instance, the first rule checks whether an ant is on the origin cell. In this case, it checks to see whether the direction is N (0) and detects collisions. It then checks the cell to the northeast, northwest, or further to the north in order to see whether the two ants intend to take the same cell. In this case, the ant rotates to the south.

Figure 11.7 shows the execution of the model. The black cells represent two ants seeking food and the gray cells in the upper left area of the graph represent two ants carrying food and their

```
[ant]
type : cell     dim : (20,20)
delay  : transport      border : nonwrapped
neighbors : (0,-2) (-1,-1) (0,-1) (1,-1) (-2,0)
neighbors : (-1,0) (0,0)   (1,0)  (2,0)  (-1,1)
neighbors : (0,1)  (1,1)   (0,2)
…
[rules]
rule : { (0,0) + 2 } 1000 { #Macro(isAnt00) and #Macro(dir00) = 0 and ((#Macro(isAnt19) and #Macro(dir19) = 3)
     or (#Macro(isAnt99) and #Macro(dir99) = 1) or (#Macro(isAnt08) and #Macro(dir08) = 2))}
rule : { (0,0) + 2 } 1000 { #Macro(isAnt00) and #Macro(dir00) = 1 and ((#Macro(isAnt19) and #Macro(dir19) = 2)
     or (#Macro(isAnt20) and #Macro(dir20) = 3) or (#Macro(isAnt11) and #Macro(dir11) = 0)) }
…
rule : { 21003 } 1000 { #Macro(isAntB00) and #Macro(dir00) = 2 and #Macro(isAntB91) and #Macro(dir91) = 1 }
rule : { 0 }     1000 { #Macro(isAntB00) and #Macro(dir00) = 2 and #Macro(isNothingAnt01) }
…
rule : {(0,0)} 10 {t}
```

FIGURE 11.6 Specification of the ants model.

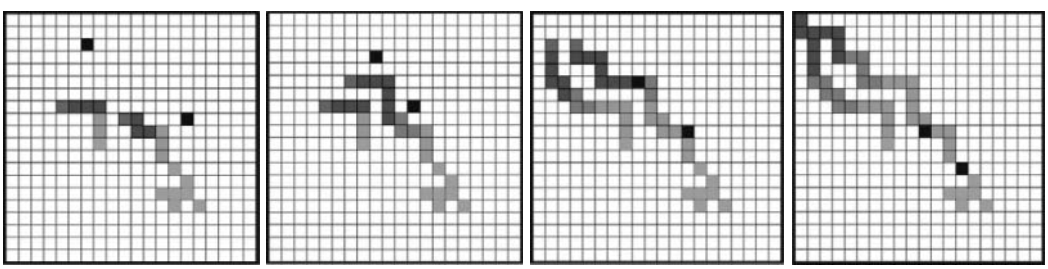


FIGURE 11.7 Ants moving on the ground: (a) two ants returning to the anthill and two ants seeking food; (b) two ants have found pheromone; (c) both ants have found the source of food using the pheromone path; (d) ants returning to the anthill following the pheromone path.

pheromone paths. The source of food is located in the lower right section of the graph, and different gray colors represent the concentration of pheromone showing the way to the food.

Exercise 11.3

Modify the cellular model and change the amount of food available and the number of ants in the field.

Exercise 11.4

Analyze the ants model found in *./ants.zip* and compare with the previous two models.

11.4 WATERSHED FORMATION

In this section we will present a watershed formation model, based on the one previously introduced in Moon et al. [7]. A watershed is a natural region that acts as the water-receiving area of a drainage basin. The water that accumulates has different origins: rain, rivers, and snow melting. The watershed is represented as a hydrology system built as a cell space.

The watershed is considered as a cellular model organized in several vertical layers: air, vegetation, surface waters, soil, ground water, and bedrock. The rainfall input is partially retained by vegetation, and the rest infiltrates. The model in Moon et al. [7] represented the water flow and accumulations based on the characteristics of the different layers, as shown in Figure 11.8.

The model considers that the height of accumulated water depends on the rainwater that reaches the ground, the water received from neighbor cells, the water that overflows to neighbor cells (which depends on the topology of the terrain), and the water that the ground absorbed. Based on the equations for this model, the CD++ model shown in Figure 11.9 was developed to simulate the accumulation of water under the presence of constant rain (7.62 mm/h), as shown in Ameghino and Wainer [4] (found in *./watershed.zip).*

The rules represent the accumulation of water. It first takes the amount of water present in the cell, and the rainfall up to the present moment (which is stored on layer 1). Then we consider how much water must be passed to the neighbors by comparing the level in the current cell and in the neighborhood and how much water is received from the inverse neighborhood.

We can see the execution results of this model in Figure 11.10. In the first figure we show an initial state, representing the slope of the terrain before raining (darker cells represent the bottom of the area; there is no water at the beginning of the simulation). Each cell occupies 1 × 1 m. The on the upper right figure shows the execution results after intense rain (0.0022 mm/s) after 10 min of simulated time. We can see how the rain accumulates in the lower levels of the terrain, the level of water rises, and a watershed is formed.

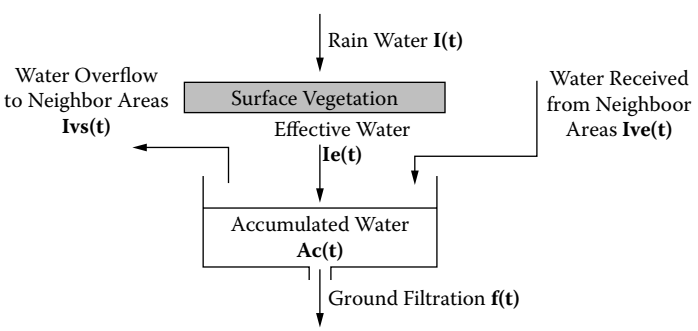


FIGURE 11.8 Hydrology model. (Adapted from Moon, Y. et al. 1996. *IEEE Transactions on Systems, Man and Cybernetics* 288–296.)

```
[Watershed]
type : cell          dim : (30,30,2)
delay : transport    border : nowrapped
neighbors : (-1,0,0) (0,-1,0) (0,0,0) (0,1,0)
neighbors : (1,0,0) (-1,0,1) (0,-1,1) (0,0,1)
neighbors : (1,0,1) (0,1,1)
localtransition : Hydrology

[Hydrology]

rule : {0.0022 + (0,0,0) - if (((0,0,1)+(0,0,0) > ((-1,0,1) + (-1,0,0)), ((0,0,0)+(0,0,1)-(-
1,0,0)-(-1,0,1))/1000)* (0,0,0))/1000),0)- if(((0,0,1)+(0,0,0))> ((1,0,1) + (1,0,0)),((0,0,0) +
(0,0,1) - (1,0,0) - (1,0,1))/1000)*(0,0,0))/1000),0) - if(( ((0,0,1)+(0,0,0)) > ((0,-1,1)+(0,-
1,0)) ,((0,0,0) + (0,0,1)-(0,-1,0)-(0,-1,1))/1000) *(0,0,0))/1000),0) - if(((0,0,1) + (0,0,0)) >
((0,1,1) + (0,1,0)),(((0,0,0) + (0,0,1) - (0,1,0) - (0,1,1))/1000) * (0,0,0))/1000),0) + if(((-
1,0,1) + (-1,0,0)) > ((0,0,1) + (0,0,0)),((-1,0,0) + (-1,0,1) - (0,0,0)-(0,0,1))*(-
1,0,0))/1000),0) + if(((1,0,1) + (1,0,0)) > ((0,0,1) + (0,0,0)),((1,0,0) + (1,0,1) - (0,0,0) -
(0,0,1)) * (1,0,0))/1000),0) + if((0,-1,1) + (0,-1,0)) >((0,0,1) + (0,0,0)), ((0,-1,0) + (0,-1,1)
- (0,0,0) - (0,0,1)) * (0,-1,0))/1000),0) + if((((0,1,1) + (0,1,0)) > (0,0,1) + (0,0,0)),((0,1,0)
+ (0,1,1) - (0,0,0) - (0,0,1)) * (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }
```

FIGURE 11.9 Watershed model specification. (From Ameghino, J., and G. Wainer. 2000. *Proceedings of the 32nd SCS Summer Computer Simulation Conference,* Vancouver, Canada.)

Exercise 11.5

Modify the shape of the terrain and the rate of rain per hour; execute the simulation. Analyze the results obtained.

The original model in Ameghino and Wainer [4] assumed the soil in the whole watershed area was of the same type. A new model originally presented in Ameghino, Troccoli, and Wainer [8] and found in *./watershed2.zip* defines areas with different soil types: one area has vegetation and the other has a rocky soil (Figure 11.11).

The value for a surface 0 cell represents the height of accumulated water and the one for a surface 1 cell represents the ground elevation. These values for ground elevation do not change throughout the simulation, and they are used to calculate the water overflow to neighbor cells. The figure shows that the model includes two zones, each representing the cells that will model vegetation and rock areas. For each zone, different sets of rules apply. Each rule calculates the new water height by applying the hydrology model equation. These rules represent the water accumulation changing the surface vegetation and ground filtration parameters shown in Figure 11.9. Figure 11.12 shows the execution results for this model.

Despite the shape of the original topology, water accumulates in the left part of the terrain faster than in the right part. This is due to the rocky soil defined in the rightmost area, which rejects most of the water. The center part of the figure has the higher filtration of the area due to the lack of vegetation.

Exercise 11.6

Modify the shape of the terrain, the rate of rain per hour, and the areas with vegetation and rocky soil; execute the simulation. Analyze the results obtained.

11.5 POLLUTION MODELS

Bianchini, Indovina, and Rinaldi [9] presented a model on the contamination of the Venetian lagoon produced by substances like nitrogen and phosphorus. The goal is to learn about these properties in order to be able to control these substances in an ecosystem formed by various lakes because of industry influence. This permits study of how fauna and flora are affected by these substances; several species do not resist pollution, and they tend to disappear, producing a change in the ecosystem.

We reproduced this model using Cell-DEVS [2], as we will show in this section. The conceptual idea is to define the diffusion of the polluting substance because of the water flow, which is

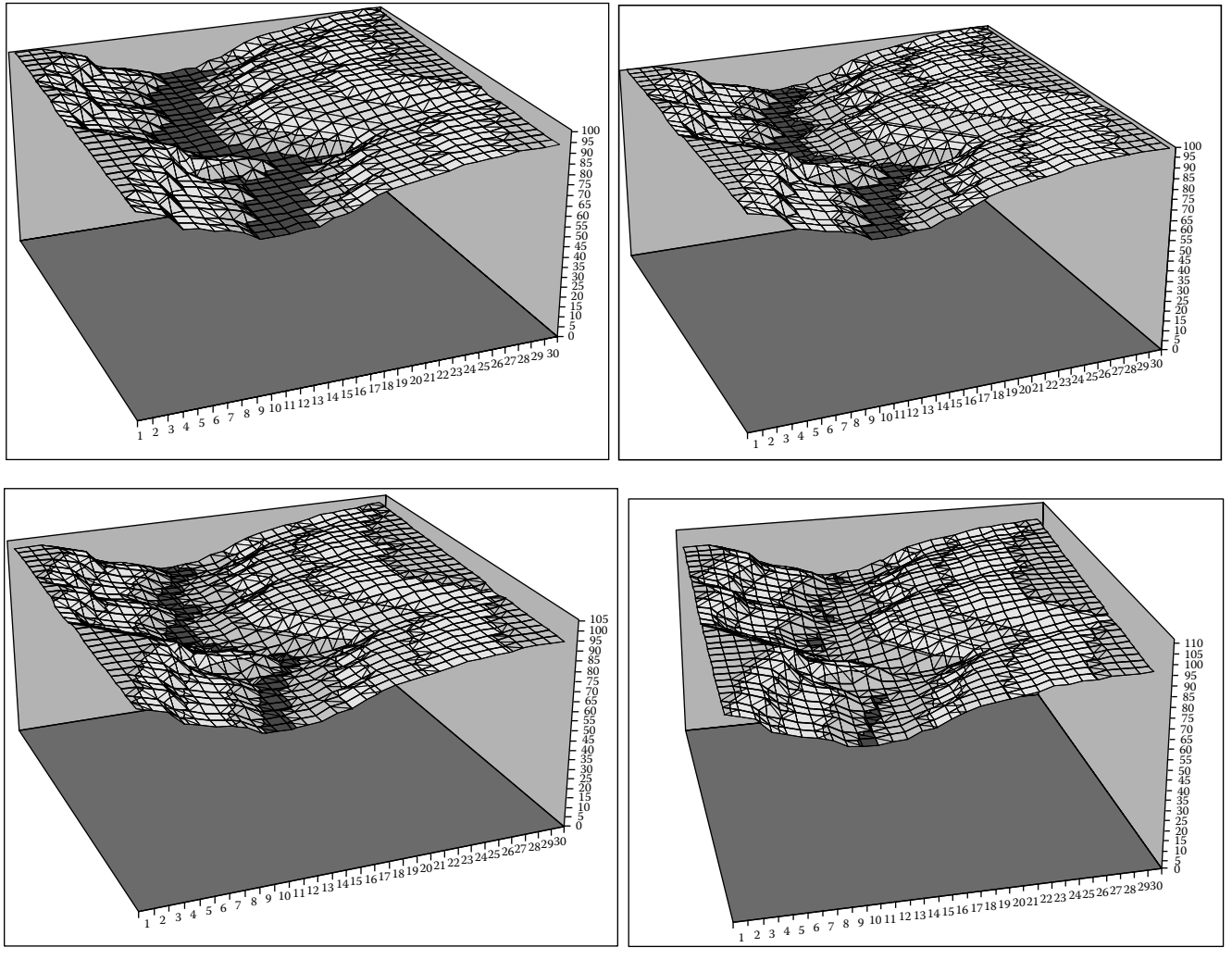


FIGURE 11.10 Initial height values for a watershed; height values after rain.

```
[Watershed]
type : cell                dim : (30,30,2)
delay : inertial           border : nowrapped
neighbors : (-1,0,0) (0,-1,0) (0,0,0) (0,1,0) (1,0,0)(-1,0,1) (0,-1,1) (0,0,1)(0,1,1)
(1,0,1)
zone : vegetation { (0,0,0)..(29,10,0) }
zone : stones { (0,20,0)..(29,29,0) }
localtransition : Hydrology
[vegetation]
rule : {0.07 + (0,0,0) - if(((((0,0,1) + (0,0,0))>((-1,0,1) + (-1,0,0)))),(((((0,0,0) +
(0,0,1) - (-1,0,0) - (-1,0,1))/1000) * (0,0,0))/1000),0) - if(((((0,0,1) +
(0,0,0))>((1,0,1)  +  (1,0,0)))),(((((0,0,0) + (0,0,1) - (1,0,0) - (1,0,1))/1000) *
(0,0,0))/1000),0) - if(((((0,0,1) + (0,0,0))>((0,-1,1)+(0,-1,0)))),(((((0,0,0) + (0,0,1)
- (0,-1,0) - (0,-1,1))/1000) * (0,0,0))/1000),0) - if(((((0,0,1) + (0,0,0))>((0,1,1) +
(0,1,0)))),(((((0,0,0) + (0,0,1) - (0,1,0) - (0,1,1))/1000) * (0,0,0))/1000),0) +
if(((((-1,0,1) + (-1,0,0))>((0,0,1) + (0,0,0)))),((((-1,0,0) + (-1,0,1) - (0,0,0) -
(0,0,1)) * (-1,0,0))/1000),0) + if(((((1,0,1) + (1,0,0))>((0,0,1) + (0,0,0)))),((((1,0,0)
+ (1,0,1) - (0,0,0) - (0,0,1)) * (1,0,0))/1000),0) + if(((((0,-1,1) + (0,-1,0))>((0,0,1)
+ (0,0,0)))),((((0,-1,0) + (0,-1,1) - (0,0,0) - (0,0,1)) * (0,-1,0))/1000),0) +
if(((((0,1,1) + (0,1,0))>((0,0,1) + (0,0,0)))),((((0,1,0) + (0,1,1) - (0,0,0) - (0,0,1))
* (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }

[stones]
rule : {0.09 + (0,0,0) - if(((((0,0,1) + (0,0,0))>((-1,0,1) + (-1,0,0)))),(((((0,0,0) +
(0,0,1) - (-1,0,0) - (-1,0,1))/1000) * (0,0,0))/1000),0) - if(((((0,0,1) +
(0,0,0))>((1,0,1)  +  (1,0,0)))),(((((0,0,0) + (0,0,1) - (1,0,0) - (1,0,1))/1000) *
(0,0,0))/1000),0) - if(((((0,0,1) + (0,0,0))>((0,-1,1)+(0,-1,0)))),(((((0,0,0) + (0,0,1)
- (0,-1,0) - (0,-1,1))/1000) * (0,0,0))/1000),0) - if(((((0,0,1) + (0,0,0))>((0,1,1) +
(0,1,0)))),(((((0,0,0) + (0,0,1) - (0,1,0) - (0,1,1))/1000) * (0,0,0))/1000),0) +
if(((((-1,0,1) + (-1,0,0))>((0,0,1) + (0,0,0)))),((((-1,0,0) + (-1,0,1) - (0,0,0) -
(0,0,1)) * (-1,0,0))/1000),0) + if(((((1,0,1) + (1,0,0))>((0,0,1) + (0,0,0)))),((((1,0,0)
+ (1,0,1) - (0,0,0) - (0,0,1)) * (1,0,0))/1000),0) + if(((((0,-1,1) + (0,-1,0))>((0,0,1)
+ (0,0,0)))),((((0,-1,0) + (0,-1,1) - (0,0,0) - (0,0,1)) * (0,-1,0))/1000),0) +
if(((((0,1,1) + (0,1,0))>((0,0,1) + (0,0,0)))),((((0,1,0) + (0,1,1) - (0,0,0) - (0,0,1))
* (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }
```

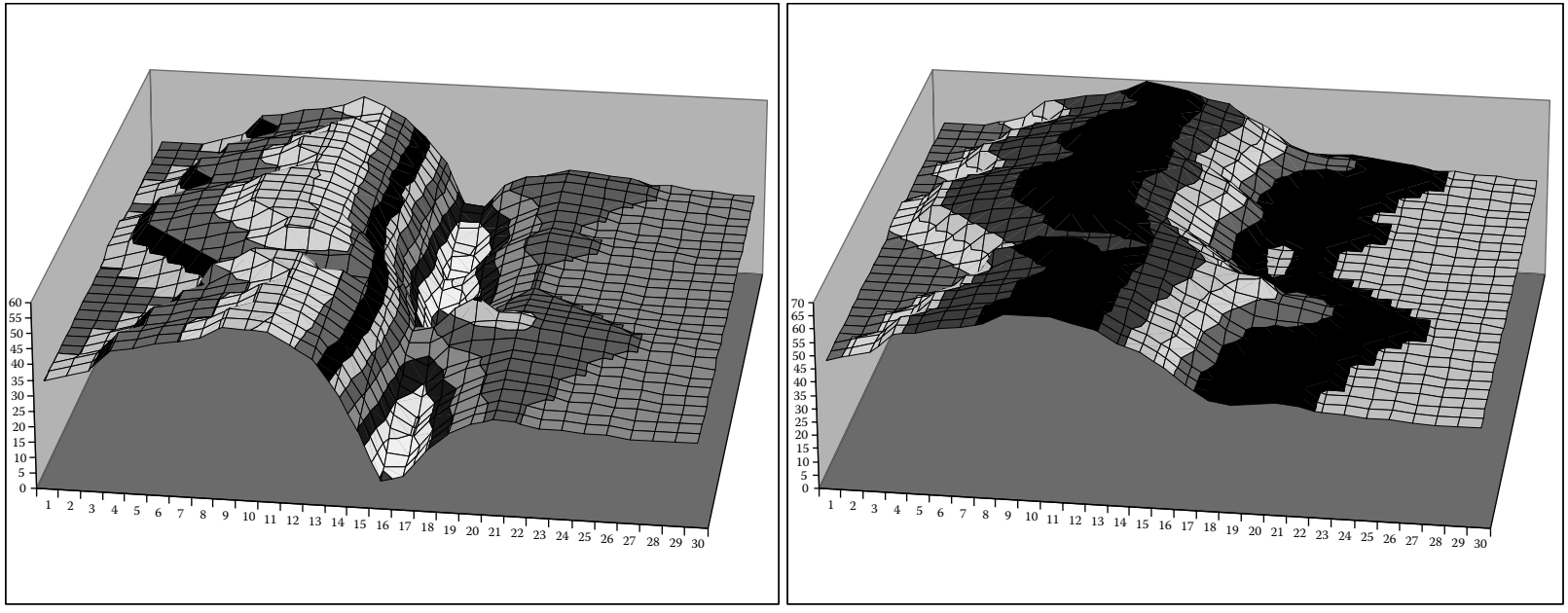
FIGURE 11.11 Specification of a watershed model.

determined by a velocity field. Each cell contains the speed of the water flowing in the cell (and its direction) and the level of contamination of the cell. Pollution is produced when the lake receives nitrogen from the exterior. The rules in the model represent how the contamination spreads.

To do so, the model uses multiple layers:

- Layer 0 is the level of pollution, and it contains information about the subsurface vegetation in the lagoon.
- Layer 1 contains information about the rules to be applied: if it contains the value 1, convection; if it is 10, diffusion; if it is 20, absorption rules should be executed.
- Layer 2 contains hydrological information of the lagoon (the flow can go to 1: N; 2: E; 3: S; and 4: W, or be nonmoving: 0).
- Layer 3 contains the translation density of the substance (a value between 0 and 1 representing the water speed).

When contamination is detected, convection rules are triggered. After this, the diffusion and absorption rules are activated, and then the convection rules can be triggered again. Each cell in the model represents 1 m^2, and 300 ms represent 1 h of simulation (the examples we executed are equivalent to 4 days of simulated time). Each contamination unit is 1 L of pollutant. There are two contamination intakes: the first receives 560 L of pollutant throughout the model's execution, and the second represents an accident in which 40 L of pollutant are received. The model, found in *./pollution.zip,* can be defined as seen in Figure 11.13.



(a) (b)

FIGURE 11.12 (a) Original topology; (b) the watershed after rainwater has accumulated.

```
[top]
components : pollution
in : inPort1 inPort2
link : inPort1 inputNitrogen1@pollution
link : inPort2 inputNitrogen2@pollution

[pollution]
type : cell
dim : (20,20,4)
delay : inertial
border : nowrapped

neighbors : (-1,-1,0)(-1,0,0)(-1,1,0)(0,-1,0)(0,0,0)(0,1,0)(1,-1,0)(1,0,0)(1,1,0)(0,0,1)

neighbors : (-1,0,2)(0,-1,2)(0,0,2)(0,1,2)(1,0,2)(-1,0,3)(0,-1,3)(0,0,3)(0,1,3)(1,0,3)


in : inputNitrogen1 inputNitrogen2
link : inputNitrogen1 in@pollution(1,1,0)
link : inputNitrogen2 in@pollution(5,19,0)

localtransition   : calculus
portInTransition  : in@pollution(1,1,0) setPollution1
portInTransition  : in@pollution(5,19,0) setPollution2

[calculus]
rule : {  (0,0,0)-
          round((if((0,0,2)=1,((20 - (-1,0,0))/20)*(0,0,0)*(0,0,3),0)+
          if((0,0,2)=2,((20 - (0,-1,0))/20)*(0,0,0)*(0,0,3),0)+
          if((0,0,2)=3,((20 - (1,0,0))/20)*(0,0,0)*(0,0,3),0)+
          if((0,0,2)=4,((20 - (0,1,0))/20)*(0,0,0)*(0,0,3),0)))+
          round(max(((if((-1,0,2)=3,(-1,0,3)*(-1,0,0),0)+
          if((1,0,2)=1,(1,0,3)*(1,0,0),0)+
          if((0,-1,2)=4,(0,-1,3)*(0,-1,0),0)+
          if((0,1,2)=2,(0,1,3)*(0,1,0),0))*((20 - (0,0,0))/20)),0)) }
          100 { cellpos(2)=0 and (0,0,1)=0 }     %convection rule

rule : {  round(((-1,-1,0)+(-1,0,0)+(-1,1,0)+(0,-1,0)+(0,0,0)+(0,1,0)+
          (1,-1,0)+(1,0,0)+(1,1,0))/9) }
          100 { cellpos(2)=0 and (0,0,1)=10 }   %diffusion


rule : 20 100 {cellpos(2)=1 and (0,0,0)=10}
rule : 10 100 {cellpos(2)=1 and (0,0,0)=0}
rule : 0  100 {cellpos(2)=1 and (0,0,0)=20}

[setPollution1]
rule : { 20 } 0 { t }

[setPollution2]
rule : { 20 } 0 { t }

[vegetation]
rule : {  (0,0,0)-
          round((if((0,0,2)=1,((20 - (-1,0,0))/20)*(0,0,0)*((0,0,3)/2),0)+
          if((0,0,2)=2,((20 - (0,-1,0))/20)*(0,0,0)*((0,0,3)/2),0)+
          if((0,0,2)=3,((20 - (1,0,0))/20)*(0,0,0)*((0,0,3)/2),0)+
          if((0,0,2)=4,((20 - (0,1,0))/20)*(0,0,0)*((0,0,3)/2),0)))+
          round(max(((if((-1,0,2)=3,((-1,0,3)/2)*(-1,0,0),0)+
          if((1,0,2)=1,((1,0,3)/2)*(1,0,0),0)+
          if((0,-1,2)=4,((0,-1,3)/2)*(0,-1,0),0)+
          if((0,1,2)=2,((0,1,3)/2)*(0,1,0),0))*((20 - (0,0,0))/20)),0)) }
          100 { cellpos(2)=0 and (0,0,1)=0 }
```

FIGURE 11.13 Pollution model definition.

As we can see, the following rules are applied [9]:

1. The value of the current cell is computed as $(0,0) + \sum [((20 - (0,0))/20) \times S_i \times W_i]$, in which the addition is carried out in all of the cells influencing the cell (0,0). In this case, S_i represents the concentration of the pollutant in the cell i, W_i the speed of the cell, and $S_i \times W_i$

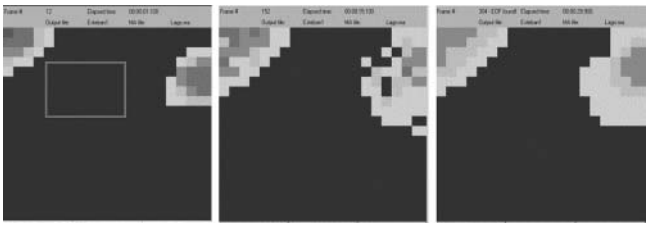


FIGURE 11.14 Two sources of constant pollution (vegetation in marked area).

the contribution of a neighboring cell in the direction to cell (0,0). Finally, (20 – (0,0))/20 represents the reception capacity of the current cell.

2. After evaluating rule 1 and after consuming a delay representing the pollution rate, the new value of contamination of the cell is computed as $(0,0) - [\,(20 - S_i / 20) \times (0,0) \times W_C]$, where W_C is the velocity on the central cell, S_i represents the concentration of the pollutant in cell i (the cell receiving pollution), and $(20 - SC(t))/20$ is the capacity of reception of the receiving cell.
3. After evaluating rules 1 and 2, the cell waits for the delay time. Then we consider the cases in which one or more velocity vectors in the surrounding cells point in the direction of the cell, and speed in the origin cell is zero. This is computed as $(0,0) + \sum [\,((20 - (0,0))/20) \times S_i \times W_i] - [\,(20 - S_i/20) \times (0,0) \times W_C]$.
4. If rules 1–3 are not executed (i.e., there is no neighboring cell pointing to the origin, and the speed of water in the cell is not affecting the origin cell), the current pollution value is maintained.

In Figures 11.14–11.16, we distinguish water (black cells) and pollution (represented in light gray; darker cells are contaminated). In the first example presented in Figure 11.14, the model receives pollution from two different sources. The simulation results in a continuous focus on pollution during several hours (the factories discharge 560 L/h of pollutant). We can see how the pollutant concentrates in the places where it is being discharged. The velocity field and the presence of the subsurface vegetation allow stationary behavior, so diffusion is slow. As we can see, the differences between the second and third graphic (which represent 24 h of simulated time) are not large.

Exercise 11.7

Modify the cellular model and increase the level of pollution generated.

Figure 11.15 shows the results obtained when vegetation in the model is eliminated. As we can see, the pollution concentrates and expands more on the leftmost part of the model than in the

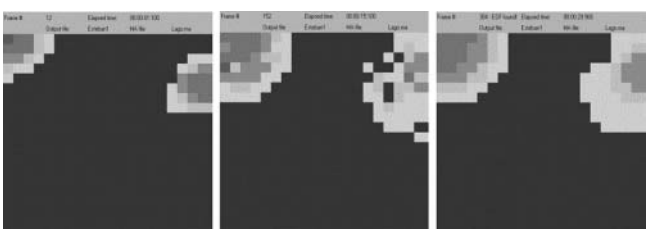


FIGURE 11.15 Two sources of constant pollution and no subsurface vegetation.

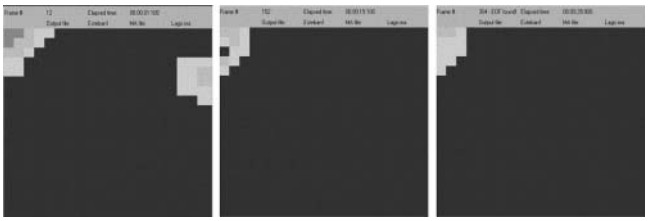


FIGURE 11.16 Behavior in a pollution accident.

previous case. This is a consequence of the lack of vegetation (because the presence of plants favors the reduction of pollution in the area). The northeast side of the figure is the same as in the previous case. Figure 11.16 presents a single input of pollution at the beginning of the simulation (representing an accident in the plant and showing how the toxic elements will spread in the case of accidents).

Because vegetation is in the lake, when the source of pollutant stops, the contamination is slowly absorbed by natural factors, which degrade the contamination up to disappearing in most of the lake. We can see that even the vegetation collaborates in eliminating the pollution, due to the hydrological characteristics of the lake; there is a contamination region on the northwest area that does not disappear or is absorbed by vegetation.

11.6 SIMULATING VEGETATION DYNAMICS

We now discuss the definition of dynamics of vegetation population, based on the work defined by Bandini and Pavesi [10]. In this case, sunlight, water, and fertilizers are factors that influence the growth of vegetation, whose competitive nature in acquiring resources for survival is depicted in the model. Each cell represents a given portion of the yard and contains multiple resources. If conditions are favorable, the cell can host a tree; otherwise, it will be empty. The tree may grow, survive, reproduce, or die, depending on the conditions. Trees growing closer to each other may have the disadvantage of losing nutrients to their competitors. Environmental factors that may affect the growth, such as rain and fauna, are also accounted for.

At each update, the tree in the cell takes the nutrients needed and uses them to grow and produce seeds. If more nutrients are available, the tree stores them. Conversely, if not enough nutrients are present, then the tree uses the stored nutrients. A tree dies if it has no stored nutrients and none are available. There can be only one tree on each cell. Trees can produce seeds that scatter in the cell or its neighbors. Each cell produces nutrients but cannot exceed the maximum value. The flow of the nutrient goes from the richer cells to poorer cells, so bigger trees in a particular cell may make use of the nutrient in neighboring cells.

The model, presented in Wainer [2] and found in *./Vegetable.zip,* uses three-dimensional cells to store different information, and each plane uses a hydrological model to determine the diffusion of water and nutrients. Each three-dimensional cell is subdivided into six different planes (Figure 11.17) to represent the attributes of the cell (water, nitrogen, potassium, and other relevant resources).

There are three types of trees: locust, pine, and oak. Thus, a cell with the values {2, 100, 2.4, 1.1, 17, 2.3} has a pine tree, 100 mL of water, 2.4 g of nitrogen, and 1.1 g of potassium available, and it receives 17 J of sunlight. The current tree height is 2.3 m.

We use Moore's neighborhood on every plane except for the *height of the tree* plane, in which we also access the upper planes (water, nitrogen, potassium, and sunlight), which are used to compute the tree's height. The Cell-DEVS atomic model [2] is defined as

$$\text{Vegetation} = < X, Y, \text{S}, delay, \delta_{int}, \delta_{ext}, \tau, \lambda, D> \tag{11.6}$$

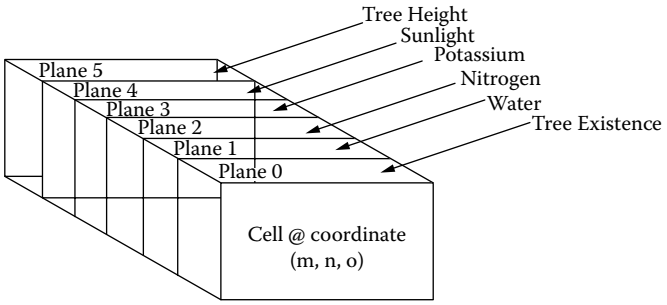


FIGURE 11.17 Cell representation with six planes.

where

X $\subseteq \boldsymbol{R}$ is the set of external input events;
Y $\subseteq \boldsymbol{R}$ is the set of external output events;
S = {R, M, P, T, Z_T, U^G_T, U^S_T, R_T, M_T, G_T, S}, with R = {water, sunlight, nitrogen, potassium}; M = maximum amount of each resource in the cell; P = amount of each resource produced by the cell at each update; T = {0—No, 1—locust, 2—pine, 3—oak}; Z_T = size of the trunk in consideration; U^G_T = vector defining the amount of each resource needed at each update step by the tree to *grow;* U^S_T = vector defining the minimum amount of each resource the tree needs at each update step to *survive*; R_T = amount of resources stored by the tree; M_T = maximum amount of each resource contained by the cell; and S is a vector defining the number of seeds present in the cell for each of the l species growing in the territory;
delay = {transport; 1000 ms}; and
τ = the rules of this model change the vegetation population, using these rules in each zone.

The coupled model is defined as

- *Update-nitrogen:* The nitrogen available in the cell and the presence of a tree are evaluated. Then, the amount required for the growth is subtracted from the available amount, and the nitrogen available in adjacent and current cells is added and equally distributed between neighbors. The concentration changes with the amount of dying vegetation and other external factors. There is a maximum amount of nitrogen for each cell, which is checked during each update.
- *Update-potassium, update-water,* and *update-sun* are similar to the *update-nitrogen* rule.
- *Reproduction:* A seed is dropped in a cell with no tree present. Conditionally, trees in adjacent cells must have reached reproduction age, which is different for different types of trees. The seed survivability is higher for the seeds dropped later in time. Only the seeds dropped later in time survive, regardless of the number of seeds dropped.
- *Update height*: A cell with a tree (or a seed) with enough resources to grow (or sprout) would do so. If there are not enough resources, then the tree dies. The amount of resources required and the growth/death rates are different for different types of trees.

$$\text{VegetationCM} = <X,Y,X_{list},Y_{list},N,\{t1,t2,t3\},C,B,Z> \tag{11.7}$$

where

$X \subseteq$ T is the set of external input events (T = {water, nitrogen, potassium, sunlight});
$Y \subseteq$ T is the set of external output events (T = {growth});

Y_{list} = { (i, j, k)/i ∈ [0,t1], j ∈ [0,t2], k ∈ [0,t3]} is the list of output coupling, where i, j, k represent the index values of the cells (that couple with its neighbors), which are bound by t1, t2, and t3 dimensions;

X_{list} = { (i, j, k)/i ∈ [0,t1], j ∈ [0,t2], k ∈ [0,t3]} is the list of input coupling, where i, j, k represent the index values of the cells (that couple with its neighbors), which are bound by t1, t2, and t3 dimensions; and

N = {(–1,–1,0), (0,–1,0), (1,–1,0), (–1,0,0), (0,0,0), (1,0,0), (–1,1,0), (0,1,0), (1,1,0), (0,0,–1), (0,0,–2), (0,0,–3), (0,0,–4), (0,0,–5), (0,0,1), (0,0,2), (0,0,3), (0,0,4), (0,0,5), (–1,–1,5), (0,–1,5), (1,–1,5), (–1,0,5), (0,0,5), (1,0,5), (–1, 1,5), (0,1,5), (1,1,5)}.

Figure 11.18 shows the model's definition. The first rule presented in the figure checks the amount of nutrients on each layer; if enough nutrients are available, we distribute water by averaging the neighbors (we subtract three units for locust, four for pine, and five for oak). The *adjust* rule adds extra resources at random. The rules for nitrogen and potassium are similar. The *reproduction* rule says that if we have a mature tree of a given species (e.g., `(-1,1,0) = 1` is locust and `(-1,1,5) > 50` is the reproduction age), and the cell is empty, a seed is put into the cell (similar rules exist for pine and oak trees). If there are enough nutrients, the tree grows, as seen in the *growing* rule. If there are not enough nutrients, the tree dies. Figure 11.19 shows the simulation results for this model.

Initially, we have a distribution of nutrients and some seeds on the ground. As we can see in the figure, the concentration of resources changes according to the rules defined for the model, and the trees grow in height as they consume the resources. The resources available change in accordance with the type of the tree in the cell; the tree height and the concentration change accordingly. The cells are gradually updated with the change in concentration of the resources, and the trees grow while resources are consumed. The fourth slice in Figure 11.19 shows trees that have matured to the stage of reproduction. Two of the trees put forth seeds and, in the next stage, they will grow if there are enough resources. The seeds contribute to the growth of new trees in the last slice.

11.7 FOREST FIRES

The spread of fire is a complex phenomenon that many have tried to study over the years. As one can imagine, forest fires depend on many different variables, including the type of fuel, the geography of the area, and the weather. Finding analytical solutions for models of fire spread is almost impossible, so various attempts have been made to use simulation as an alternative. Simulations have been found that accurately represent the way in which fire spreads, and they are now generally the preferred solution for predicting the behavior of wildfires.

In this section, we introduce different methods based on Cell-DEVS that can be used for modeling fire spreading. The models presented use a simple set of equations to determine the temperature of each cell at regular time intervals.

11.7.1 Modeling Fire as a Percolation Process

Percolation theory studies the process of filtration in heterogeneous media, as originally proposed by Broadbent and Hammersley [11], who discovered this phenomenon while studying the reasons for obstruction in the air intake for a gas mask. The main issue was to find out the concentration of waste material needed to produce the obstruction. Originally, it was thought that the relationship between the proportion of blocked holes in the mask and the difficulties for breathing was linear (i.e., the higher the number of blocked holes was, the higher was the lack of air). Nevertheless, they found that, although one could breathe without any problems below a given threshold, if at least 40% of the holes were blocked, the airflow was cut abruptly.

```
[Vegetable]
type : cell
dim : (10, 10,6)
delay : transport
border : wrapped
neighbors : (-1,1,0) (0,1,0) (1,1,0) (-1,0,0) (0,0,0)  (1,0,0) (-1,-1,0) (0,-1,0)
(1,-1,0) (0,0,-1) (0,0,-2) (0,0,-3) (0,0,-4) (0,0,-5) (-1,1,5) (0,1,5) (1,1,5) (-1,0,5)
(1,0,5) (-1,-1,5) (0,-1,5) (1,-1,5) (0,0,1) (0,0,2) (0,0,3) (0,0,4) (0,0,5)
localtransition : update_rule

zone : adjustNitr { (9,0,2)..(9,9,2) }
zone : adjustPota { (9,0,3)..(9,9,3) }
zone : adjustWater { (9,0,1)..(9,9,1) }
zone : adjustSun { (9,0,4)..(9,9,4) }

zone : update_Nitr { (0,0,2)..(8,9,2) }
zone : update_Pota { (0,0,3)..(8,9,3) }
zone : update_Water { (0,0,1)..(8,9,1) }
zone : update_Sun { (0,0,4)..(8,9,4) }

zone : update_height { (0,0,5)..(9,9,5) }
zone : reproduction { (0,0,0)..(9,9,0) }

[update_Water]
% consume Water rule with locust tree growth and Water distribution to adjacent cells
rule : { (0,0,0)-3+((-1,1,0)+(0,1,0)+(1,1,0)+(-1,0,0)+(0,0,0)+(1,0,0)+(-1,-1,0)+(0,-
          1,0)+(1,-1,0))/9 } 1000 { (0,0,3)>=5 and (0,0,1)>=70 and (0,0,2) >= 80 and (0,0,0) >= 5
          and (0,0,-1) = 1 }
...

% if no tree or enough resources available, distribute the resource to adjacent cells.

rule : {((-1,1,0)+(0,1,0)+(1,1,0)+(-1,0,0)+(0,0,0)+(1,0,0)+(-1,-1,0)+(0,-1,0)+(1,-
1,0))/9} 1000 { (0,0,0) < 100  }

[adjustWater]
% add extra resource to this row using uniform distribution and consume Water rule with
    locust tree growth and Water distribution to adjacent cells
rule : { ((0,0,0)- 3) + uniform(5,15) + ((-1,1,0)+(0,1,0)+(1,1,0)+(-
          1,0,0)+(0,0,0)+(1,0,0)+(-1,-1,0)+(0,-1,0)+(1,-1,0))/9 } 1000 { (0,0,3) >= 5 and  (0,0,1)
          >= 70 and (0,0,2) >= 80 and (0,0,0) >= 5 and (0,0,-1) = 1 }

...

[update_Nitr]
% consume Nitro rule with locust tree growth and nitro distribution to adjacent cells
rule : { ((0,0,0)- 18)+ ((-1,1,0)+(0,1,0)+(1,1,0)+(-1,0,0)+(0,0,0)+(1,0,0)+(-1,-1,0)+(0,-
          1,0)+(1,-1,0))/9 } 1000 { (0,0,2) >= 5 and  (0,0,0) >= 70 and (0,0,1) >= 80 and (0,0,-1)
          >= 5 and (0,0,-2) = 1 }

...

[reproduction]
rule : { 1 } 1000 { (0,0,0) = 0 and (-1,1,0)= 1 and (-1,1,5) > 50 }
rule : { 1 } 1000 { (0,0,0) =0 and (-1,0,0)= 1 and (-1,0,5) > 50 }
rule : { 1 } 1000 { (0,0,0) =0 and (-1,-1,0)= 1 and (-1,-1,5) > 50 }
rule : { 1 } 1000 { (0,0,0) =0 and (0,1,0)= 1 and (0,1,5) > 50 }
rule : { 1 } 1000 { (0,0,0) =0 and (0,-1,0)= 1 and (0,-1,5) > 50 }
...

[update_height]
% growing rule for locust
rule : { (0,0,0) + 5 } 1000 { (0,0,-1) >= 5 and  (0,0,-2) >= 70 and (0,0,-3) >= 80 and
          (0,0,-4) >= 5 and (0,0,-5) = 1 and (0,0,0) < 75  }

% dying rule for locust
rule : { 0 } 1000 { (0,0,-1) < 5 or  (0,0,-2) < 70 or (0,0,-3) < 80 or (0,0,-4) < 5 and
          (0,0,-5) = 1 and (0,0,0) >0  }
```

FIGURE 11.18 Cell representation with six planes.

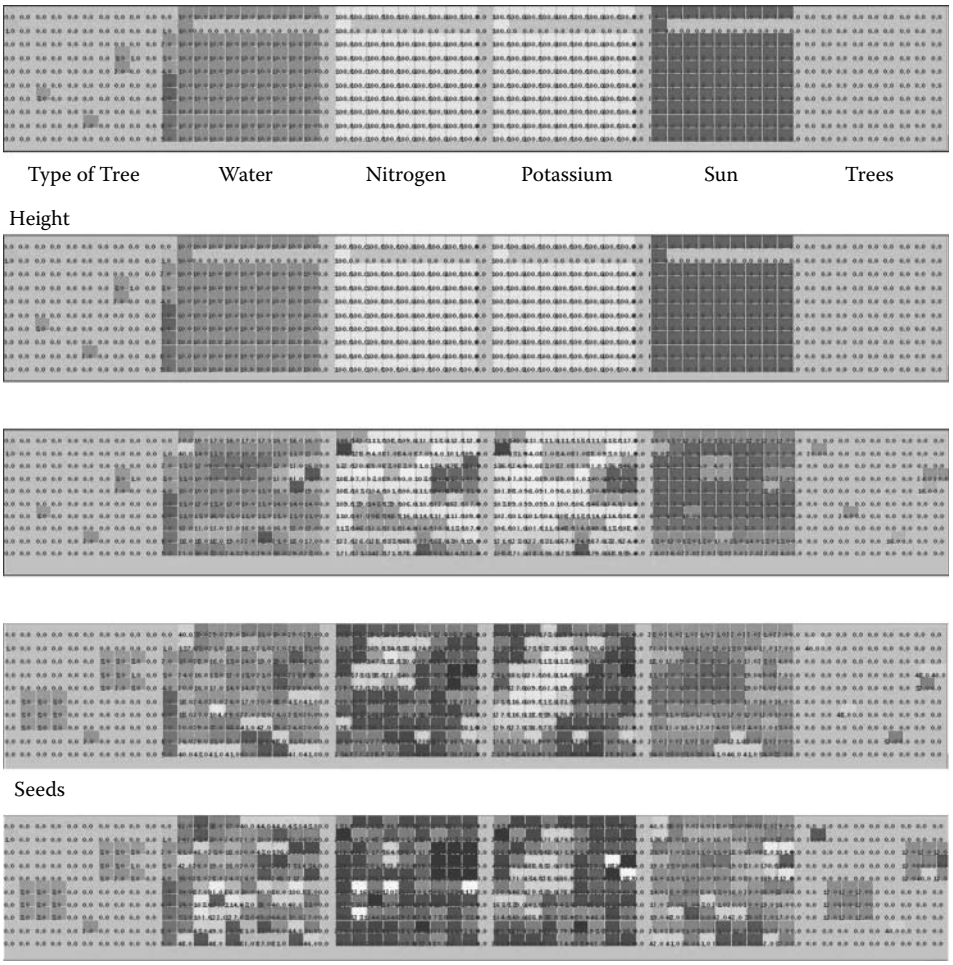


FIGURE 11.19 Vegetation model execution.

The theory of percolation has been used for studying other phenomena (e.g., plague expansion, epidemics, oil accumulation in rocks), modeling the system as a filtration process in heterogeneous media in which there is an element opposed to filtration (e.g., impurities, absence of plants or people). Formally, percolation studies consist of analyzing the different paths available in a two-dimensional grid; if there is a path from one point of the grid to another, percolation can occur. Therefore, the probability for percolation depends on the number of open spaces and blocking elements.

Fire spreading can be modeled as a percolation process. By considering trees as agents able to propagate fire and a piece of soil (or a burned tree) as a blocking agent within the forest, we can apply percolation theory to these problems. In order to permit fire propagation, the trees must be close enough to each other. As we will see in the following examples, the threshold level for this model depends on the kind of neighborhood used (about 60% for von Neumann's and 43% for Moore's neighborhoods). The forest is represented by a grid (100 × 100 cells in our example), where each cell represents a tree or a portion of land. The model, found in *./percol.zip,* is defined as found in Figure 11.20.

The model uses the value 1 for unburned trees, 2 for trees catching fire, 3–7 for a tree burning, and 8 for a burned tree. The first rule in the model makes a burned cell equivalent to an empty one (because it cannot spread fire anymore). Rules 2–7 change the state of the tree. Rule number 8 is in charge of starting the burning process, based on the current state of the cell and the inputs

```
[perco]
type : cell
width : 100
height : 100
delay : transport
border : nowrapped
neighbors : (-1,-1)  (-1,0) (-1,1) (0,-1) (0,0)  (0,1) (1,-1)   (1,0)  (1,1)
localtransition : perco-rule

[perco-rule]
rule : 0 100 { (0,0) = 8 and falsecount = 8}
rule : 8 100 { (0,0) = 7}
rule : 7 100 { (0,0) = 6}
rule : 6 100 { (0,0) = 5}
rule : 5 100 { (0,0) = 4}
rule : 4 100 { (0,0) = 3}
rule : 3 100 { (0,0) = 2}
rule : 2 100 { (0,0) = 1 and (statecount(2) > 0 or statecount(3) > 0 or statecount(4) > 0)}
rule : 1 100 { (0,0) = 1 }
rule : 0 100 { t }
```

FIGURE 11.20 Percolation model.

received from the neighbors: if a tree is not burning and no neighbors are burning, the tree remains unchanged.

The examples in Figures 11.21–11.26 show different simulation results for the forest fire model based on percolation. Our first case has a forest with a density of 40%; according to the generic theoretical results, percolation should not occur. As we can see in Figure 11.21, the fire (whose source is on the right) burns part of the area on the right, but percolation behavior does not occur and the fire has not spread to the rest of the forest.

As we can see in Figure 11.22, by changing the density only 1%, we can observe percolation behavior. Although percolation occurs, about 50% of the trees still survive. Again, by increasing the density only 1%, percolation occurs, and fire burns the forest almost completely (showing the threshold behavior previously discussed; Figure 11.23). Figure 11.24 shows that, independently of the place where the fire originated, the same behavior is observed. Initially, we have a density of 43%,

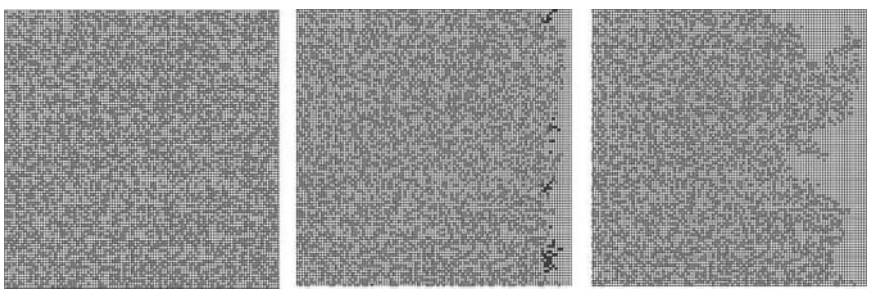


FIGURE 11.21 40% density.

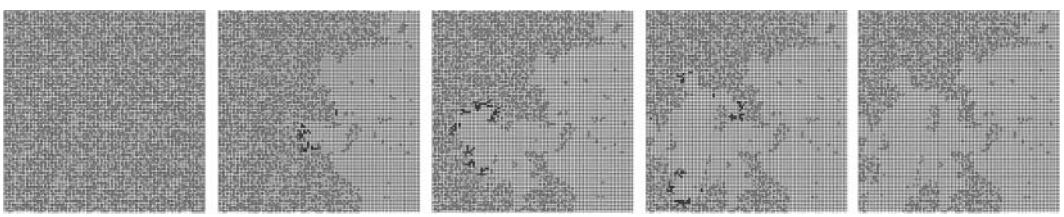


FIGURE 11.22 41% density.

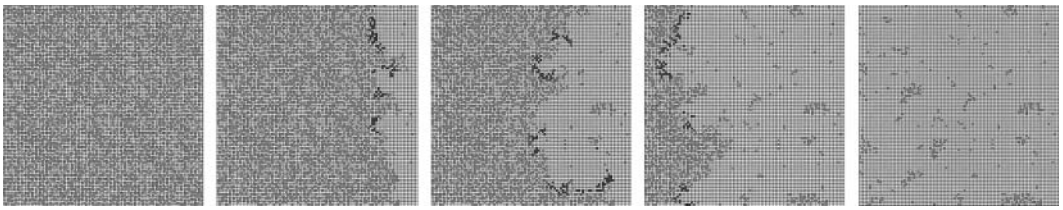


FIGURE 11.23 42% density.

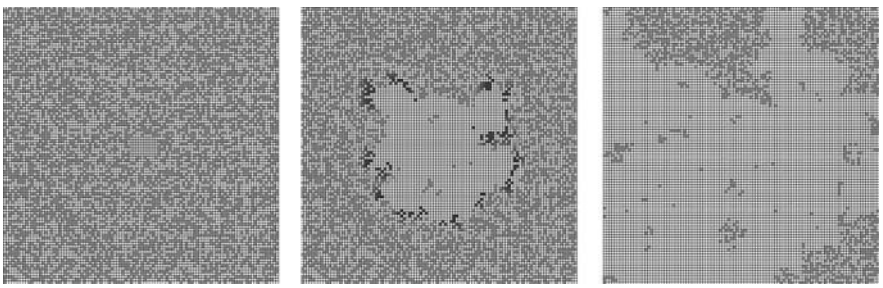


FIGURE 11.24 Fire source in the center, left and right.

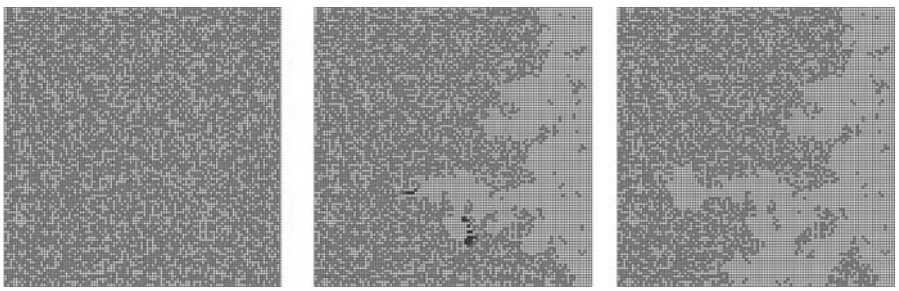


FIGURE 11.25 von Neumann's neighborhood, 57% density.

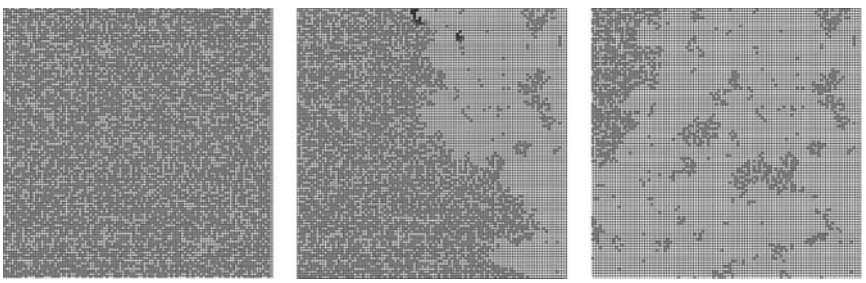


FIGURE 11.26 von Neumann's neighborhood, 59% density.

but in this case, we change the fire source to the left cells and also a few cells to the right and the middle of the forest. The figure shows the center case.

In Figures 11.25 and 11.26, we changed the neighborhood definition and used von Neumann's neighborhood. As discussed earlier, the threshold is now 60% (±2%). In our initial case, where the density is 57%, percolation does not occur. When we change the density to 59%, percolation happens.

11.7.2 Fire Spreading Using Rothermel's Rules

A well-known model for fire propagation in forests is due to Rothermel [12]. Based on environmental and vegetation conditions, it computes the ratio of spread and intensity of fire. Three parameter groups determine the fire spread ratio: (1) the vegetation type (caloric content, mineral content, and density), (2) the fuel properties (the vegetation is classified according to its size), and (3) environmental parameters (wind speed, fuel humidity, and field slope). For the model in this section, we used the NFFL (Northern Forest Fire Laboratory), which classifies vegetation in 13 groups, representing the majority of existing forest types in the region.

When Rothermel's rules are applied to a given fuel model and environmental parameters, the spread ratio (i.e., the distance and direction the fire moves in a minute) can be determined. The first step is to use the fuel model, the speed and direction of the wind, the terrain topology, and the dimensions of the cellular space to obtain the spread ratio in every direction. These values are used to write a specific model for the given parameters using CD++. For instance, Figure 11.27 shows the values obtained for a fuel model group number 9, a southeast wind of 24.135 km/h, and a cell size of 15.24 × 15.24 m.

These parameters were used to write a specific Cell-DEVS model using CD++, which can be found in *./fire.zip*. The specification in Figure 11.28 shows a 20 × 20 Cell-DEVS representing the terrain and vegetation. The state variables of the cells use a 0 value to indicate the absence of fire, and a value different from 0 indicates the time the fire has started on that cell.

```
Wind direction = 45.000000 (bearing)
Wind speed = 8.045000 [kph] NFFL model = 1
Cell Width = 15.240000 [m] (E-W)
Cell Height  = 15.240000 [m] (N-S)
Max. Spread = 17.967136 [mpm]
0° Spread = 5.106976 [mpm] Distance = 15.2400 [m]
45° Spread = 17.967136   Distance = 21.552615
90° Spread = 5.106976    Distance = 15.240000
135° Spread = 1.872060   Distance = 21.552615
180° Spread = 1.146091   Distance = 15.240000
225° Spread = 0.987474   Distance = 21.552615
270° Spread = 1.146091   Distance = 15.240000
315° Spread = 1.872060   Distance = 21.552615
```

FIGURE 11.27 Parameter definition computed using the Rothermel model.

```
[ForestFire]
type : cell              dim : (20,20)
delay : inertial         border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
localtransition : FireBehavior

[FireBehavior]
rule : {(1,-1)+(21.552615/17.967136)} {(21.552615 / 17.967136)*60000} {(0,0)=0 and
       0<(1,-1)}
rule : {(1,0)+(15.24/5.106976)} {(15.24 / 5.106976)*60000} {(0,0)=0 and 0<(1,0)}
rule : {(0,-1)+(15.24/5.106976)} {(15.24 / 5.106976)*60000} {(0,0)=0 and 0<(0,-1)}
rule : {(-1,-1)+(21.552615/1.872060)} {(21.552615 / 1.872060)*60000} {(0,0)=0 and
       0<(-1,-1)}
rule : {(1,1)+(21.552615/1.872060)} {(21.552615 / 1.872060)*60000} {(0,0)=0 and 0<(1,1)}
rule : {(-1,0)+(15.24/1.146091)} {(15.24 / 1.146091)*60000} {(0,0)=0 and 0<(-1,0)}
rule : {(0,1)+(15.24/1.146091)} {(15.24 / 1.146091)*60000} {(0,0)=0 and 0<(0,1)}
rule : {(-1,1)+(21.552615/0.987474)} {(21.552615 / 0.987474)*60000} {(0,0)=0 and
       0<(-1,1)}
rule : {(0,0)} 0 { t }
```

FIGURE 11.28 Definition of a fire forest model.

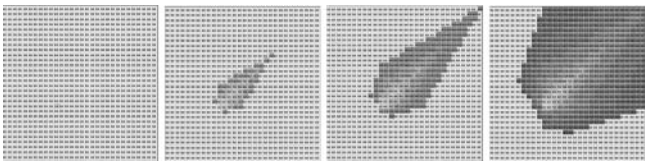


FIGURE 11.29 Fire propagation results in a 2-h period (each zone represents 20 min).

The rules defining the local computing function are devoted to detecting the presence of fire in the eight neighboring cells. If there is fire in one of them, then the current cell will burn. For instance, the first rule checks whether the current cell is not burning (`(0,0) = 0`) and the southwest neighbor has started to burn (`0<(1,-1)`). If this condition holds, the value will be `(1,-1) + (21.552615/17.967136)`, which is the time the fire will take to traverse the cell in this direction. Because the spread ratio is `17.967136` mpm and a cell has a diagonal of `21.552615` m, it will take `21.552615/17.967136` min for the fire to reach a cell once it has started in its southwest neighbor. Therefore, we use a delay of `(21.552615/17.967136)*60000` ms, after which the present cell state will spread to the neighbors. The results of running this model are shown in Figure 11.29; the behavior is similar to that discussed previously by Vasconcelos [13] and Vasconcelos, Pereira, and Zeigler [14].

As we can see, the burning time of a cell depends on the spread ratio in the direction of the burning cell. This value is used as the delay component for the rules. It is important to notice that the cells are updated at different times, as set by a rule's delay component (a nonburning cell in the direction of the fire spread will be updated in a shorter period than a nonburning cell in the opposite direction).

Exercise 11.8

Using the FireLib library application found at http://www.fire.org/, find the parameters for different fuel (NFFL) models using the same initial parameters of Figure 11.27. Change the implementation of the model according to the new computed arguments.

Exercise 11.9

Repeat Exercise 11.8 by changing the wind speed and direction.

Exercise 11.10

Incorporate the FireLib equations in each cell of the model, using the mechanism shown in the appendix in Chapter 8, and execute the model directly.

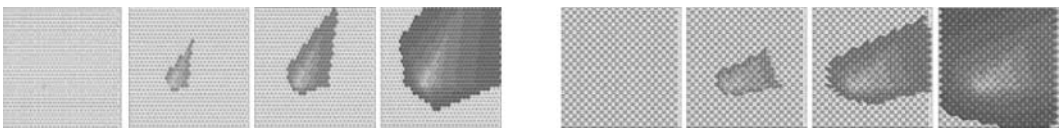
As discussed in Chapter 4, hexagonal Cell-DEVS models can be built in CD++ and then translated using the lattice translator. Figure 11.30 shows the implementation of the rules for this model using a hexagonal mesh. In this case, the first rule checks if the current cell is not burning ([0] = 0) and if the southwest neighbor has started to burn ([5] > 0). If this condition holds, the new value of the cell will be `[5]+(15.24/13.680)`. This is used in all the remaining rules (because, in this case, fire spreading in every direction is symmetric; in the hexagonal lattice, the distance between two neighbor cells is the same in every direction, so we use a distance of 15.24 m for all of the rules). As

```
[FireBehavior]
rule: {[5]+(15.24/13.680)} {(15.24/13.680) * 60000} {[0]=0 and [5]!=? and [5]>0}
rule: {[6]+(15.24/5.10)} {(15.24 /5.106 ) * 60000} {[0]=0 and [6]!=? and [6]>0}
rule: {[4]+( 15.24/2.950)} {(15.24/2.950) * 60000} {[0]=0 and [4]!=? and [4]>0}
rule: {[1]+(15.24/1.630)} {(15.24/ 1.630) * 60000} {[0]=0 and [1]!=? and [1]>0}
rule: {[3]+(15.24/1.146)} {( 15.24 / 1.146) * 60000} {[0]=0 and [3]!=? and [3]>0}
rule: {[2]+(15.24/1.040)} {( 15.24/ 1.040) * 60000} {[0]=0 and [2]!=? and [2]>0}
```

FIGURE 11.30 Rothermel's forest fire model using a hexagonal mesh.

```
[FireBehavior]
rule: {[3]+(4.40/5.106)} {(4.40/5.106) * 60000} {[0]=0 and [3]!=? and [3]>0 and
       odd(cellpos(0)+cellpos(1))}
rule: {[1]+(4.40/2.950)} {(4.40/2.950) * 60000} {[0]=0 and [1]!=? and [1]>0 and
       odd(cellpos(0)+cellpos(1))}
rule: {[2]+(4.40/1.040)} {(4.40/1.040) * 60000} {[0]=0 and [2]!=? and [2]>0 and
       odd(cellpos(0)+cellpos(1))}
rule: {[3]+(4.40/8.573)} {(4.40/ 8.573) * 60000} {[0]=0 and [3]!=? and [3]>0 and
       even(cellpos(0)+cellpos(1)}
rule: {[2]+(4.40/1.630)} {(4.40/1.630) * 60000} {[0]=0 and [2]!=? and [2]>0 and
       even(cellpos(0)+cellpos(1)}
rule: {[1]+( 4.40/1.146)} {(4.40/1.146) * 60000} {[0]=0 and [1]!=? and [1]>0 and
       even(cellpos(0)+cellpos(1)}
```

FIGURE 11.31 Rules using triangular topology.



(a) (b)

FIGURE 11.32 Fire propagation results (2-h period): (a) hexagonal lattice; (b) triangular lattice.

discussed in Chapter 4, we use a different notation to represent each of the six neighbors ([1]…[6] in a counterclockwise direction starting at 0°). We can also create the triangular models whose implementation is shown in Figure 11.31.

In this case, there are six rules because we need rules for even and odd triangles. For both triangular and hexagonal models, the rules are translated into a square grid, as shown earlier in Chapter 4. The simulation results of these models are shown in Figure 11.32. The burning time of a cell depends on the spread ratio in the direction of the burning cell. Changes in the propagation here are related to the changes produced by the adjacency properties derived from using different topologies.

11.7.3 Fire Suppression Definition

Ameghino and Wainer [4] discussed simple mechanisms for fire suppression. Ntaimo et al. [15] discussed different techniques for suppression in forest fire models. Here, we present different mechanisms for fire suppression [4] that show basic techniques to eliminate fire in the models presented in the previous section. In the first case, we define a rainstorm moving to the southeast, extinguishing the fire on burning cells. To allow this behavior, the rules shown in Figure 11.33 were added to the previous model.

We use negative values to represent the effects of rain. A cell whose value is –1 is a wet cell where no fire was presented previously. A value of –2 or –3 indicates the cell was previously on fire and is now cooling down, and a value of –4 means the fire on that cell has been extinguished. The first rule in the previous figure defines rain spreading to the southwest. The second defines the cooling process on a burning cell, and the third and fourth rules represent advance in the cooling process. The model assumes that the fire on a cell will take 16 min to extinguish (in stages of different length). The simulation results of this model, found in *./fireandrain.zip,* can be seen in

```
rule : -1 {60000*3}   {(0,0)=0 and ((-1,0)=-1 or (0,1)=-1 or (-1,0)=-2 or (0,1)=-2)}
rule : -2 {60000*3.5} {(0,0)>0 and ((-1,0)=-1 or (0,1)=-1 or (-1,0)=-2 or (0,1)=-2)
rule : -3 {60000*4.5} {(0,0)=-2}
rule : -4 {60000*5}   {(0,0)=-3}
```

FIGURE 11.33 Rules defining rain.

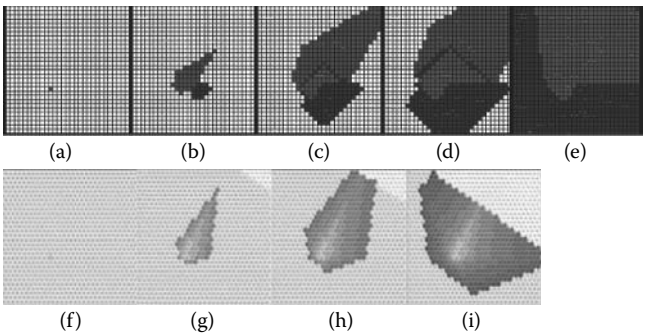


FIGURE 11.34 Fire evolution with rain: (a) start; (b) fire advance and rain; (c, d) rain cooling fire areas; (e) rain extinguished fire areas; (f–i) rain coming from the NE-SE in a hexagonal model.

Figure 11.34. The basic model is the same as that presented in Figure 11.29, including the influence of rain (in dark gray). We can see rain cooling the fire areas (in gray), and after a while the rain has extinguished fire in some areas (in light gray). Figure 11.34(f–i) shows the execution of the same model when we use a hexagonal topology, as defined in Figure 11.31.

It is important to notice that if any of the cells are scheduled to start burning and get wet before the fire starts, they will not burn. This was easily defined by an inertial delay, which preempts any scheduled event if a new event from a neighbor cell arrives before the scheduled time and the present cell gets a different value.

Exercise 11.11

Modify the model to change the direction of rain.

Exercise 11.12

Modify the model defined in Exercise 11.8 to include rain influence.

A second suppression technique allows us to analyze the influence of firefighters. A negative value is still used for wet or cooling cells and a positive value is used for burning cells, but the way in which the water is spread has been changed, as seen in Figure 11.35, which is defined in *./fireandf.zip*. In this case, firefighters move from north to south spreading water to nonburning vegetation. Once they reach a burning cell, they will hold their positions until the fire is extinguished, and then they will move toward the southwest, as shown in Figure 11.36. The figure shows how firefighters spread coolant from north to south and how, while fire is still spreading, in the zones where firefighters are working (light gray), fire is extinguished.

Exercise 11.13

Analyze the fire suppression techniques presented in Ntaimo et al. [15] and define such a fire suppression model using Cell-DEVS and CD++.

```
rule : -1 60000     {(0,0)=0 and (-1,0)=-1}
rule : -2 {60000*7} {(0,0)>0 and ((-1,1)=-1 or (-1,1)=-4) }
rule : -3 {60000*9} {(0,0)=-2}
rule : -4 {60000*9} {(0,0)=-3}
```

FIGURE 11.35 Rules defining firefighter behavior.

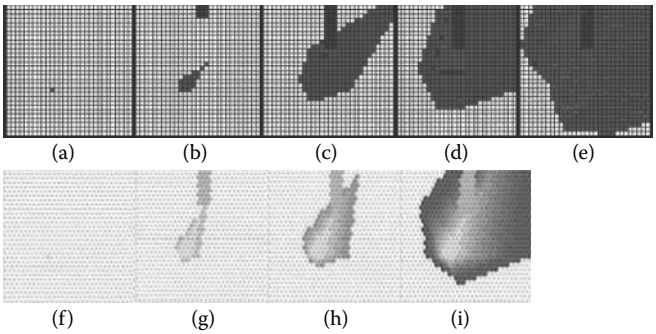


FIGURE 11.36 Fire evolution and firefighters: (a) start; (b) fire suppression from north to south; (c) fire spreading, firefighter zones cooled down (light gray); (d, e) areas of fire extinguished; (f–i) same model with hexagonal cells.

11.7.4 A Semiempirical Model

In this section, we present a model based on the research introduced in references 16–18. In order to understand how to model fire at large scale, the authors modeled fire spread across a 1-m^2 experimentation bed with pine needles fuel (without considering the influence of wind slope) [18]. The original study used elementary cells composed of earth and plant matter, and the energy transferred from the cell to the surrounding air was considered proportional to the difference between the temperature of a cell and the ambient temperature. In order to model the combustion reaction, the authors assumed that combustion occurs above a threshold temperature T_{ig}. Above this threshold, the fuel mass decreases exponentially, and the quantity of heat generated by the combustion reaction per unit fuel mass is constant. This can be represented by the following equations:

$$\frac{\partial T}{\partial t} = -k(T - T_a) + K\Delta T - Q\frac{\partial \sigma_v}{\partial t} \text{ in the domain} \tag{11.8a}$$

$$\frac{\partial \sigma_v}{\partial t} = 0 \text{ for an inert cell} \tag{11.8b}$$

$$\frac{\partial \sigma_v}{\partial t} = -\alpha\sigma_v \text{ for a burning cell} \tag{11.8c}$$

$$T(x,y,t) = T_a \text{ at the boundary} \tag{11.8d}$$

$$T(x,y,0) = T_a \text{ for the nonburning cells at } t = 0 \tag{11.8e}$$

$$T(x,y,0) = T_{ig} \text{ for the burning cells at } t = 0 \tag{11.8f}$$

In Figure 11.37, we show the temperature curve of a cell in the domain and its associated phases, based on the previous equations.

Muzy and colleagues [17] presented two numerical methods that can be used to discretize the model based on finite elements and finite difference methods. The discretized version uses the following algebraic equation:

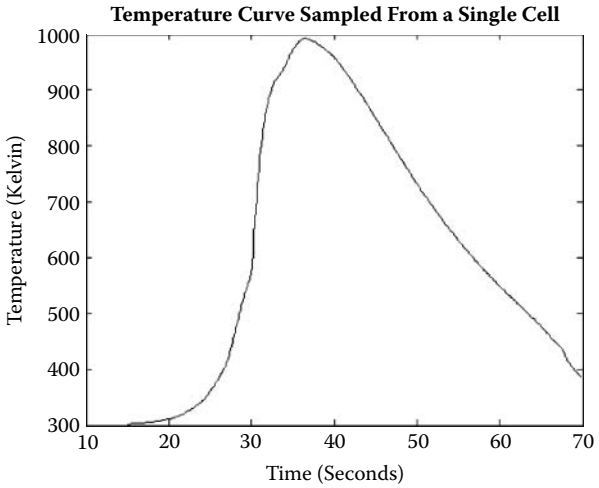


FIGURE 11.37 Temperature curve of a cell of the domain.

$$T_{i,j}^{k+1} = aT_{i-1,j}^{k} + aT_{i+1,j}^{k} + bT_{i,j-1}^{k} + bT_{i,j+1}^{k} + cQ\left(\frac{\partial \sigma_v}{\partial t}\right)_{i,j}^{k+1} + dT_{i,j}^{k} \tag{11.9}$$

where T_{ij} is the temperature of a grid node. The coefficients *a, b, c,* and *d* depend on the considered time step and mesh size.

We used CD++ to create a Cell-DEVS version of this model, which can be found in *./fireCorse.zip.* The model, presented in Muzy et al. [16], can be described as seen in Figure 11.38. Each cell evolves through four phases: *inactive, unburned, burning,* and *burned.* We consider that, above a threshold temperature T_i, there is combustion; when temperature is below T_f, the combustion finishes (we voluntarily neglect the end of the real curve to save simulation time). We use two planes to model fire spreading: plane 0 stores the cell temperatures, and plane 1 stores the ignition time for each of the cells.

```
[ForestFire]
dim : (100,100,2)    border : nowrapped
neighbors : (-1,0,0) (0,-1,0) (1,0,0) (0,1,0)(0,0,0)(0,0,-1)(0,0,1)
zone : ti { (0,0,1)..(99,99,1) }
localTransition : FireBehavior

[ti]
rule:{ time/100 } 1 { cellpos(2)=1 AND (0,0,-1)>=573 AND (0,0,0) = 1.0 }

[FireBehavior]
rule: {#unburned} 1 {(0,0,0)<300 AND (0,0,0)!=26 AND (#unburned>(0,0,0) OR time<=20)} %Unburned
rule: {#burning} 1 {cellpos(2)=0 AND ( ((0,0,0) > #burning AND (0,0,0)>333) OR (#burning> (0,0,0)
       AND (0,0,0)>=573) ) AND (0,0,0)!=209 } %Burning
rule: {26} 1 { (0,0,0)<=60 AND (0,0,0)!=26 AND      (0,0,0)>#burning } %Burned
rule : { (0,0,0) } 1 { t }   %Stay Burned or constant

#BeginMacro(unburned)
(0.98689 * (0,0,0) + 0.0031 * ( (0,-1,0) + (0,1,0) + (1,0,0) + (-1,0,0) ) + 0.213 )
#EndMacro

#BeginMacro(burning)
(0.98689*(0,0,0)+.0031*((0,-1,0)+(0,1,0)+(1,0,0)+ (-1,0,0))+2.74*exp(-.19*((time+1)*.01-
     (0,0,1)))+.213)
#EndMacro
```

FIGURE 11.38 Fire spread model specification and model macros.

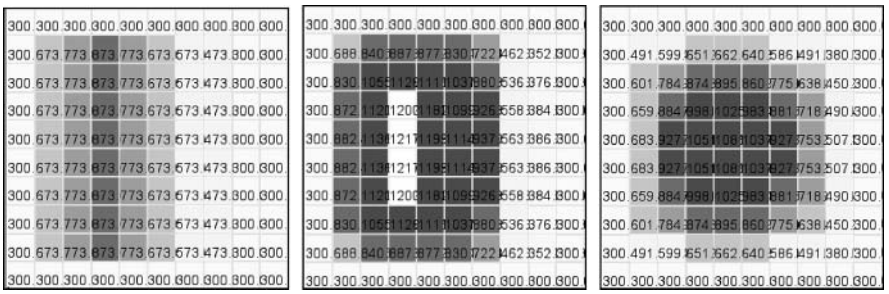


FIGURE 11.39 Simulated temperatures field representation of a line ignition.

The *ti* rules show how to store ignition times: if a cell in plane 0 burns, we record the current simulation time in plane 1 (computed as the simulation time multiplied by the time step). Temperature is computed in plane 0; different rules are used for the temperature calculus if cells can be inactive, unburned, burning, and burned (using Equations 11.8a–11.8f); and macros are used to make the definition more compact.

The first rules in the figure correspond to the phase *unburned,* whose cell's temperature is lower than 573°C. If the cell belongs to the plane 0 and its temperature at the next time step is greater than the current one, the cell will take the value given by the macro *unburned* (we ignore the transient period below 20 time units, burning and burned cells). A cell starts burning at 573°C, and its temperature increases as the fuel mass is consumed; then it starts decreasing. When the temperature goes below 333°C, the cell enters the *burned* phase (signaled by a constant temperature of 209°C). The first rule in Figure 11.38 applies to unburned cells, whose temperature in the next step will be higher than its current one. The second rule applies to burning cells. The third rule sets the burned flag (temperature = 209°C) if a burning cell goes below 333°C, and the fourth rule keeps the burned cells constant.

Figure 11.39 shows the simulation results for a laboratory experiment using a combustion table of 30 × 60 cm, a homogenous fuel bed of pine needles, no wind, and linear ignition. The prediction of spread rate (2.96 mm/s) and the propagation are in agreement with the experimental data. The darker cells represent the position of the experimental isothermal line of 300°C (ignition interface).

A different implementation of this model was defined in Parallel CD++ [2,19]. In this case, the *temperature* is stored as the cell's value and the ignition time *ti* is stored in a state variable (the cell's values are automatically transmitted to the neighbor cells, while the *ti* value is used internally). The first step was to add a state variable *ti,* to remove the higher layer of cells, and to replace all the references to this layer with references to the state variable. The original *burning* and *ti* rules were replaced as shown in Figure 11.40.

This version of the model can be optimized because CD++ is capable of using shortcut evaluation (in the same style as the C programming language). When the left expression of an *AND* operation evaluates to *false*, the whole operation will evaluate to *false*. Similarly, when the left expression of an *OR* operation evaluates to *true,* the whole operation will evaluate to *true*. Thus, by sorting the operations as shown in Figure 11.41, we can save execution time. This problem can also be solved

```
stateVariables: ti       stateValues: 0

[FireBehavior]
rule : { #unburned } 1 { (0,0)!=209 AND (0,0)<573 AND ( time<=20 OR #unburned>(0,0) ) }
rule : { #burning } 1 { (0,0)>333 AND ( (0,0)<573 OR $ti != 1.0 ) AND   (0,0)>#burning }
rule : { #burning }{ $ti := if($ti = 1.0, time / 100, $ti); } 1
   { (0,0) >= 573 AND #burning >= (0,0) }
rule : { #burning } { $ti := time/100; } 1 { $ti = 1.0 AND (0,0) >= 573 AND #burning < (0,0) }
rule : { 209 } 100 { (0,0) != 209 AND (0,0)<=333 AND (0,0) > #burning }
```

FIGURE 11.40 Fire spread model specification.

```
%Unburned
rule : { #macro(unburned) } 1{ (0,0) != 209 AND (0,0) < 573 AND
         ( time <= 20 OR #macro(unburned) > (0,0) ) }
%Burning and ti
rule : { #macro(burning) } 1 { (0,0) > 333 AND ( (0,0) < 573 OR $ti != 1.0 )
         AND (0,0) > #macro(burning) }
rule : { #macro(burning) } { $ti := if($ti = 1.0, time / 100, $ti); } 1
       { (0,0) >= 573 AND #macro(burning) >= (0,0) }
rule : { #macro(burning) } { $ti := time / 100; } 1 { $ti = 1.0 AND (0,0) >= 573 AND
         #macro(burning) < (0,0) }
%Burned
rule : { 209 } 100 { (0,0) != 209 AND (0,0) <= 333 AND (0,0) > #macro(burning) }
%Stay Burned or constant
rule : { (0,0) } 1 { t }
```

FIGURE 11.41 Fire spread model optimization.

```
%Unburned
rule : { temp:= #unburned; } 1 { (0,0)temp=209 AND (0,0)temp573 AND
         (time<=20 OR #unburned>(0,0)temp) }
%Burning and ti
rule : { temp:= #burning; } 1 { (0,0)temp333 AND ( (0,0)temp573 OR (0,0)ti!=1.0 ) AND
         (0,0)temp> #burning }
rule : { #burning } 1 { (0,0)> 333 AND ( (0,0)< 573 OR $ti != 1.0) AND (0,0)>#burning }
rule : { #burning }
       { $ti := if($ti = 1.0, time/100, $ti); } 1 { (0,0)>=573 AND #burning>=(0,0) }
rule : { #burning } { $ti := time / 100; } 1 { $ti=1.0 AND (0,0)>=573 AND #burning<(0,0) }
%Burned
rule : { temp:= 209; } 100
       { (0,0)temp> #macro(burning) AND (0,0)temp<= 333 AND (0,0)temp!= 209 }
%Stay Burned or constant
rule : { } 1 { t }
```

FIGURE 11.42 Fire spread model with I/O ports on each cell.

using multiple ports to replace the extra plane. When we use multiple ports, we do not need to store the values internally but, rather, to transmit them through the ports (Figure 11.42).

In this case, two ports are declared: *temp* and *ti*. Port *temp* exports the cell's temperature, and port *ti* exports the ignition time. When we use multiple ports, we do not need to store the values internally; instead, we transmit them through the ports. Therefore, we do not need to set values but, rather, just to send them out though the corresponding port. Because the initial value for both ports is the same and this model needs different values, it can be solved by assigning negative initial values that will never appear during the simulation and adding two rules that generate the real initial state when the cell has these special values.

11.7.5 Quantizing the Fire Spread Cell-DEVS Model

In order to increase the speed of the simulation, we used quantization to reduce the number of messages exchanged among the cells [20]. First, if we were able to keep the unburned cells completely in the passive state until they reached the ignition temperature, we would reduce the number of cells that sent out messages to their neighbors.

As discussed in Chapter 2, using QDEVS, the cells will send output to their neighbors only if the temperature has exceeded the next quantum threshold. The quantizer acts as the detector that decides when a threshold has been crossed, and it sends out the output only in that case. By implementing quantization as described here, the number of messages exchanged between cells will be reduced, thus increasing the speed of the simulation. However, the accuracy of the simulation will also be reduced. The key is to select a quantum size that gives a good performance increase for a small reduction in accuracy.

In order to create a quantized version of this model, we need to calculate time based on temperature, rather than temperature as a function of time. The first task was to find the inverse of the temperature curve for a typical cell. Given such a function $\mathbf{f(T)}$, we can calculate the amount of time

it will take to reach the next quantum level as a simple difference $\mathbf{f(T_2) - f(T_1)}$. By doing this, cells can be kept quiescent until they reach the next quantum threshold. After this time has passed, they will activate, calculate the next time at which they will cross a threshold, and return to the quiescent state. This saves unnecessary calculations because cells will become active only when a significant change in temperature occurs.

To obtain the required function **f**, we started with a typical temperature curve of a cell in the burning phase. This function fails the horizontal line test and thus is not directly invertible. Therefore, it must be divided into increasing and decreasing components, giving us two invertible functions. A state variable can then be used to choose between them during execution. We found two functions that approximate the two curves. The fit for the increasing temperature portion can be defined as a sum of two exponential functions:

$$f(T) = 11.56 \times e^{0.0005187 \times T} - 784.7 \times e^{-0.01423 \times T} \tag{11.10}$$

where T is the temperature in degrees Kelvin. Similarly, burning down is fit with the linear function:

$$f(T) = 0.052 \times T \tag{11.11}$$

Collecting data for this version of the model would also be more efficient because, instead of sampling every cell of the real model every millisecond, samples would have to be recorded only at threshold crossings (Figure 11.43). These functions were used to develop the time advance portion of the model rules, which are implemented in Cell-DEVS delay functions, as seen in Figure 11.44.

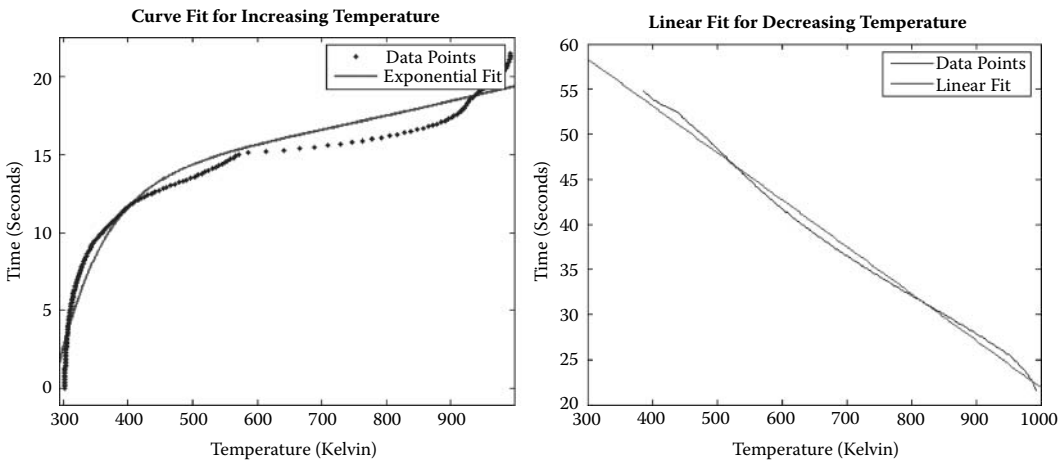


FIGURE 11.43 Inverted temperature function: (a) increasing; (b) decreasing. (Adapted from MacLeod, M. et al. 2006. *Proceedings of ACRI 2006*.)

```
[FireBehavior]
%Unburned
rule : { #macro(unburned) + #macro(q) }
{ round ( ((11.56*exp(0.0005187*((0,0,0)+#macro(q))) - 784.7*exp(0.01423*((0,0,0)+#macro(q)) ) )
        -  (11.56*exp(0.0005187*(0,0,0)) - 784.7 * exp(-0.01423 * (0,0,0) ) ) ) * 100)}
{ cellpos(2)=0 and ( #macro(unburned)>(0,0,0) OR time<=20 ) AND (0,0,0)<573 AND (0,0,0) != 209 }

%Burning
rule : { #macro(burning) + #macro(q) }
{round (((11.56*exp( 0.0005187*((0,0,0)+#macro(q)) ) - 784.7*exp(-0.01423*((0,0,0)+#macro(q)) ) )
       - (11.56*exp( 0.0005187*(0,0,0) ) - 784.7 * exp(-0.01423 * (0,0,0) ) ) ) * 100)}
{ cellpos(2)=0 AND ( ((0,0,0)>#macro(burning) AND (0,0,0)>333) OR
                        (#macro(burning) > (0,0,0) AND (0,0,0) >= 573) )AND (0,0,0) != 209 }
```

FIGURE 11.44 Inverted temperature function rules.

11.8 SUMMARY

In this chapter, we have introduced the use of DEVS and Cell-DEVS in environmental sciences. We started discussing a model on the viability of population spread in a field and different ant foraging models. We then discussed a model on the formation of watersheds, a pollution diffusion model, and a model on vegetation dynamics. We have focused on how these techniques can facilitate the task of the environmental modeler, showing how to deal with these problems using a discrete-event-based approach.

In the case of a quantized continuous model, this requires a fundamental shift in the mechanisms to collect experimental data and to define model equations. For these models, instead of determining what value a dependent variable will have at a given time, we must determine at what time a dependent variable will enter a given state (therefore, the data collection must focus on the time for the state changes). The Cell-DEVS delay function provides a natural mechanism for implementing the quantization function.

Other models in this area can be found in the model repository, including a model of percolation of pesticides in the soil (*./Pesticide_Percolation.zip*) and theoretical examples like the Daisy World model (*./Daisyworld.zip*) and a quantized version of the semiempirical model (*./FireCorseQuantum.zip*).

REFERENCES

1. Darwen, P. J., and D. G. Green. 1996. Viability of populations in a landscape. *Ecological Modeling* 85:165.
2. Wainer, G. 2006. Applying cell-DEVS methodology for modeling the environment. *Simulation: Transactions of the Society for Modeling and Simulation International* 82:635–660.
3. Langton, C. 1986. Studying artificial life with cellular automata. *Physica* 22D:120–149.
4. Ameghino, J., and G. Wainer. 2000. Application of the cell-DEVS paradigm using N-CD++. *Proceedings of the 32nd SCS Summer Computer Simulation Conference,* Vancouver, Canada.
5. Nishidate, K., M. Baba, and R. Gaylord. 1996. Cellular automaton model for random walkers. *Physical Review Letters* 77:1675–1678.
6. Ameghino, J., E. Glinsky, and G. Wainer. 2003. Applying cell-DEVS models of complex systems. *Proceedings of 35th Summer Computer Simulation Conference,* Montreal, QC, Canada.
7. DEVS Representation of Spatially Distributed Systems: Validity, Complexity Reduction. 1996. *Proceedings 6th AI, Simulation and Planning in High Autonomy Systems*, San Diego, CA.
8. Ameghino, J., A. Troccoli, and G. Wainer. 2001. Modeling and simulation of complex physical systems using cell-DEVS. *Proceedings of 34th IEEE/SCS Annual Simulation Symposium*, Seattle, WA.
9. Bianchini, A., F. Indovina, and E. Rinaldi. 1999. Cellular automata for the study of the diffusion of pollutants within the basins of the lagoon: The case of the Venetian lagoon. *Proceedings of 6th International Conference on Computers in Urban Planning and Urban Management,* Venice, Italy.
10. Bandini, S., and G. Pavesi. 2002. Simulation of vegetable population dynamics based on cellular automata. *Proceedings of 5th International Conference on Cellular Automata for Research and Industry. Geneva,* Switzerland, LNCS 2493.
11. Broadbent, S. R., and J. M. Hammersley. 1957. Percolation processes. I. Crystals and mazes. *Proceedings of the Cambridge Philosophical Society* 53:629–641.
12. Rothermel, R. 1972. A mathematical model for predicting fire spread in wildland fuels. Research paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station. 40 pp.
13. Vasconcelos, M. 1988. Simulation of fire behavior with a geographical information system. MSc thesis, University of Arizona.
14. Vasconcelos, M., J. Pereira, and B. Zeigler. 1995. Simulation of fire growth using discrete event hierarchical modular models. EARSeL. *Advances in Remote Sensing* 3:54–62.
15. Ntaimo, L., B. Khargharia, B. Zeigler, and M. Vasconcelos. 2004. Forest fire spread and suppression in DEVS. *Simulation: Transactions of the Society for Modeling and Simulation International* 80:479–500.
16. Muzy, A., E. Innocenti, A. Aiello, J. Santucci, and G. Wainer. 2005. Discrete-event modeling and simulation of fire spreading across a fuel bed. *Simulation: Transactions of the Society for Modeling and Simulation International* 81:103–117.

17. Muzy, A., T. Marcelli, A. Aiello, P. A. Santoni, J. F. Santucci, and J. H. Balbi. 2001. An object-oriented environment applied to a semiphysical model of fire spread across a fuel bed. *Proceedings of European Simulation Symposium 2001—DEVS Workshop,* Marseille, France.
18. Balbi, J. H., P. A. Santoni, and J. L. Dupuy. 1999. Dynamic modeling of fire spread across a fuel bed. *International Journal of Wildland Fire* 9:275–284.
19. López, A., and G. Wainer. 2004. Improved cell-DEVS model definition in CD++. In *ACRI 2004,* LNCS 3305, ed. P. M. A. Sloot, B. Chopard, and A. G. Hoekstra. New York: Springer–Verlag.
20. MacLeod, M., R. Chreyh, and G. Wainer. 2006. Improved cell-DEVS models for fire spreading analysis. *Proceedings of ACRI 2006*, LNCS Vol. 4173, Perpignan, France.

12 Models in Physics and Chemistry

12.1 INTRODUCTION

As discussed in Chapter 1, complex problems in the domain of physics and chemistry are usually modeled with differential equations and solved using numerical approximation methods. In Chapter 5, we introduced some basic models with application in physics and chemistry based on DEVS and Cell-DEVS. This chapter focuses on more advanced models in these fields. We first introduce different examples on reaction–diffusion systems, including snowflake formation and binary solidification. We also present a model of wave propagation, in which wave interference and bouncing are represented using Cell-DEVS. We present a model used for flow-injection analysis, which shows how to model automated analysis of liquid samples in a reactor. We then present various heat transfer models, using adapted versions of the finite-element method. Finally, we present an advanced lattice gas model that represents the movement of particles in a field and three-dimensional representation of particles that can be deformed in a three-dimensional virtual clay environment.

12.2 REACTION–DIFFUSION SYSTEMS

Reaction–diffusion processes are characterized by two or more chemicals that diffuse over a surface and react with one another to produce stable patterns. Reaction–diffusion processes can produce a variety of spot and stripe patterns and are some of the most popular cellular models available. A number of these applications can be found in references 1–3. In this section, we describe different reaction–diffusion models implemented as Cell-DEVS models using CD++ [4].

12.2.1 Diffusion-Limited Aggregation

Diffusion-limited aggregation (DLA) is a phenomenon that occurs when diffusing particles stick to and progressively enlarge an initial seed represented by a fixed object. The seed typically grows in an irregular shape resembling frost on a window. Some examples of DLA can be found in Toffoli and Margolus [3] and Halsey [5]. In these models, diffusion is represented as a random motion with respect to the direction. There are two kinds of particles in the grid: fixed (seeds) and mobile. A mobile particle has the same probability of walking in each direction. When a mobile particle finds a seed, it sticks to the fixed particle and becomes fixed, forming aggregates.

In *./DiffusionLimitedAgregation.zip* we introduce a DLA model implemented as a two-dimensional Cell-DEVS [4]. Initially, a certain percentage of the cells is occupied by mobile particles, and there are one or more seeds. The system evolves with the following rules:

- A particle can move in four directions (north, N; south, S; east, E; west, W).
- A particle becomes fixed if an adjacent cell contains fixed particles.
- An empty cell will be occupied if there is at least one mobile particle trying to move in, and there is no seed adjacent to the mobile particle.

- If there are more than one particle moving toward the same empty cell, the moving direction is used as priority.
- A mobile particle that cannot move will select a new direction at random.
- A mobile particle disappears if it strays too far from the center.

In our implementation, a cell with a value of 0 is empty, values 1–4 represent a mobile particle and its moving direction, and 5 indicates a seed. When a cell is empty, it checks to see if there are any mobile particles wanting to move to that cell. Such a mobile particle can move only if it does not have any adjacent seeds, as follows:

```
rule : { round(uniform(1,4)) } 100 { (0,0)=0 and (
   ( (0,-1)=2 and (-1, -1) !=5 and (1,-1) !=5 and (0,-2) !=5 ) or
   ( (-1,0)=3 and (-1,-1) !=5 and (-2,0) !=5 and (-1,1) !=5 ) or
   ( (0,1) =4 and (1,1) !=5 and (0,2) !=5 and (1,1) !=5 ) or
   ( (1,0) =1 and (1,1) !=5 and (2,0) !=5 and (1,-1) !=5 ) ) }
```

The first condition in this rule checks for empty cells. We then verify the cell to the N, in order to see if the particle wants to move S (while checking that it is not adjacent to a seed). The remaining conditions check the cells to the S, E, and W in a similar way. If any of these conditions hold, the next step is chosen at random.

The following rules illustrate how to resolve conflicts for a particle that is attempting to move a cell up. A mobile particle with moving direction 1 (up) can move to an empty cell above if there is no other mobile particle that attempts to move in:

```
% direction=1 (up): change direction when nowhere to move
rule : {round(uniform(1,4))} 100 { (0,0)=1 and (-1,0)!=0}
rule : {round(uniform(1,4))} 100 { (0,0)=1 and (-1,0)=0 and (
     ( (-2,0) =3 and (-2,-1)!=5 and (-3,0) !=5 and (-2,1)!=5 ) or
     ( (-1,-1)=2 and (-1,-2)!=5 and (-2,-1)!=5 and (0,-1)!=5 ) or
     ( (-1,1) =4 and (-2,1) !=5 and (-1,2) !=5 and (0,1) !=5 ) )}
```

The first rule checks whether the cell is moving N and whether the N cell is occupied. In this case, the direction is chosen at random. The second rule checks whether the N cell is empty. In this case, we first check to see whether the second cell to the N wants to move to the same cell (if the cell is not surrounded by seeds). Similar checks are done on the cells to the S, E, and W.

Whenever a mobile particle is in a cell with any fixed particle adjacent to it, it becomes fixed, as follows:

```
% particle becomes fixed if adjacent cell contains a seed
rule : 5 100 { (0,0)> 0 and (0,0)<5 and ( (-1,0)=5 or (0,-1)=5 or
(0, 1)=5 or (1,0)=5 ) }
```

Several scenarios were executed with different numbers of seeds and percentages of concentration. Figure 12.1 presents a version with a concentration of 30% (grid size: 71 × 71).

Exercise 12.1

Change the initial conditions of the model (including varied concentrations and number of seeds) and study the resulting deposition patterns for each of the initial conditions.

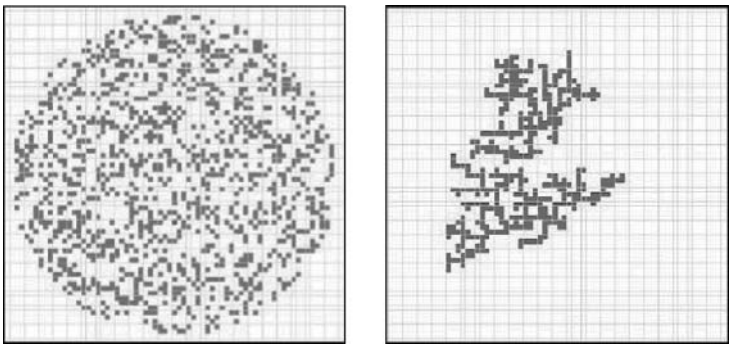


FIGURE 12.1 Initial/final execution results (two seeds and 30% concentration).

12.2.2 A Three-Dimensional Reaction–Diffusion Model

We created a three-dimensional reaction–diffusion model based on the one presented in Weimar [6]. Reaction–diffusion systems can be described by a set of partial differential equations as follows:

$$x_i' = D_i \nabla^2 x_i + f_i(x_1, x_2, \ldots x_n),\ i = 1, \ldots n \tag{12.1}$$

where the first term represents the diffusion equation and the second term is the reaction equation (f_i are always nonlinear; $x_i' = dx_i/dt$). To simulate reaction–diffusion systems, we apply the diffusion rule first and then the reaction rules. In this case, diffusion was implemented as a Cell-DEVS using the following formula:

$$\hat{x} = \frac{1}{cardN} \sum_{x_i \in N} x_i \tag{12.2}$$

that is, an average of the neighbors. Then we apply the reaction rules on this new value. The reaction equation is defined as

$$\frac{x(t) - x(t - \Delta t)}{\Delta t} = f(x) \tag{12.3}$$

We use a three-dimensional von Neumann neighborhood as we need to know the previous state of the cell, we use a second three-dimensional hyperplane as memory of the previous state (i.e., so we use four dimensions). The model, presented in Checiu and Wainer [7] and found in *./ReactionDiffusion.zip,* is defined as shown in Figure 12.2.

The memory rule is in charge of saving the value from the cell below; in the *rd-rule,* we calculate first the diffusion as follows, computed as in Equation (12.3):

```
#BeginMacro(diffusion)
( ((0,0,0,0)+(-1,0,0,0)+(1,0,0,0)+(0,-1,0,0)+(0,1,0,0)+(0,0,-1,0) ) / 7 )
#EndMacro
```

```
type : cell          dim : (5,5,5,2)
delay : transport    border : nowrapped
neighbors :  (0,0,-1,0) (-1,0,0,0)  (0,0,1,0) (0,-1,0,0) (0,0,0,0)  (0,1,0,0)  (1,0,0,0)
  (0,0,0,-1) (0,0,-1,1) (-1,0,0,1) (0,0,1,1)  (0,-1,0,1)  (0,0,0,1)  (0,1,0,1) (1,0,0,1)
localtransition : rd-rule

zone : rd-rule { (0,0,0,0)..(4,4,4,0) }
zone : memory-rule { (0,0,0,1)..(4,4,4,1) }

[memory-rule]
rule : {(0,0,0,-1)} 70 { t }

[rd-rule]
rule : {(#macro(diffusion)-(0,0,0,1))/ 100 } 100 { t }
```

FIGURE 12.2 Three-dimensional reaction–diffusion model.

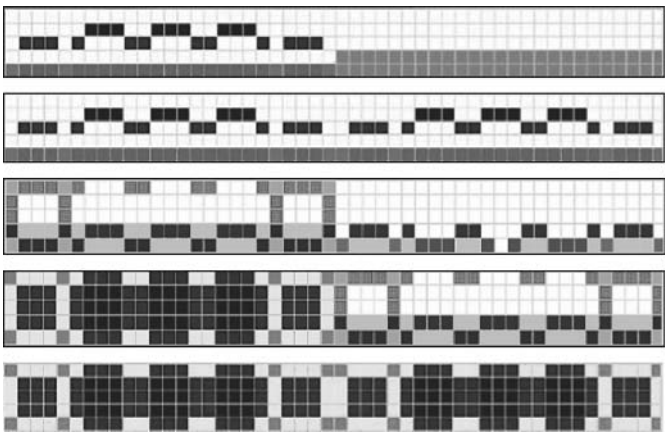


FIGURE 12.3 Three-dimensional reaction–diffusion results.

Then we subtract the previous value of the cell stored in the cell above. The *rd-rule* is applied in the reaction diffusion zone { (0,0,0,0)..(4,4,4,0) }, while the *memory-rule,* which saves the previous state of all cells from the reaction diffusion zone, is saved in { (0,0,0,1)..(4,4,4,1) }.

Figure 12.3 represents the different reaction of two substances. Each row in the figure represents a simulation step and it represents the two cubes as two groups of five squares (each square consisting of 5 × 5 cells). The first group represents the model's state (each of the five planes in the three-dimensional model), and the second group is the memory. For instance, the first five groups in the first row in Figure 12.3 represent the initial state of the model. The next group from the same row represents the initial state of the memory hyperplane.

The second row in Figure 12.3 represents the second step in our simulation: the first five squares represent the actual state of the system and the next five squares store the results of the previous step for computing the next one. The third row is the state of the system in the 20th simulation step. The fourth row is the 34th simulation step, where the system has reached equilibrium (the fifth row shows that the current state is the same as that of the previous state, represented by the memory layer).

12.2.3 Driven Diffusion

Driven diffusion models describe the random motion of two types of particles in a system under the influence of an external field. The field may drive one species of particles to move along the field direction, while the other species moves against that direction. This kind of model can simulate the behavior for certain kinds of materials, such as superionic conductors, fast ion conductors, and solid electrolytes [8]. These two species of particles are differentiated by their positive or negative charge.

The particle space contains approximately the same amount of positive and negative particles so that the total charge of the system is zero. We created a Cell-DEVS model to simulate the system and to study how the density of the particle space affects the behavior of the system [4].

Initially, the space is occupied by a number of particles A and B, which are randomly distributed. Each particle has a randomly chosen direction to face (N/E/S/W). Let us assume that the field points to the NE. If an external electrical field appears, the preferable moving direction of particle A is N or E, and the preferable moving direction of particle B is S or W. The probability of A and B moving along that preferable direction is a, while the probability of moving against the direction is $(1 - a)$.

The rules for a particle of either type to move are as follows:

1. The cells move only toward the direction that they are facing (the neighboring cell in that direction is called the *adjacent* cell).
2. If the adjacent cell is occupied, then the particle remains in its current place and chooses a new direction at random.
3. If the adjacent cell is empty but faced by one or more particles, the particle remains in its current place and randomly chooses a new direction.
4. If the adjacent cell is empty and faced by no other particles, then the particle moves to the adjacent cell and chooses a new direction randomly.

The rules for updating cells are as follows:

1. If the cell is empty and faced by no particles, then it remains empty.
2. If the cell is empty and faced by exactly one particle, then the cell will be occupied.
3. If the cell is empty and faced by two or more particles, then it remains empty.
4. If the cell is occupied and the inside particle faces an empty cell that is not faced by other particles, then it will be vacated.

These rules, which can be found in *./DrivenDiffusion.zip,* are presented in Figure 12.4.

The rules used by particles A or B to choose a direction to face at random are as follows:

A. The probabilities of choosing N, E, S, or W to face are $(a/2)$, $(a/2)$, $(1 - a)/2$, and $(1 - a)/2$, respectively (defined in macro *RandA*); $a \in [0,1]$.
B. The probabilities of choosing N, E, S, or W to face are $(1 - a)/2$, $(1 - a)/2$, $(a/2)$, and $(a/2)$, respectively (defined in macro *RandB*); $a \in [0,1]$.

We use the first digit in the cell state to identify the kind of particle (1 = A, 2 = B) and the second digit to identify its direction. In the first rule in the model (rule 2), the preconditions test that particle A moves to N or E and that particle B moves to S or W. In this case, we move the particle at random (using the random probability function for each of the two particles, as defined by the *RandA* and *RandB* macros). We then define preceding rule 3: if the adjacent cell is empty, but faced by one or more particles, we choose a direction at random while keeping that cell (the postcondition of the rule activates macros *RandA* or *RandB* according to the first digit in the cell's value, accordingly). To check this precondition, we first check whether the current cell is heading N (21 or 11) and the N cell is empty. We then verify that the cells around the N cell are not pointing to it (the rest of the rule repeats the test in every other direction). Then we implement rule 4: we check whether the cell is empty; in such a case, we verify whether any of the neighboring cells are facing it. We want to receive a particle from the N, and thus we check to see whether the cells to the E, S, or W are facing it; there are symmetric rules for receiving a particle from the S, E, or W. The following rule is in charge of implementing the cell updating process. In this case, if the origin cell is moving toward N, the N cell is empty, and the cells surrounding the N cell are not going to occupy it, we empty the current cell.

```
[DrivenDiffusion]
type : cell
dim : (16,16)
delay : transport
border : wrapped
neighbors : (-1,-1)(-1,0)(-1,1)(0,-1)(0,0)(0,1)(1,-1)(1,0)(1,1)(2,0)(-2,0)(0,2)(0,-2)
            (0,2) (0,-2)
localtransition : diffusion-rule

[diffusion-rule]
%Rule 2
rule : {#macro(RandA)} 100 {( (0,0)=11 and (0,1)>0 or (0,0)=12 and (1,0)>0 or (0,0)=13 and

                              (0,-1)>0 or (0,0)=14 and (-1,0)>0 )}
rule : {#macro(RandB)} 100 {( (0,0)=21 and (0,1)>0 or (0,0)=22 and (1,0)>0 or (0,0)=23
                               and (0,-1)>0 or (0,0)=24 and (-1,0)>0 )}

%Rule 3
rule : {if ((0,0)>20, #macro(RandB),#macro(RandA)) } 100  {
     ((0,0)=21 or (0,0)=11) and (0,1)=0 and % Heading N, and N cell empty
     ((1,1)=14 or (1,1)=24  or (0,2)=13 or (0,2)=23 or (-1,1)=12 or (-1,1)=22)
                                                          %No one pointing N
  or ((0,0)=22 or (0,0)=12) and (1,0)=0 and % Idem, heading S
     ((2,0)=14 or (2,0)=24  or (1,1)=13 or (1,1)=23 or (1,-1)=11 or (1,-1)=21)
  or ((0,0)=23 or (0,0)=13) and (0,-1)=0 and % Idem, heading E
     ((0,-2)=11 or (0,-2)=21 or (1,-1)=14 or (1,-1)=24 or (-1,-1)=12 or (-1,-1)=22)
  or ((0,0)=24 or (0,0)=14) and (-1,0)=0 and % Idem, heading W
     ((-1,-1)=11 or (-1,-1)=21  or (-1,1)=13 or (-1,1)=23 or (-2,0)=12 or (-2,0)=22)
}

%Rule 4
rule : {if ((1,0)>20, #macro(RandB),#macro(RandA))} 100 { (0,0)=0 and ((1,0)=14 or
(1,0)=24) and not ( (0,1)=13 or (0,1)=23 or (0,-1)=11 or (0,-1)=21
       or (-1,0)=12 or (-1,0)=22)}

...

%If the cell occupied by a particle that can move, then this cell is vacated
rule : 0 100 { ((0,0)=14 or (0,0)=24) and (-1,0)=0 and
             not ((-1,-1)=11 or (-1,-1)=21 or (-2,0)=12 or (-2,0)=22 or (-1,1)=13 or
                  (-1,1)=23)}
...
```

FIGURE 12.4 Driven diffusion definition in CD++.

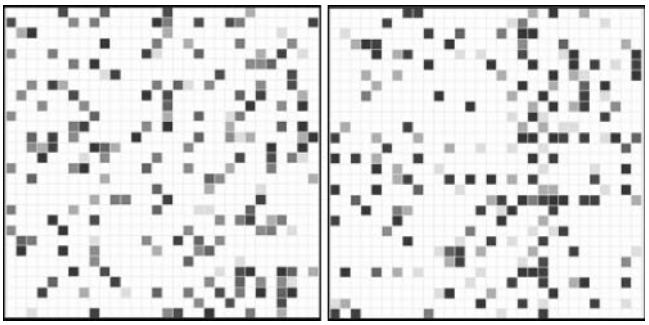
We show various test cases considering different density values, space size, and initial states, which have particles initially distributed at random in the space according to the given density value. In Figure 12.5, about 20% of the cell space is occupied by the randomly distributed particles.

After 100 time steps, the particles are still randomly distributed. The cell space remains disordered over the simulated time steps, while the distributions of particles A and B are homogeneous (high current in the system). Similar results were obtained with a density of 10%. Then we tested a case in which density of the whole space is higher (50%). The simulation results are illustrated in Figure 12.6.

From the results at time step 100 as shown in Figure 12.6, we can see that the distribution of the two particles exhibits a striped structure. Within each strip, there are two substrips with approximately the same number of particles. This indicates the nonhomogeneities of the distribution of two particles and thus results in reduced current in the system. Similar results were obtained when we used densities between 40 and 70%.

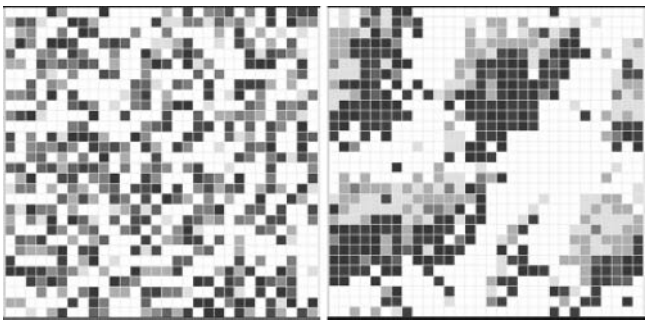
Exercise 12.2

Investigate different scenarios using varied initial conditions and densities.



(a) (b)

FIGURE 12.5 Low density of 20%: (a) initial state; (b) after 100 time steps.



(a) (b)

FIGURE 12.6 Density of 50%: (a) initial cell space; (b) 100 time steps.

12.2.4 Snowflake Formation

We built a model of snowflake growth based on the local cellular model for snow crystal growth described by Reiter [9]. The dendrite growth of snowflake is very complex and influenced by many factors in its natural environment. The shape of snow crystals depends upon the saturation and temperature during growth. The forms observed include dendrites, needles, spatial columns, scrolls, etc. Wolfram [1] introduced a simplified Boolean model for snowflake growth in which, at each time step, each cell either contained a particle of ice or not. In the subsequent step, cells that contain ice remain solid; cells that without ice become solid if exactly one of the neighboring cells is.

We defined such a model using a two-dimensional Cell-DEVS [4] which can be found in *./Snow.zip*. The model presented is more realistic, but the transition rules are similar to the ones presented in Wolfram [1]. Each cell contains a real value, representing the relative humidity in the cell. Values of one or higher represent ice; lower values represent humidity (which can spread to neighboring cells). Each cell is classified as either receptive or nonreceptive. The first stage is to determine the receptive sites: those that are ice or have an immediate ice neighbor. In the next stage, the values of the receptive cells are computed as a constant γ plus a diffusion term. The diffusion term is a local average of a modified cellular field obtained by setting the receptive sites to zero and computed as

$$Vu = 0.5 \times Vo + 0.5 \times \sum Vn/8 \tag{12.4}$$

where *Vu* is the update value of the cell, *Vo* is the original value of the cell, and *Vn* is the value of the neighbors. Therefore, the center cell has a weight of 50%, and each of the eight remaining cells has a weight of 1/16. Receptive cells are seen as permanent (storing any mass arriving at them); nonreceptive sites are free to move. The constant γ added to receptive sites informally captures the idea that some humidity may be available from outside the plane of growth. The second parameter used is the background level β. We begin with a single cell of value 1 (an ice seed) in a constant field.

The model uses three types of cells and three rules addressing each cell group independently:

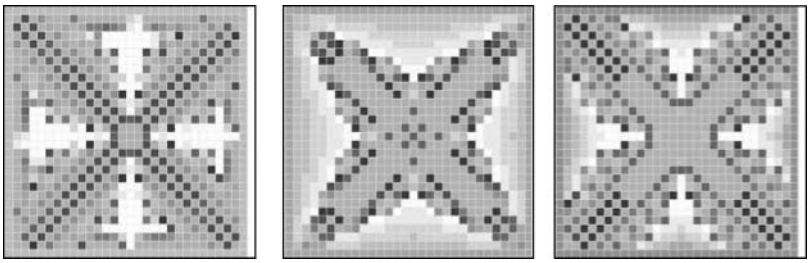
- Cells with ice: every ice cell will absorb more and more water in the air continuously.
- Receptive cells are not solid and have no ice neighbors.
- Nonreceptive cells are not solid, but they have ice neighbors.

In Figure 12.7, we present the rules used for receptive cells (i.e., those computed based on Equation 12.4.) The original value of the center cell has 50% of the weight of the final updated value of the cell ((0,0) × 0.5). The remaining 50% is contributed by eight groups of nested *ifs,* each of them representing the value of one adjacent neighbor. In each nested *if,* the rule also estimates the values of the neighbor's neighbor to determine the contribution value. For instance, if the values of the immediate neighbors of cell (–1,1) are all zero, this means they are all non-ice cells. In this case, we will leave the contribution value of (–1,1) to the center cell unchanged; otherwise, this value will be zero. After the predefined elapsed time, the value of the center cell will be updated by the new value that we computed using this rule.

In order to investigate different results from the model by using different variables, we set three different group variable vectors (γ,β) and size of the cell space to 30 × 30. This model was tested using different models for the three vectors, as shown in Figure 12.8.

```
rule : {(0,0)/2 +
      (
   if(  ( if((-2,2)>=1, 1,0) + if((-1,2)>=1, 1,0) + if((0,2)>=1, 1,0) + if((-2,1)>=1, 1,0) +
          if((0,1) >=1, 1,0) + if((-2,0)>=1, 1,0) + if((-1,0)>=1,1,0))=0, (-1,1), 0)
+
   if(  ( if((-1,2)  >=1,1,0) +  if((0,2) >=1,1,0) +  if((1,2) >=1,1,0)  +
          if((-1,1)  >=1,1,0) +  if((1,1) >=1,1,0) +  if((-1,0)  >=1,1,0) +  if((1,0) >=1,1,0))=0,
             (0,1),0)
+
   if(  ( if((0,2) >=1,1,0) + if((1,2) >=1,1,0) + if((2,2) >=1,1,0) + if((0,1) >=1,1,0) +
          if((2,1) >=1,1,0) + if((1,0) >=1,1,0) + if((2,0) >=1,1,0) ) = 0,(1,1),0)
+
   if(  ( if((-2,1) >=1,1,0) + if((-1,1)>=1,1,0) + if((0,1) >=1,1,0)+ if((-2,0) >=1,1,0) +
          if((-2,-1) >=1,1,0)+ if((-1,-1) >=1,1,0)+ if((0,-1) >=1,1,0)) = 0,(-1,0),0)
+
   if(  ( if((0,1) >=1,1,0) + if((1,1) >=1,1,0) + if((2,1) >=1,1,0) + if((2,0) >=1,1,0) +
          if((0,-1) >=1,1,0) + if((1,-1) >=1,1,0) + if((2,-1) >=1,1,0) ) = 0,(1,0),0)
+
   if(  ( if((-2,0) >=1,1,0)+ if((-1,0) >=1,1,0) + if((-2,-1) >=1,1,0)+ if((0,-1) >=1,1,0) +
          if((-2,-2) >=1,1,0)+ if((-1,-2) >=1,1,0)+ if((0,-2) >=1,1,0)) = 0,(-1,-1),0)
+
   if(   ( if((-1,0) >=1,1,0)+ if((1,0) >=1,1,0)+ if((-1,-1) >=1,1,0)+ if((1,-1) >=1,1,0) +
          if((-1,-2) >=1,1,0)+ if((0,-2) >=1,1,0)+ if((1,-2) >=1,1,0)) = 0,(0,-1),0)
+
   if(  ( if((1,0) >=1,1,0) + if((2,0) >=1,1,0)+ if((0,-1) >=1,1,0)+ if((2,-1) >=1,1,0) +
          if((0,-2) >=1,1,0)+ if((1,-2) >=1,1,0)+ if((2,-2) >=1,1,0)) = 0,(1,-1),0)
) /16 }

10

{ (0,0)<1 and (if((-1,1)>=1 ,1,0)+ if((0,1)>=1,  1,0)+ if((1,1)>=1 ,1,0)+ if((-1,0)>=1,1,0)+
               if((1,0) >=1, 1,0)+ if((-1,-1)>=1,1,0)+ if((0,-1)>=1,1,0)+ if((1,-1)>=1,1,0) ) =0}
```

FIGURE 12.7 CD++ model specification for snowflake growth.



(a) (b) (c)

FIGURE 12.8 Snowflake formation after 10,000 iterations: (a) $\beta = 0.3$, $\gamma = 0.001$; (b) $\beta = 0.4$, $\gamma = 0.01$; (c) $\beta = 0.05$, $\gamma = 0.0035$.

Exercise 12.3

Investigate different simulation results using varied initial conditions and values of γ and β.

Exercise 12.4

Implement the model in Wolfram [1] as a Cell-DEVS model. Investigate how variable delays can improve the definition of the original cellular model.

12.2.5 Binary Solidification

In this section, we present a model defining a binary solidification process based on the work presented in Kremeyer [10]. The model represents the solidification of particles of ammonium chloride (NH_4Cl) in water solution (H_2O); it can be used to analyze the crystal structure, the speed of generation of the crystals, and the patterns generated. The model considers the state (solid or liquid) and the quantity of solute (i.e., the volume of H_2O) on each particle. The values in each cell oscillate between zero and one, depending on the quantity of solute in each of them. Solid cells have a solute concentration of zero. The model executes the following steps:

1. Identify all the liquid particles.
2. Analyze each particle and, based on the solute concentration of each of them, decide whether the particle solidifies.
3. Every solidifying particle must expel all the available solute to the liquid neighbors because solute cannot deposit in solid particles. If there are no liquid neighbors, the particle cannot solidify because the system equilibrium must be kept (maintaining the number of particles of H_2O).
4. Once the whole surface has been analyzed, a solute diffusion model must be applied to the liquid particles in order to keep the solute equilibrium in the system.

In order to define the first step, we have to pick the cells whose values are different from one (solid). In the second step, we need to know when a liquid cell becomes solid. To do so, we need to determine different physical aspects in the cell (surface tension, interface curvature, and crystalline anisotropy), which play a role in the process. We first compute the concentration equilibrium, using the first and second near neighbors (i.e., those neighbors within a radius of two cells). We consider only the solid neighbors, defined as

$$C_{eq} = C_l - \gamma \left(flat - \sum_{\substack{k \in neighbors(1) \\ k \in neighbors(2)}} (\phi_k \varepsilon_k) \right) \tag{12.5}$$

where

C_{eq} = the concentration equilibrium;
C_l = the concentration equilibrium on a flat surface;
γ = the surface tension;
ϕ_k = the state of neighbor k (solid or liquid); and
ε_k = the distance to the neighbor k.

Here, *flat* is half of the addition of all the distances between the neighbors (in this case, 20, because we have 8 neighbors with a distance of 1 and 16 with a distance of 2). Once C_{eq} is computed, if the cell being analyzed is under solute equilibrium, it will solidify, expelling the remaining solute to the neighbors.

The following step is in charge of diffusing the solute expelled by the cells. Whenever a number of neighbors in liquid state can absorb the expelled molecules, they do; otherwise, the cell cannot freeze. The following equation defines such behavior:

$$C_{new} = C_{old}(1 - 8\lambda) + \lambda \sum_{k \in neighbors(1)} \phi_k \tag{12.6}$$

where

C_{new} represents the quantity of solute that the cell will have after receiving the quantity expelled by the solidified neighbors;
C_{old} is the current quantity of solute;
λ is the diffusion constant (which depends on the distance between the cells and the delay of the diffusion algorithm); and
ϕ_k is the status of the neighbor k.

This process is executed a few times to improve the uniformity of diffusion.

Figure 12.9 shows the specification of this model in CD++, which can be found in *./BinSol.zip*. This is a 30 × 30 model using two layers. In the first layer, we have the state value (solid or liquid) for every particle, and the second layer is used to know what the current phase in the process is. We use the integer part of the cell's value to store the current state (solid or liquid) and the floating point part to store the percentage of solute contained in the cell.

The first rule is evaluated if the cell is in plane 1. This rule computes the concentration equilibrium (C_{eq}), using the following parameters: $C_l = 0.8$ (i.e., 80% of solute) and $\gamma = 0.01$. Once the cell is in equilibrium, it is "masked" by putting a value of two in the integer part of the cell. The following rule is used to analyze whether the cells marked as potential cells for solidification have any neighbors in liquid state. If this is not the case, the cell cannot freeze because it cannot expel the particles in excess. The next step is used to determine how many neighbors are liquid (in order to expel solute in isotropic fashion, i.e., each neighbor receives an equal and proportional number of particles).

The following rule is used to verify that the quantity of solute received in each cell is below 100% of the volume. If this is not the case, the cells that were ready to solidify should abort the execution of this rule because they have no room to deposit the solute they need to expel (we mark these cells using the value 99). In the following rule, the cells detected in the previous step (i.e., those that were able to freeze and have a marked neighbor) are converted into liquid again. The following rule

```
[field]
dim : (30,30,2)
delay : transport
border : nowrapped
neighbors : (-2,-2,0) (-2,-1,0) (-2,0,0) (-2,1,0) (-2,2,0) (-1,-2,0) (-1,-1,0) (-1,0,0)
(-1,1,0) (-1,2,0)(0,-2,0) (0,-1,0) (0,0,0) (0,1,0) (0,2,0) (1,-2,0) (1,-1,0) (1,0,0)
(1,1,0) (1,2,0)    (2,-2,0) (2,-1,0) (2,0,0) (2,1,0) (2,2,0) (0,0,1)
localtransition : calculus

[calculus]
rule : { if( fractional(0,0,0) < (0.8-(0.01*(20 - 2*trunc(-2,-2,0) + 2*trunc(-2,-1,0) +
     2*trunc(-2,0,0) + 2*trunc(-2,1,0) + 2*trunc(-2,2,0) + 2*trunc(-1,-2,0) + 2*trunc(-1,2,0) +
     2*trunc(0,-2,0) + 2*trunc(0,2,0) + 2* trunc(1,-2,0) + 2*trunc(1,2,0) + 2*trunc(2,-2,0) +
     2*trunc(2,-1,0) + 2*trunc(2,0,0) + 2*trunc(2,1,0 ) + 2*trunc(2,2,0) + 1*trunc(-1,-1,0) +
     1*trunc(-1,0,0) + 1*trunc(-1,1,0) + 1*trunc(0,-1,0) + 1*trunc(0,1,0) + 1*trunc(1,-1,0) +
     1*trunc(1,0,0) + 1*trunc(1,1,0) ), 2+ fractional(0,0,0) , (0,0,0)}
               100 { Cellpos(2) = 0 and (0,0,1) = 1 and trunc(0,0,0)) = 0}


rule : { if
    ( if(trunc(-1,-1,0)=0,0,1) + if(trunc(-1,0,0)=0,0,1) + if(trunc(-1,1,0)=0,0,1) +
      if(trunc(0,-1,0)=0,0,1) + if(trunc(0,1,0)=0,0,1) +  if(trunc(1,-1,0)=0,0,1) +
      if(trunc(1,0,0)= 0,0,1) + if(trunc(1,1,0)=0,0,1)
     ) = 8, fractional(0,0,0) ,(0,0,0))}
               100 { cellpos(2) = 0 and (0,0,1) = 2 and trunc((0,0,0)) = 2}

rule : {  (if(trunc(-1,-1,0)=0,0,1) + (if(trunc(-1,0,0)=0,0,1) +
          (if(trunc(-1,1,0)=0,0,1) + (if(trunc(0,-1,0)=0,0,1) +
          (if(trunc(0,1,0)=0,0,1) + (if(trunc(1,-1,0)=0,0,1) +
          (if(trunc(1,0,0)=0,0,1) + (if(trunc(1,1,0)=0,0,1) + (0,0,0) }
               100 { cellpos(2)=0 and (0,0,1)=3 and trunc(0,0,0)=2}

rule : { if(
if(trunc(-1,-1,0)<2, 0, fractional(-1,-1,0)/ (trunc(-1,-1,0)-2) ) +
if(trunc(-1,0,0) <2, 0, fractional(-1,0,0) / (trunc(-1,0,0) -2) ) +
if(trunc(-1,1,0) <2, 0, fractional(-1,1,0) / (trunc(-1,1,0) -2) ) +
if(trunc(0,-1,0) <2, 0, fractional(0,-1,0) / (trunc(0,-1,0) -2) ) +
if(trunc(0,1,0)  <2, 0, fractional(0,1,0)  / (trunc(0,1,0)  -2) ) +
if(trunc(1,-1,0) <2, 0, fractional(1,-1,0) / (trunc(1,-1,0) -2) ) +
if(trunc(1,0,0)  <2, 0, fractional(1,0,0)  / (trunc(1,0,0)  -2) ) +
if(trunc(1,1,0)  <2, 0, fractional(1,1,0)  / (trunc(1,1,0)  -2) ) +
                        fractional(0,0,0)) > 1, 99+fractional(0,0,0),
   (
if(trunc(-1,-1,0) < 2,0, fractional(-1,-1,0)/(trunc(-1,-1,0) - 2))) +
if(trunc(-1,0,0)  < 2,0, fractional(-1,0,0) /(trunc(-1,0,0) - 2))) +
if(trunc(-1,1,0)  < 2,0, fractional(-1,1,0) /(trunc(-1,1,0) - 2))) +
if(trunc(0,-1,0)  < 2,0, fractional(0,-1,0) /(trunc(0,-1,0) - 2))) +
if(trunc(0,1,0)   < 2,0, fractional(0,1,0)  /(trunc(0,1,0) - 2))) +
if(trunc(1,-1,0)  < 2,0, fractional(1,-1,0) /(trunc(1,-1,0) - 2))) +
if(trunc(1,0,0)   < 2,0, fractional(1,0,0)  /(trunc(1,0,0) - 2))) +
if(trunc(1,1,0)   < 2,0, fractional(1,1,0)  /(trunc(1,1,0) - 2))) + (0,0,0)
  )
  ) }
                100 { cellpos(2)=0 and (0,0,1)=4 and trunc((0,0,0)) = 0 }

rule : { if(
      ( if(trunc(-1,-1,0)=99,1,0) + if(trunc(-1,0,0)=99,1,0) + if(trunc(-1,1,0)=99,1,0) +
        if(trunc(0,-1,0)=99,1,0) + if(trunc(0,1,0)=99,1,0) + if(trunc(1,-1,0)=99,1,0) +
        if(trunc(1,0,0)=99,1,0) + if(trunc(1,1,0)=99,1,0)
       ) > 1, fractional(0,0,0),(0,0,0))}
               100 { cellpos(2) = 0 and (0,0,1) = 5 and trunc(0,0,0) = 2}

rule : 1 100 { cellpos(2) = 0 and (0,0,1)=6 and trunc((0,0,0))>1 and trunc((0,0,0)) != 99 }
  }
rule : { fractional((0,0,0)) } 100 {cellpos(2) = 0 and (0,0,1) = 7 and trunc((0,0,0)) = 99 }


rule : { (1 - (0.00001 *
```

FIGURE 12.9 Definition of the binary solidification model.

```
            (
              if(trunc(-1,-1,0)=1,0,1) + if(trunc(-1,1,0) = 1,0,1) + if(trunc(1,1,0) =
                        1,0,1) + if(trunc(1,-1,0) = 1,0,1)
             )                    )
          -
           (0.00001 *
              (
              if(trunc(-1,0,0) = 1,0,1) + if(trunc(0,1,0) = 1,0,1) + if(trunc(1,0,0) =
                        1,0,1) + if(trunc(0,-1,0) = 1,0,1)
              )            )
              ) * fractional(0,0,0)
       +
           (0.00001 *               (
               if(trunc(-1,-1,0)= 1,0,fractional(-1,-1,0)) +
               if(trunc(-1,1,0) = 1,0,fractional(-1,1,0) ) +
               if(trunc(1,1,0)  = 1,0,fractional(1,1,0)  ) +
               if(trunc(1,-1,0) = 1,0,fractional(1,-1,0) )             )            )
        +
            (0.00001 *                 (
               if(trunc(-1,0,0) = 1,0,fractional(-1,0,0)) +
               if(trunc(0,1,0) =  1,0,fractional(0,1,0)) +
               if(trunc(1,0,0) =  1,0,fractional(1,0,0)) +
               if(trunc(0,-1,0) = 1,0,fractional(0,-1,0) )                    )
}         100 { cellpos(2) = 0 and (0,0,1)>= 8 and trunc(0,0,0)=0 }

rule : { (0,0,0) + 1 } 100 { (0,0,0) <= 15 and (0,0,0) >= 1 and Cellpos(2) = 1 }
rule : 1 100 { Cellpos(2) = 1 }
rule : { (0,0,0) } 100 { t }
```

FIGURE 12.9 (continued).

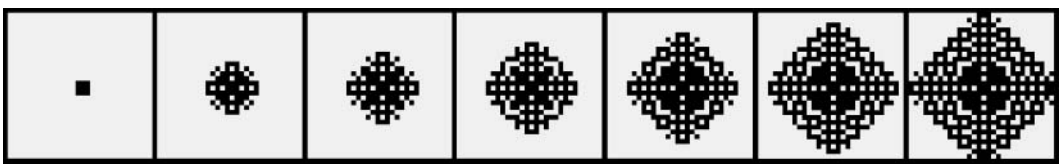


FIGURE 12.10 Structure of the growing crystals.

solidifies all the cells meeting the previous conditions. The next rule deletes all existing marks, and then we generate the diffusion process (which is repeated 15 times). The last rule is used to update the current step number as a reference to know which following rule to analyze.

Figure 12.10 shows the simulation results. We initially freeze the four central cells, and the rules expand to form the crystal.

12.3 A MODEL OF WAVE PROPAGATION

In this section we will present a model of waves propagating in water, previously defined in Ameghino and Wainer [11] and found in *./Wave.zip.* Nutaro [12] presents a DEVS implementation of two different simulation schemes, called *finite difference time domain* and *digital wave network*. He introduces the application of discrete-event techniques for propagation in one- and two-dimensional fields. In our case, we will present a model of this phenomenon using Cell-DEVS. Our model addresses the analysis of two kinds of phenomena: wave interference and medium changes.

The state of a wave on a liquid medium can be represented as sinusoidal functions, and it is characterized by the phase, intensity, direction, and frequency of such functions. When we combine two or more waves with the same amplitude and frequency propagating on the same media, there is an

interference pattern caused by their superposition. This interference can be either constructive or destructive. Constructive interference is produced when two waves of equal amplitude, frequency, and phase interfere with each other. The resulting wave is equivalent to the addition of both waves. Destructive interference is produced by two waves of equal amplitude and frequency crossing, but with different phase. The resulting wave is the sum of both; however, after superposition, the wave will be canceled in those zones where there is overlap. The amount of interference depends of the phase difference at the point of interference.

When a wave encounters a change in the medium, some of or all the changes can propagate into the new medium (or the wave can be reflected from it). The part that enters the new medium is called the *transmitted* portion, and the rest is called the *reflected* portion. The reflected portion depends on the characteristic of the incident medium; if this has a lower index of refraction, the reflected wave has a 180° phase shift upon reflection. Conversely, if the incident medium has a larger index of refraction, the reflected wave has no phase shift.

In order to simulate the interference between waves and the propagation in CD++, we defined a three-dimensional model with five overlapping planes, each plane defining the wave movement in each direction (only four directions for this example). The main plane, which is a composition of the direction planes, contains the value or intensity of the wave corresponding to this position. Every cell in the cell space represents the minimal possible portion of the medium in which the wave propagates. Each cell has two values: phase and intensity. An integer value between zero and eight represents phase and a fractional value between zero and one represents its intensity.

Figure 12.11 shows the rules used in CD++ to specify the wave propagation behavior. The first rule governs the attenuation of the wave. If the wave intensity is below 0.0001, the wave propagation stops. The second rule contemplates the spread of the wave toward its neighbors, which preserves the phase of the wave but attenuates its intensity (because of propagation). The third rule contemplates the spread of the wave in the current cell. In this case, the cell intensity value does not change (only the phase).

Figure 12.12 shows the integration rule, which describes the combination of the values in the direction planes used in order to obtain the wave value in the medium in which it is traveling (the value corresponds to the discretization in eight phases of a sine wave). As we can see, if any of the four direction planes is not zero, the rule takes the data stored on each of the planes and computes a sine function that depends on the current phase (*trunc(0,0,i)*) and its amplitude (*functional(0,0,i)*).

Figure 12.13 shows the simulation results of the wave model. Only the integration plane is shown (direction and intensity). It is possible to appreciate that the wave propagation produces attenuation (the intensity of the wave is reduced while the signal travels).

Exercise 12.5

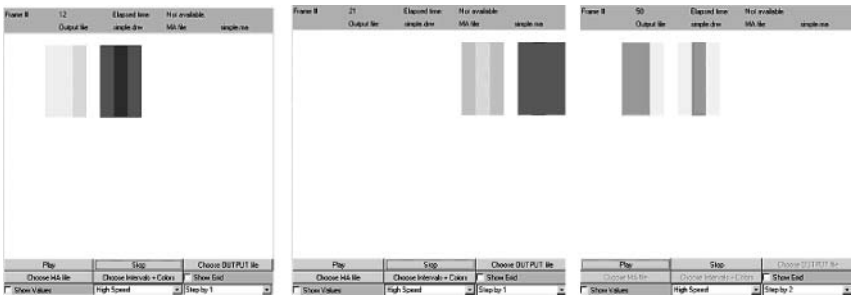
Modify the wave model and create waves in different directions. Study constructive interference, destructive interference, and propagation in different media.

```
rule: {0} 100 {trunc(0,0,0)=#maxFase or (fractional(0,0,0)<.0001 and fractional((0,0,0))>0)}
rule: {trunc(1,0,0)+fractional(1,0,0)*#attenuation} 100 {(1,0,0)!=0}
rule: {trunc(0,0,0)+1+fractional(0,0,0)} 100 { (0,0,0)!=0}
```

FIGURE 12.11 Rules for wave propagation.

```
rule: {((sin(PI/4*trunc((0,0,1))) * fractional((0,0,1)) + sin(PI/4 * trunc((0,0,2))) *
    fractional((0,0,2)) + sin(PI/4 * trunc((0,0,3))) * fractional((0,0,3)) +
    sin(PI/4*trunc((0,0,4))) * fractional((0,0,4)) )) * 10 } 100 {(0,0,1)!=0 or (0,0,2)!=0 or
    (0,0,3)!=0 or (0,0,4)!=0}
```

FIGURE 12.12 Integration rule.



(a) (b) (c)

FIGURE 12.13 Result of wave propagation: (a) a wave traveling from left to right; (b) the wave reflects with the right border; (c) the wave before reflecting with the left border.

12.4 FLOW INJECTION ANALYSIS (FIA)

Flow injection analysis (FIA) is used for the automated study of liquid samples. In a flow injection analyzer, a small, fixed volume of a liquid sample is injected into a liquid carrier, which flows through a narrow tube. As a result of convection at the beginning, and later of axial and radial diffusion, this sample is progressively mixed into the carrier as it is transported along the tube. The addition of reagents at different confluence points (which mix with the sample due to radial dispersion) produces reactive or detectable species, which can be sensed by flow-through sensors. Figure 12.14 presents a simple FIA apparatus. This device (called an FIA manifold) consists of a pump (P) that adds carrier solution (nitric acid—HNO_3) into a valve that connects to a tube-shaped reactor (L). At the end of the tube, a sensor (B) detects specific properties of the flowing solution. The valve can be turned to allow the flow of the sample (water) into the reactor. The sample is held in a loop (l) and when the valve is rotated, its content flows into the reactor. As a result of the chemical activity between the sample and the carrier solution, a change will be observed in the sensor (B), making it possible to compare the results with those obtained by known samples [13].

In FIA systems, convective transport yields a parabolic velocity profile with molecules at the tube walls having speed zero and those at the center having twice the average velocity. At the same time, the presence of concentration gradients develops axial and radial diffusion of sample molecules. In FIA systems of practical interest, axial molecular diffusion has almost no influence in the overall dispersion, and radial diffusion is the main contributor [13]. For a pump providing a net flow of q mL/min in a coil of radius a, the average flow velocity is given by

$$V_a = \frac{q}{60 \cdot (\pi \cdot a^2)} \tag{12.7}$$

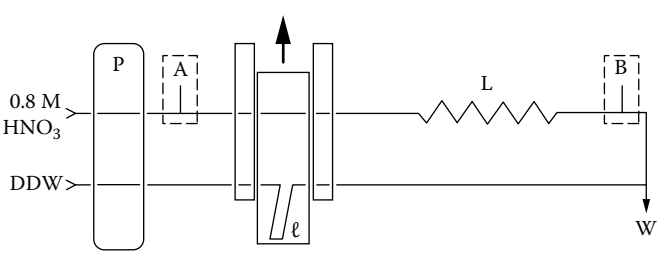


FIGURE 12.14 FIA manifold: P = pump; l = loop; L = reactor; W = waste; A, B = detection points. (From Andrade, F. J. et al. 1998. *Analytica Chimica Acta* 19211.)

At a point at distance r from the center, the flow velocity is described by

$$v(r) = 2 \cdot V_a \cdot \left(1 - \frac{r^2}{a^2}\right) \tag{12.8}$$

As convective transport and the diffusion gradient force the water sample to be released from the walls inside the reactor L, a reduction of the blocking area is produced. This allows electric current to flow, enabling measuring conductivity values different from zero.

Troccoli and colleagues [14] presented a Cell-DEVS model describing the integrated conductivity flow-injection system (ICM) in detail. For this system, a cell space of 25 rows and 200 columns was defined, each cell representing 0.001 × 0.1 cm of a half-tube section. Row 0 represents the center of the tube and row 24 represents the section of the tube touching its walls; the value of each cell will represent the nitric acid concentration. To deal with convective transport and radial diffusion at the same time, the model reacts in two phases: transport and diffusion. The local computing function simulates the transport phase, and all cells are connected to an external generator sending an event, which triggers the diffusion phase. The model is built as a coupled DEVS model with two components: a Cell-DEVS (named *fia*) representing the tube and a DEVS atomic model (a DEVS *generator* model). The model, found at *./fia.zip,* is shown in Figure 12.15. As we can see, we use inertial delays (in order to permit transport rules to be preempted by diffusion, if needed). The basic behavior of each cell is defined by the transport rules.

The convective transport has been arbitrarily defined in the direction of increasing column values (in visual representations, the carrier will be seen flowing from left to right). The local transition rule for the transport phase should set a cell's value to the current value of its (0,–1) neighbor cell at a rate depending on the velocity of the flow at the cell (maximum at the center of the tube and decreasing toward the walls.

The delay is calculated using Equations (12.7) and (12.8). For a pump with a constant flow of 1.33 mL/min, the average speed is 11.29 cm/s (substituting this value in Equation 12.8, we obtain 22.57878). We also need to know the distance to the center of the tube. Consequently, `cellPos(0)*0.001+0.0005` is the distance of the center of the cell to the center of the tube and, therefore, `22.57878*( 1-power(cellPos(0)*0.001+0.0005,2)/0.000625 )` is the solution to Equation (12.8) (a = 0.025 cm). Having the velocity of flow $v(r)$, the delay will be the time

```
components : fia generator@ConstGenerator
link : out@generator diffuse@fia

[fia]
in : diffuse
width : 200
height : 25
delay :  inertial
border :  nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1) (1,0) (1,1)
localtransition : transport
link : diffuse diffuse@fia(x,y)
PortInTransition : diffuse@fia(x,y) diffusion
...

[transport]
rule : {(0,-1)} { 0.1/( 22.57878* ( 1 - power(cellPos(0)*0.001+0.0005 , 2) / 0.000625 )) * 1000 }
                              { cellPos(1) != 0 }
rule : {0.8} { 0.1 / ( 22.57878 * ( 1 - power(cellPos(0)*0.001+0.0005 , 2) / 0.000625 )) * 1000 }
                              { cellPos(1) = 0 }

[diffusion]
rule : { ((-1,0) + (0,0) + (1,0)) / 3 } 1 { cellPos(0) != 0 AND cellPos(0) != 24 }
rule : { ((-1,0) + (0,0)) / 2 } 1 { cellPos(0) != 0 AND cellPos(0) = 24 }
rule : { ((0,0) + (1,0)) / 2 } 1 { cellPos(0) = 0 AND cellPos(0) != 24 }
```

FIGURE 12.15 Definition of the FIA model.

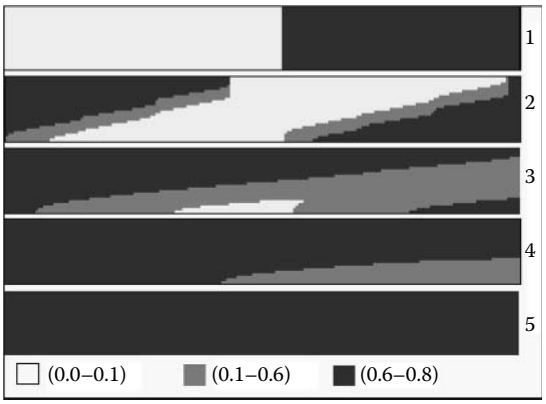


FIGURE 12.16 Different execution stages of the FIA model [14].

in milliseconds for a particle moving at speed $v(r)$ cm/s to travel across a 0.1-cm cell, given by `0.1/v(r)*1000.`

The second rule is used for the left border cells (because cells in column 0 do not have a valid (0,–1) neighbor). For these cells, the new value should be 0.8, which corresponds to the concentration of the carrier solution being pumped into the tube.

When the simulation starts, all cells will evaluate their local transition functions and schedule their next change. A cell in row 2, for instance, will schedule an internal transition at time $t = 4$ ms and a cell in row 3 at $t = 5$ ms. Thus, at $t = 4$ ms, all cells in row 2 will send an output event to their neighbors. Cells in row 3 will receive this event and evaluate the local transition function, which says they should take the value of their left neighbor. However, their left neighbor has not changed yet, so the new value will be the same as the previous *future value*. Therefore, they will keep their scheduled internal transition for $t = 5$ ms. At this time, all cells in row 2 with a scheduled internal transition will send their new value to their neighbors. A cell in row 2 receiving an input from its left neighbor will again evaluate its local transition function. In this case, the delay has already expired and there is no *future value* scheduled, so the result of this evaluation will be scheduled as the *future value* for time $t = 10$ ms.

The *diffusion* rules are in charge of modeling radial diffusion, and they are activated when the generator sends a new value to the *diffuse* input port. For a cell with valid top and bottom neighbors, the diffusion rule states that the new cell value will be the average of the three cells. The following rules cover the special case of top and border cells.

Figure 12.16 shows five different stages in the model's execution after 10 s (only half of the tube is shown because it is assumed that the other half is symmetrical). The upper cells represent the center of the tube, and the lower cells represent the part of the solution touching the walls of the tube. The experiment starts at time 0, where the sample (white) is injected. At this moment, half of the tube contains the carrier solution (dark gray). In the following stages, the convective transport makes the sample disperse faster at the middle of the tube than near the walls. The experiment finishes when the whole tube contains carrier solution.

Exercise 12.6

The simulation results can be used to obtain the conductivity curve for the system. To do so, we must divide the cell space in axial segments, calculate the resistance of each segment, and compute the whole resistance as the result of combining all segments. If we consider that each segment is a column of cells, resistance can be computed as

$$R_{total} = \sum_{column=0}^{199} \left(\sum_{row=0}^{24} \frac{1}{R_{cell(row,col)}} \right)^{-1} \tag{12.9}$$

in which R is the cell's resistance, computed as

$$G_{cell} = \frac{1}{R_{cell}} = G_{HNO_3} + G_{H_2O} = \frac{Area_{cell}}{Length_{cell}} \left(\kappa_{HNO_3} \cdot [HNO_3] \right) \tag{12.10}$$

As we can see, resistance of a cell can be obtained by calculating the inverse of the conductivity. In this case, we use the sizes of the cells and the concentration of nitric acid on each of them. Using the simulation results obtained after running the FIA model and the previous equations, reconstruct the conductivity curve.

12.5 NUMERICAL APPROXIMATION OF HEAT SPREADING

In Chapter 5, we presented two- and three-dimensional models of heat diffusion. In this section, we will introduce different advanced versions of numerical approximations for heat spreading models. Our first example is focused on the application of QDEVS; we then present extensions to the finite elements and finite differences methods and their adaptation using Cell-DEVS.

12.5.1 QDEVS for Heat Spreading

In this section, we discuss the application of QDEVS to the heat spreading model introduced in Chapter 5. The idea is to reduce the level of activity in the simulation model by quantizing the cells in the model. The models were originally presented in Wainer and Zeigler [15] and include different test cases:

(a) a two-dimensional model (10 × 10 cells) with only one "hot" cell;
(b) a similar model, but using 87% of active cells;
(c) a three-dimensional extension of the previous model; and
(d) a dynamic heat seeker—a model consisting of two adjacent planes. One of the surfaces executes the two-dimensional heat diffusion model. The other includes a set of heat-seeking devices that follows the heat cells toward a local maximum (found in *./seekers.zip*).

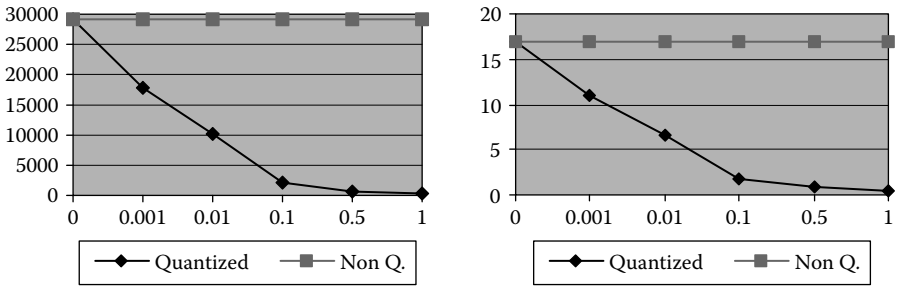
The number of messages involved in the execution for Cell-DEVS spaces can be expressed as follows:

$$m_i = \sum_{j=1}^{i} n_j \cdot \mu \tag{12.11}$$

where

m_i = the number of messages distributed up to the ith simulation step;
n_j = the number of active cells in the jth simulation step; and
μ = neighborhood size.

The results presented in Figure 12.17 show a reduction according to bx^{-a} (with $x \in (0,1]$) in the number of messages involved for the test cases (a) and (b) just discussed. Analyzing Equation (12.11), we can see that n_j is reduced in each time step. In addition, the use of a quantized version provides fewer steps to be executed, reducing the i value in the equation [15].



(a) $f(x) = 441x^{-0.58}$ (b) $f(x) = 0.61x^{-0.45}$

FIGURE 12.17 (a) Number of messages involved; (b) execution time.

These results approximate the theoretical optimum results presented in Zeigler [16]. Message reductions for model (b) were slightly less than for model (a) case because the number of active initial cells is higher. Therefore, in the first simulation steps, n_j is greater than the previous case. The total execution time can be expressed as:

$$t_i = \sum_{j=1}^{i} \left[(n_j \cdot \mu \cdot x_j) + (n_j \cdot \tau_j) \right] \tag{12.12}$$

where

t_i = the total execution time up to the ith simulation step;
n_j = the number of active cells in the jth simulation step;
x_j = the transmission time for each message;
μ = the neighborhood size; and
τ_j = the execution time for the local computing function.

The curve shapes for all the executed examples are similar to those in Figure 12.17, as discussed in Wainer and Zeigler [15]. The results obtained in models (a) and (b) are proportional to those obtained analyzing the number of messages involved. In this case, the execution and transmission times for each cell are equivalent. The results obtained with larger quantum have increased proportionally to the number of messages involved. The error behavior of these models can be expressed as

$$e(C_c, i) = \sum_{j=1}^{i} \left| \tau_{cj}(N_c) - \left[\tau_{cj}(N_c) \right]_q \right| \tag{12.13}$$

Here, $e(C_c, i)$ is the accumulated error up to the ith simulation step in cell C_c (c is an n-dimensional index of the cell). N_c are the inputs of the cell c, τ_{cj} is the execution result of jth step of the local computing function of cell c, and $[\ ]_q$ represents the quantized value of the last change. The error obtained is thus a function of the local computing function, the number of simulation steps, and the quantum. The use of a higher quantum reduces the number of steps, but each of them will have higher error rates. The experimental results validate this behavior, as seen in Figure 12.18. It can be seen that the error grows as $f(x) = ax^b$. This error can be linear when there is no influence between cells. In the figure we can see that in the (a) case, the error hardly increases, while the messages go down by approximately 1/10. Nevertheless, the error can lead to undesired behavior.

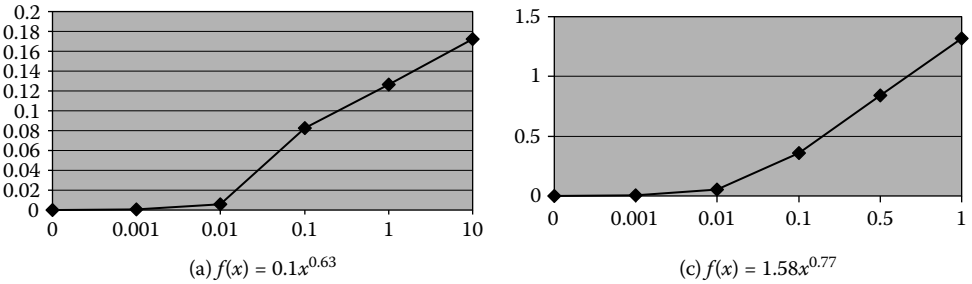


FIGURE 12.18 Accumulated error behavior.

12.5.2 Heat Approximation Using Discrete-Event Finite Elements

Finite element analysis [17,18] is a numerical approximation method to solve differential equations that has been used successfully to analyze complex engineering and physical systems. Typical areas of applications include structural analysis, thermal analysis for heat transfer, electromagnetic analysis, fluid analysis, etc. The Finite Elements Method (FEM) defines a solution that satisfies the partial differential equation on average over only a finite portion of the field under analysis. Every portion (or element) is connected to neighboring elements, and the field under study is analyzed by propagating the current values from one element to another through connection points. FEM provides piecewise approximation of a partial differential equation over a continuum, and a finite element is a discrete piece of that continuum. By assuming a simple function over the finite element, we can approximate the solution of the partial differential equation over that element.

We usually find two major components that can be identified within each element: **field** and **potential**. The field is a quantity that varies with its position within the structure analyzed. The fields are related to the potentials as their derivatives with respect to position. The potential can be thought of as the driving force for the spread of the field in the material. For example, temperature difference in a material would cause a heat flux to be transferred from one point to another in that material. The heat flux direction and quantity are related to the difference in temperature in that material (temperature gradient). Elements in the structure are considered to be connected together through the vertices on boundaries of each of the elements, which are called **nodes**.

In order to solve the problem:

1. We must divide the structure under study into a large number of elements, each of them with a simple geometry.
2. An interpolation function is assumed over the element, representing the shape of the spatial solution in the element.
3. The differential equations can be solved for this particular element by assuming the shape of the change of potential function in the element. This gives an approximate solution for a single element: simple algebraic equations are obtained for the element, represented in a matrix.
4. Because all the elements in the structure are connected together through nodes located at their edges, we obtain a system of equations represented in $N \times N$ matrices. We only know the values at certain points in the structure (usually at its boundaries). These values are used to get the unknown potential inside the structure.
5. The global equations are solved, and the solution gives the distribution of the potential over the structure, represented by the values obtained at the nodes of each element. The precision can be enhanced by dividing the structure into more elements or by assuming a more precise distribution of the potential inside the element itself.

FEM models resemble, to a large extent, Cell-DEVS models, in which changes of a cell value would trigger neighboring cells to change themselves, as though a field is propagating through all of them. The following sections will be devoted to showing how Cell-DEVS can be used to describe FEM models. The idea is to describe the model in terms of cell behavior, discrete event interaction, and timing delays. We will show how to use Cell-DEVS to model and solve problems usually tackled by FEM, having FEM precision for defining the problem, and the simplicity of a cellular approach to facilitate model definition.

12.5.2.1 One-Dimensional Heat Transfer: Mapping FEM into Cell-DEVS

In this section, we show how FEM models can be mapped into Cell-DEVS using a traditional example found in Chandrupatla and Belegundu [17] and presented in Saadawi and Wainer [19,20]. This model represents steady-state heat transfer with convection from a fluid into a composite wall of different materials (i.e., the heat flux is fixed with regard to time, as opposed to non-steady-state heat transfer, where temperature distributions change over time). This resembles the heat flow through a wall of a heated furnace to ambient air.

Heat transfer occurs when there is a temperature difference within a body or between a body and its surrounding medium. This temperature difference constitutes the potential driving the heat flux through the material. The temperature difference over an infinitely small piece of material would give us the *temperature gradient* over this element. Heat flows from hot spots toward cooler ones. Heat conduction in a two-dimensional, steady-state isotropic medium is given by Fourier's law:

$$q_x = -k\frac{\partial T}{\partial x},\ q_y = -k\frac{\partial T}{\partial y} \tag{12.14}$$

where

q = the heat flux (W/m^2);
q_x = the heat flux component in the x direction;
q_y = the heat flux component in the y direction;
k = the thermal conductivity of the material (W/m°C);
$T = T(x,y)$ is the temperature field in the medium and is a function in x and y;
$\partial T/\partial x$ and $\partial T/\partial y$ are the temperature gradients over x and y, respectively; and
the minus sign indicates that the direction of heat flux is opposite to the direction of increasing temperature [17].

In convection heat transfer, heat flux is given by

$$q = Ah(T_\infty - T_s) \tag{12.15}$$

where

h (W/m^2°C) is the film (a property of the fluid around the surface);
T_∞ and T_s are fluid and surface temperature, respectively; and
A is the surface area exposed to the flow.

For a small element assuming a linear temperature distribution along its unit length and a unit area perpendicular to heat flow direction, the heat flux conduction would be

$$q = k\frac{dT}{dx} = k\frac{(T_h - T_l)}{1} = k(T_h - T_l) \tag{12.16}$$

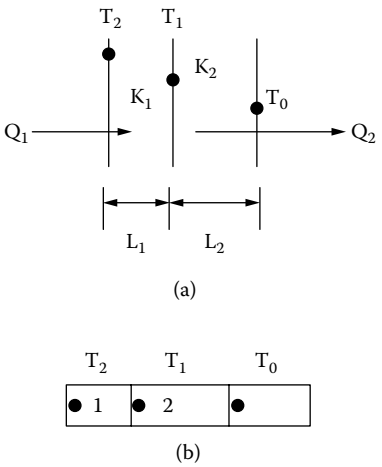


FIGURE 12.19 (a) Two elements physically connected; (b) elements represented as finite elements/nodes.

where T_l is the high and low temperature of its ends [17].

In order to get the updating rules for a cell in Cell-DEVS, we first show a subset of the complete problem. Figure 12.19 shows two layers of the wall connected together through the surface in the middle. Each layer i has different physical properties: K_i is the thermal conductivity, L_i is the length, and Q_i is the heat flux through that wall. Temperature distribution on each surface on the walls is denoted as T_2, T_1, and T_0, as shown in Figure 12.19(a) [19].

Each layer can be represented by one finite element. Elements 1 and 2 contain two nodes, one at each end; the elements are connected through their nodes, and the middle one is shared between them. Every node represents a surface of a wall and the corresponding node value represents its surface temperature. From Equation (12.16), by assuming a linear temperature distribution along the elements, we get

$$Q_1 = K_1/L_1.\ (T_2 - T_1) \tag{12.17}$$

$$Q_2 = K_2/L_2.\ (T_1 - T_0) \tag{12.18}$$

Due to the conservation of energy equation over a control volume containing both elements 1 and 2 (input heat flux equals output heat flux), we have $Q_1 = Q_2$ and we obtain

$$T_1 = \frac{\frac{K_1}{L_1}T_o + \frac{K_2}{L_2}T_2}{\frac{K_1}{L_1} + \frac{K_2}{L_2}} \text{ for heat conduction} \tag{12.19}$$

Similarly, when we study two elements in which one is convective and the other is conductive, we get

$$T_1 = \frac{hT_\infty + \frac{K_1}{L_1}T_o}{h + \frac{K_1}{L_1}} \text{ for heat convection} \tag{12.20}$$

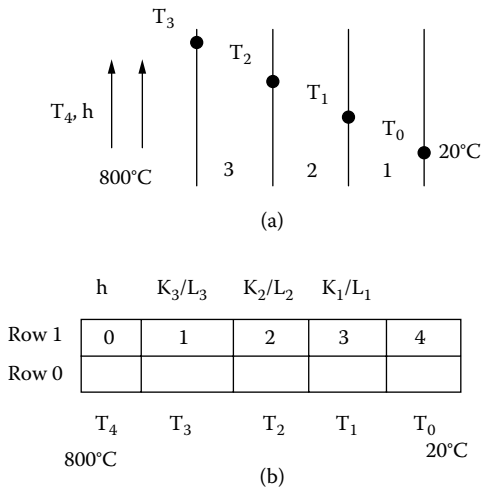


FIGURE 12.20 (a) Steady-state heat transfer through a composite wall; (b) the problem as a Cell-DEVS space [20].

Equations (12.19) and (12.20) can be used as the updating rules for a Cell-DEVS model. Equation (12.19) describes the heat conduction rule inside the material, specifying the middle node T_1 temperature as a function of its two adjacent nodes and constant material properties. Equation (12.20) describes the middle node temperature as a function of adjacent nodes of fluid temperature T_∞ and inner node temperature T_0 inside the material. This represents the case as at a convective boundary, and T_1 is the surface temperature. Every cell value would thus be a function of its right cell value, its stored physical properties, its left cell value, and its left cell physical properties. Note that in the case of having identical elements (same K and L), the updating rule for a cell's temperature would be a simple arithmetic mean of its two neighboring cell temperatures, as in the model discussed in Chapter 5, section 5.5.).

Figure 12.20 shows an extended version of the example presented in Chandrupatla and Belegundu [17]. Figure 12.20(a) represents a composite wall of three materials. The outer temperature is T_0 = 20°C, and convection heat transfer takes place on the inner surface of the wall with temperature T_4 = 800°C and film coefficient h = 25 W/m^2°C. We need to determine the temperature distribution in the wall (i.e., on the surface of each layer). Composite layer lengths are L_1 = 0.3 m and $L_2 = L_3$ = 0.15 m. Conductivities are K_1 = 50 W/m°C, K_2 = 25 W/m°C, and K_3 = 20 W/m°C for layers numbered 1, 2, and 3, respectively.

We defined the complete Cell-DEVS model in CD++, which can be found in *./heatFEM.zip*. Figure 12.21 presents the model's rule definition. The model represents each point of the temperature measure as in Figure 12.20, and the computing cells are those in the bottom row. Thus, cell (0,0) represents fluid temperature T_4 (T_3 = (0,1), T_2 = (0,2), T_1 = (0,3), and T_0 = (0,4)). Cells in row 1 store the physical properties corresponding to wall layers, and they are constant: cell (1,0) contains h, (1,1) contains K_3/L_3, (1,2) contains K_2/L_2, (1,3) contains K_1/L_1, and (1,4) is not used. Cells (0,0) and (0,4) contain constant temperature (boundary conditions). The model is initially loaded with values representing material properties on row 0, boundary values in cells (1,0) and (1,4), and arbitrary values in other cells.

The *conduction-rule* implements Equations (12.19) and (12.20). It also works for convection, as it uses the film coefficient value instead of thermal conductivity at cells adjacent to fluid cells. The *constants* define the updating rule for cells in row zero, while *boundary* defines the updating rule for boundary cells.

In order to test the model, we execute the model until the cell values become stable, converging to a solution for our problem. The resulting values would represent temperature distribution over the

```
[heatcond]
type : cell  dim : (2,5) delay : transport
border : nowrapped
neighbors : (-1,0) (0,0) (0,1) (-1,-1) (0,-1)
localtransition : conduction-rule
zone : Constants { (0,0)..(0,4) }
zone : Boundary { (1,0) (1,4) }

[conduction-rule]
rule : { ((-1,0)*(0,1)+(-1,-1)*(0,-1)) / ( (-1,0)+(-1,-1) ) } 1 { t }
; Implements Equations 12.19 and 12.20 Physical properties K/L are stored in (-1,0) and
                                                                              (-1,-1).

[Constants]
rule : {(0,0)} 1 {t}
; Constant physical properties stored in row 0 as defined in a "zone" in the model.

[Boundary]
rule : {(0,0)} 1 {t}              ; boundary conditions.
```

FIGURE 12.21 CD++ rules for the heat transfer model.

```
Line : 29 - Time: 00:00:00:000
             0          1            2            3             4
 +-------------------------------------------------------------------+
0|  25.00000000  66.66666412 200.00000000 333.33334351   1.00000000|
1| 800.00000000  20.00000000  20.00000000  20.00000000  20.00000000|
 +-------------------------------------------------------------------+

Line : 44 - Time: 00:00:00:001
             0          1            2            3             4
 +-------------------------------------------------------------------+
0|  25.00000000  66.66666412 200.00000000 333.33334351   1.00000000|
1| 800.00000000 232.72727966  20.00000000  20.00000000  20.00000000|
 +-------------------------------------------------------------------+
...
Line : 473 - Time: 00:00:00:023
             0          1            2            3             4
 +-------------------------------------------------------------------+
0|  25.00000000  66.66666412 200.00000000 333.33334351   1.00000000|
1| 800.00000000 304.74679565 119.02681732  57.13505936  20.00000000|
 +-------------------------------------------------------------------+
```

FIGURE 12.22 Simulation results for initial temperature = 20°C.

structure. The example presented in Figure 12.22 uses a temperature of 20°C inside the wall as the initial value. The final step shows T_3 = 304.75°C, T_2 = 119.03°C, and T_1 = 57.14°C, corresponding to nodes' temperatures.

Exercise 12.7

Initialize T_3, T_2, and T_1 with temperatures of 3000°C. Analyze whether the results converge into the final correct answer. Compare with the number of steps in Figure 12.22.

Exercise 12.8

Considered what happens when the hot fluid temperature T_4 is 850°C.

Exercise 12.9

Initialize the cells with arbitrary values and repeat the experiments. Analyze the results.

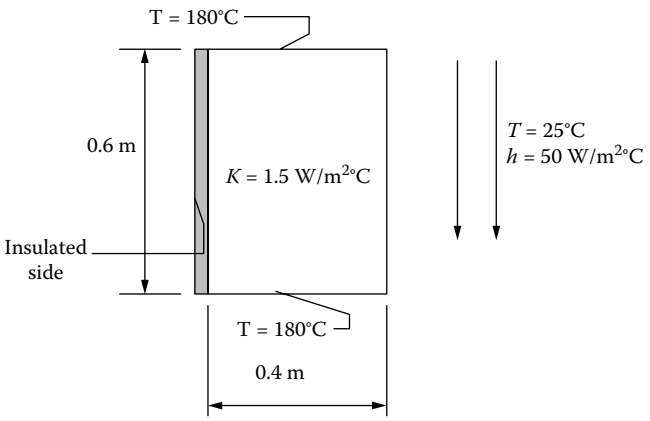


FIGURE 12.23 Steady-state heat transfer in a long bar [20].

12.5.2.2 Two-Dimensional Heat Transfer with Cell-DEVS

In this section, we will show how to expand the heat transfer model presented in the previous section into a two-dimensional model [20]. In order to do so, we need to describe every cell's local transition function as explained in the previous section. Let us consider the heat transfer model originally presented in Chandrupatla and Belegundu [17]. This example represents a steady-state, two-dimensional heat transfer in a bar of rectangular cross-section with thermal conductivity coefficient k = 1.5 W/m²°C. Two opposite sides are kept at constant temperature of 180°C; one side is insulated and the other is exposed to a fluid with temperature of 25°C and convection heat transfer coefficient h = 50 W/m²°C. A graphical representation of the problem is depicted in Figure 12.23 [20,21].

A steady heat transfer without heat generation in the body in two dimensions is represented by equations in the previous section and the following diffusion equation [17]:

$$k\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}\right)=0 \tag{12.21}$$

To solve the preceding equation, we need to get the second derivative of the temperature gradient. To do so, we study a steady-state heat transfer in a long rod, as represented in Figure 12.24. In this figure, we study a very small section of a one-dimensional rod. Points A, B, and C along the rod have corresponding temperatures of T_1, T_2, and T_3, respectively. Distances between points in the section are as indicated in the figure. To get the temperature gradient along a small section, we assume a linear temperature change in the x direction over the very small finite space Δx:

$$\frac{\partial T}{\partial x}=\left(\frac{T_1-T_2}{\Delta x}\right) \tag{12.22}$$

is the temperature gradient at point A and

$$\frac{\partial T}{\partial x}=\left(\frac{T_2-T_3}{\Delta x}\right) \tag{12.23}$$

is the temperature gradient at point B. Thus, the temperature gradient at point C is

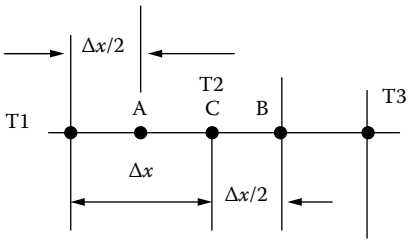


FIGURE 12.24 Heat transfer in one dimension [20].

$$\frac{\partial^2 T}{\partial x^2} = \frac{\left[\left(\frac{T_1 - T_2}{\Delta x}\right) - \left(\frac{T_2 - T_3}{\Delta x}\right)\right]}{\Delta x} = \frac{T_1 + T_3 - 2T_2}{\Delta x^2} \tag{12.24}$$

By applying the previous result in a two-dimensional space, we can approximate the solution of the previous PDE as

$$k\left[\left(\frac{T_1 + T_3 - 2T_0}{a^2}\right) + \left(\frac{T_2 + T_4 - 2T_0}{a^2}\right)\right] = 0 \tag{12.25}$$

From this we obtain

$$T_1 + T_2 + T_3 + T_4 - 4T_0 = 0 \tag{12.26}$$

This equation relates node temperatures of the grid, giving us the updating rule for an internal node as

$$T_0 = \frac{T_1 + T_2 + T_3 + T_4}{4} \tag{12.27}$$

We still need the updating rules for a point on the insulated surface and on the convective side of the rod. Using a similar procedure, we can obtain

$$T_0 = \frac{T_1 + T_3 + 2T_2}{4} \tag{12.28}$$

$$T_0 = \frac{T_1/2 + T_3/2 + T_2 + Tf \times h \times a/k}{2 + h \times a/k} \tag{12.29}$$

We can apply a similar method to deduce Cell-DEVS model updating rules from finite elements. The idea is to apply the equations of steady state heat transfer to a two-dimensional triangular finite element. To do so, we used a linear change function of the field under study over the triangular element. We used the triangular element depicted in Figure 12.25 (called *constant strain triangular*), which was used historically to analyze body strain problems.

The element in Figure 12.25 has three nodes, each at a vertex of the triangle (containing temperature values). Any internal point in the element as P is evaluated as a function of the values of the three nodes. We use a linear function over the element for the field under study; that is,

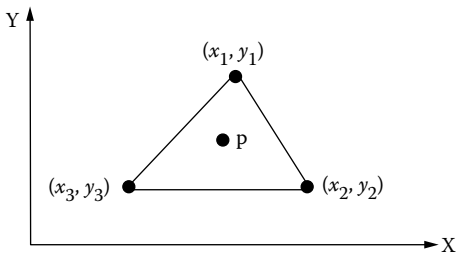


FIGURE 12.25 Triangular element.

$Tp = N_1T_1 + N_2T_2 + N_3T_3$, where N1, N2, and N3 are linear functions. In order to get the updating rules for the Cell-DEVS model, we construct a mesh of elements to represent a recurring pattern inside the structure under study, as in Figure 12.26.

The middle node 0 is shared among all the elements; thus, its value would be a function in all other elements. By studying this structure, we would be able to deduce the updating rules for node 0, which would then repeat for all similar internal nodes. Using similar techniques to the ones explained in this and previous sections, we were able to find the equations corresponding to the local rules of our Cell-DEVS model, which resembles the result obtained using the finite differences method for an internal node:

$$T0 = \frac{T_5 + T_1 + T_2 + T_4}{4} \tag{12.30}$$

Figure 12.27 shows how to apply these equations to the model introduced in Figure 12.23. As shown in Figure 12.27, we divide the bar into a grid of 6 × 4 (i.e., 7 × 5 nodes located at every intersection in the grid). The updating rules we presented previously are used to model each node on the grid.

Figure 12.28 represents the model definition in CD++, found in *./2dheatconduction.zip,* considering that cells on constant temperature boundaries are initialized with a value of 180°C.

The `conduction-rule` uses Figure 12.27, defined previously. Likewise, `Insulated-Boundary` uses Figure 12.28, and `Convective` uses Figure 12.29. When we executed this model, we obtained the results shown in Figure 12.29. We can see that at the start of the simulation, cells (0,0) to (0,4) and (6,0) to (6,4) all had a value of 180, and the rest of the cells had a value of 20. At the end, cell values did not change. At this step, cell values would represent the solution of the problem—namely, the temperature distribution though the bar.

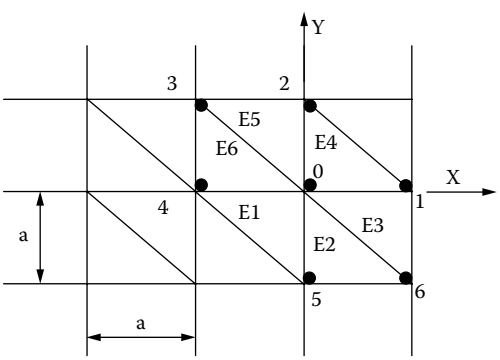


FIGURE 12.26 Internal mesh of triangular elements [20].

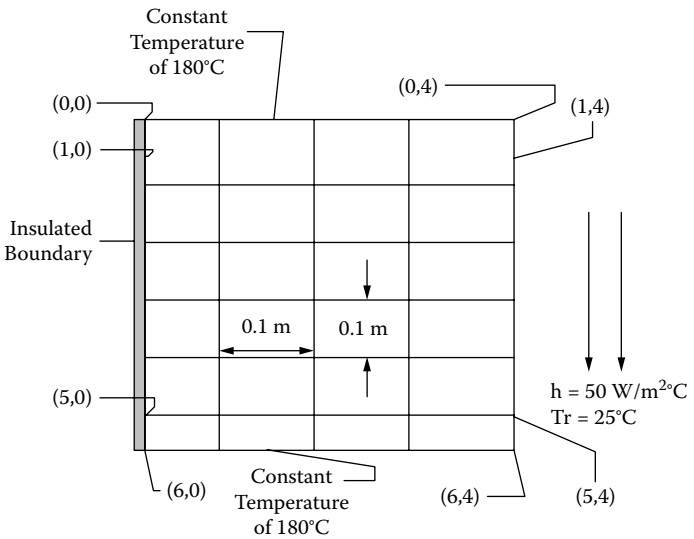


FIGURE 12.27 7 × 5 node grid of finite differences [20].

```
[heatcond]
type : cell     dim : (7,5)
delay : transport  border : nowrapped
neighbors : (-1,0) (0,0) (0,1) (1,0) (0,-1)
localtransition : conduction-rule
zone : Insulated-Boundary { (1,0)..(5,0) }
zone : Constant-Temp { (0,0)..(0,4) }
zone : Constant-Temp { (6,0)..(6,4) }
zone : Convective { (1,4)..(5,4) }

[conduction-rule]
rule : { ((0,1)+(-1,0)+(0,-1)+(1,0)) / 4 } 1 { t }

[Insulated-Boundary]
rule : {((-1,0)+(1,0)+2*(0,1)) / 4} 1 {t}

[Constant-Temp]
rule : {(0,0)} 1 {t}

[Convective]  % Fluid temperature: 25, and h.a/k is (10/3)
rule : {( ((1,0)/2)+((-1,0)/2)+(0,-1)+ (25 * (10/3)) )/(2 + (10/3)) } 1 {t}
```

FIGURE 12.28 Model definition in CD++ for 7 × 5 node grid.

12.5.3 Lattice Gas Models

Hardy, de Pazzis, and Pomeau [22] introduced the HPP lattice gas automata model interaction potential between particles in order to mimic molecular dynamics. Different lattice gas models have been used in applications in chemistry (fluid phase separation, miscible fluids, viscosity) and physics (colloids, optics, porous media analysis, hydrodynamics) [23], although other models have been used in biology and medicine applications [1]. The idea is that a number of particles are placed into a grid, with a fixed speed and mass. Particles interact through local instantaneous collisions, conserving mass and momentum, as shown in Figure 12.30.

In each step, particles move to their nearest neighbors following their current direction. If there is a chance of collision, particles change direction. As we can see in Figure 12.30, when particles

```
Line : 91 - Time: 00:00:00:000
     0  1  2  3  4
  +--------------------------------+
  0| 180.0 180.0 180.0 180.0 180.0 |
  1|  20.0  20.0  20.0  20.0  20.0 |
  2|  20.0  20.0  20.0  20.0  20.0 |
  3|  20.0  20.0  20.0  20.0  20.0 |
  4|  20.0  20.0  20.0  20.0  20.0 |
  5|  20.0  20.0  20.0  20.0  20.0 |
  6| 180.0 180.0 180.0 180.0 180.0 |
  +--------------------------------+
   . . .
Line : 29358 - Time: 00:00:00:204
     0  1  2  3  4
  +-------------------------------+
  0| 180.0 180.0 180.0 180.0 180.0|
  1| 158.4 155.3 144.1 118.7  58.7|
  2| 143.3 138.4 122.6  92.0  42.1|
  3| 137.9 132.6 115.6  84.8  39.4|
  4| 143.3 138.5 122.6  92.0  42.1|
  5| 158.4 155.3 144.1 118.7  58.7|
  6| 180.0 180.0 180.0 180.0 180.0|
  +-------------------------------+
```

FIGURE 12.29 Results for 7 × 5 node grid of finite differences.

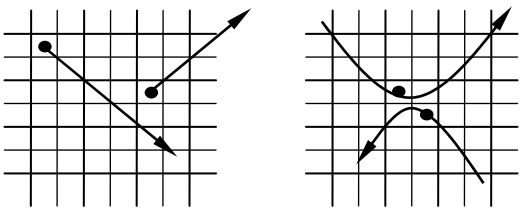


FIGURE 12.30 HPP lattice gas particle behavior.

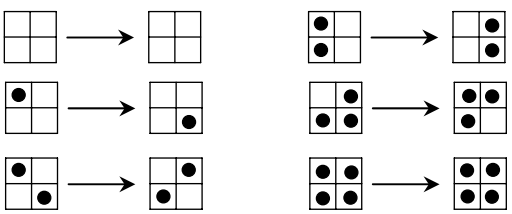


FIGURE 12.31 Rules for the HPP gas model.

are isolated, they keep their current direction, and whenever they collide, they change direction to the opposite diagonal.

Lattice gas models are usually implemented using a Margolus neighborhood, which was introduced in Chapter 3 (we use one block of cells in each simulation step, and we switch to a different block in the next). The lattice gas model defines a mechanism for the uniform movement of the particles. These particles are identical and move at the same speed. Energy must be conserved; thus, the number of particles in the model should be constant. Figure 12.31 shows a description of the rules we used for the lattice gas model we defined.

The first and last rules define nonchanging cells' behavior. The remaining ones define diagonal movement and collision detection. In order to build a Cell-DEVS model with a Margolus

```
[top]
type : cell
dim : (40,40,2)
delay : transport
border : wrapped
neighbors : (-1,-1,0) (-1,0,0) (-1,1,0) (0,-1,0) (0,0,0) (0,1,0) (0,0,1) (1,-1,0) (1,0,0)
            (1,1,0)
localtransition : calculus

[calculus]
rule : { (1,1,0) } 100 { cellpos(2)=0 and ((even(cellpos(0)) and even(cellpos(1)) and
         (0,0,1)=0) or (odd(cellpos(0)) and odd(cellpos(1)) and (0,0,1)=1)) }
rule : { (1,-1,0) } 100 { cellpos(2)=0 and ((even(cellpos(0)) and odd(cellpos(1)) and
         (0,0,1)=0) or (odd(cellpos(0)) and even(cellpos(1)) and (0,0,1)=1)) }
rule : { (-1,1,0) } 100 { cellpos(2)=0 and ((odd(cellpos(0)) and even(cellpos(1)) and
         (0,0,1)=0) or (even(cellpos(0)) and odd(cellpos(1)) and (0,0,1)=1)) }
rule : { (-1,-1,0) } 100 { cellpos(2)=0 and ((odd(cellpos(0)) and odd(cellpos(1)) and
         (0,0,1)=0) or (even(cellpos(0)) and even(cellpos(1)) and (0,0,1)=1)) }

rule : 1 100 { cellpos(2)=1 and (0,0,0)=0 }
rule : 0 100 { cellpos(2)=1 and (0,0,0)=1 }
```

FIGURE 12.32 Implementing the HPP gas model.

neighborhood, we used a three-dimensional model, in which plane 0 was used to model the desired behavior and plane 1 was used to determine which neighbors are used on each step (i.e., those in the even/odd grid). We used Moore's neighborhood in the two planes. Figure 12.32 represents the implementation of this model in CD++, found in *./HPP.zip*.

As we can see, the model is a three-dimensional lattice of 40 × 40 × 2 cells. The cells take the value of their opposite neighbor in a diagonal direction, as dfined in Figure 12.30. The following rules in the figure define, for each of the four cells, which value to take and when.

The model's rules show that we evaluate the rules in two different ways, according to the current position of the cell and the simulation step. In one case, we check whether the cell is on an even row and column; in this case, we evaluate the neighbors in line with the even grid. In the second case, the cell is on an odd row or column, and we analyze the neighbors in line with the odd grid. In either of the two cases, the cell takes the value of the neighbor (1,1,0), as we can see in Figure 12.33. Using the *even* and *odd* functions allows us to determine the parity of a given value combined with the *cellpos* function, which allows us to determine the position of a cell in any plane. For instance, the first rule checks whether we are on an even row or column and whether the cell on top of it is empty (which means we are using the even grid). If that is not the case, we see whether we are on an odd row or column and using the odd grid ((0,0,1) = 1). In either of these cases, the cell will tack the value of the cell to the SE. The rules in Figure 12.33 are symmetric for movement from the NW. The last two rules deal with collisions.

The next rules use the same idea, but a different cell is evaluated. The last rules in the model switch the values of the lower plane, which is used to determine which grid we are using (if the

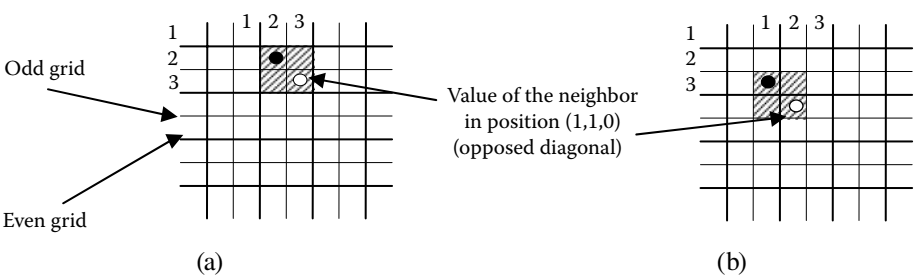


FIGURE 12.33 (a) Cell in a position with even row and column position, in line with the even grid; (b) cell in a position with odd row or column, in line with the odd grid.

value is zero, we consider the cells aligned with the even cell; if the value is one, we use the cells aligned with the odd cell).

12.6 A THREE-DIMENSIONAL MODEL OF VIRTUAL CLAY

Representation of solid three-dimensional objects usually requires using restrictive geometrical operations, and they are usually based on strict physical models, such as finite element methods, methods based on elasticity theory, and applications of particle systems. All these methods and applications need considerable time for computing deformations according to the laws, and human interactions are not permitted, especially for complex shapes.

Instead, it has been proposed to represent three-dimensional objects as clay that can be freely deformed, in order to understand problems on three-dimensional objects [24–26]. Some of the ongoing efforts considering volume sculpting in a three-dimensional virtual space use a discretization of the space in two- or three-dimensional cells. Arata and colleagues [25] used three-dimensional cellular automata (CA) to simulate plastic deformations of clay, and each cell is allocated a finite state automaton, which is given the simple distribution rules of the virtual clay instead of complicated physical laws. An extension presented in Druon, Crosnier, and Brigandat [26] includes new repartition algorithms. We will show how to model such a three-dimensional free-form object using Cell-DEVS based on the state transition rules presented in Arata et al. [25]. This model, originally presented in Wu, Wu, and Wainer [27] and found in *./plastic.zip,* describes effectively the behavior of a free-form object: compression (from outside) and deformation (from inside).

In virtual clay models, three-dimensional object deformation is considered as a physical process that equally distributes the virtual clay to the adjacent areas. A threshold is associated with the deformation of the object; when the density is under the threshold, the object keeps its shape. If a portion receives an external force, its density changes; if the density is above the threshold, the object is deformed and clay is transported from high-density to low-density portions. However, the total mass of the clay should be conserved.

Arata and colleagues [25] define the model using a Margolus neighborhood and the following rules for each block:

[Step A] For each cell i whose state is 1,
$dm_i = m_i \times \alpha$; and
$m_i = m_i - dm_i$.
[Step B] For each cell j whose state is 0,
$m_j = m_j + ((dm_1 + dm_2 + \ldots + dm_t)/n)$

where α is a constant rate for distribution ($0 < \alpha < 1$), t is the number of cells over threshold and n is the number of cells under threshold. Here, we denote the state of a cell as one if its virtual clay is over the threshold. Otherwise, the state is zero. The value dm_i represents the excess of clay in cell i, which will be distributed to the neighboring cells. From these two steps, we can see that the total mass of virtual clay within a block is conserved during the state transitions. Figure 12.34 illustrates the transition rules in two dimensions.

The deformation of a virtual clay object is based on a push operation. The clay is transported from a cell into the adjacent ones along the direction of pushing. The surface of a virtual clay object can be pushed at most one cell in depth per step, as seen in Figure 12.35.

We used Cell-DEVS to simulate the behavior of a three-dimensional free-form object built as an extension to the rules in two dimensions. Figure 12.36 illustrates the three-dimensional Margolus neighborhood we used, in which the nearest eight cells make one block. The values in each cell represent the mass of that cell. A cell with a value of zero means this cell is out of the object, and a positive value means the cell is within the object. The final state contains the free-form object in stable state after deformation. The transition procedure is done in two stages:

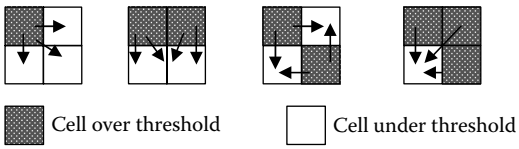


FIGURE 12.34 Two-dimensional block patterns.

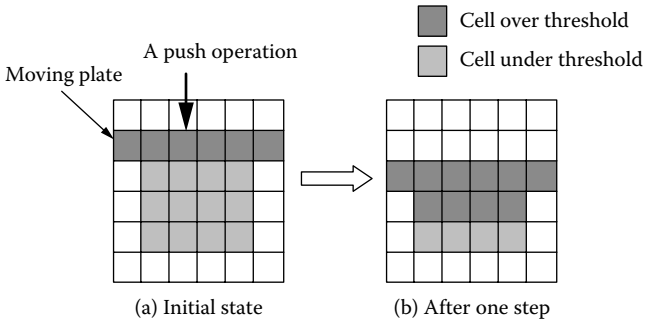


FIGURE 12.35 Push operation by a moving plate.

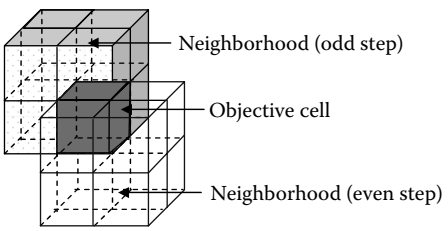


FIGURE 12.36 Three-dimensional Margolus neighborhood.

- Deformation: If there are cells with density over the threshold, the deformation transition rules are applied.
- Compression: We assume that there is a moving plate and the virtual clay next to the plate is transferred from a cell into the adjacent cell along the direction of pushing. In the model, this plate is handled as a dummy plate with each cell having a value of zero, staying right on top of the object.

During the transition procedure, each cell uses its different neighbors at odd and even steps. The neighborhood of each cell is identified according to its location in the block. Meanwhile, each cell has different transition policies for the deformation and compression stages. Therefore, we used a four-dimensional Cell-DEVS model, in which each three-dimensional hyperplane of (x, y, z) represents a set of state variables and the fourth dimension is a control variable, as follows:

- Cube 0 (x, y, z, 0) represents the free-form object.
- Cube 1 (x, y, z, 1) defines the odd or even step so that each cell in cube 0 can identify its Margolus neighborhood.
- Cube 2 (x, y, z, 2) is used for control. Compression will be performed if (x, y, z, 2) = 1 and deformation if cell (x, y, z, 2) = 0.

```
[plastic]
dim : (10,9,12,3)   delay : transport
border : nowrapped
neighbors:(-1,-1,-1,0)(-1,0,-1,0)(-1,1,-1,0)(0,-1,-1,0)(0,0,-1,0)(0,1,-1,0)
    (1,-1,-1,0)(1,0,-1,0)(1,1,-1,0)(-1,-1,0,0)(-1,0,0,0)(-1,1,0,0)
localtransition : deformation-rule

[deformation-rule]
...

[compression-rule]
%plate moving. step 1: add the first row to the second row
rule : {(0,0,0,0)+(0,0,1,0)} 100 {(0,0,0,2)=1 and cellpos(3)=0 and cellpos(0)>0 and
     cellpos(0)<9 and cellpos(1)>0 and cellpos(1)<8 and cellpos(2) <10 and
     ((-1,-1,2,0)+(-1,0,2,0)+(-1,1,2,0)+(0,-1,2,0)+(0,0,2,0) +(0,1,2,0)+(1,-1,2,0)+(1,0,2,0)
     +(1,,1,2,0))=0 and ((-1,-1,1,0)+(-1,0,1,0)+(-1,1,1,0)+(0,-1,1,0)+(0,0,1,0)+(0,1,1,0)
     +(1,-1,1,0)+(1,0,1,0)+(1,1,1,0))>0 }

%step2 :change the first row to 0, the plate has moved one step further
rule : 0 100 {(0,0,0,2) = 1 and cellpos(3)=0 and cellpos(0)>0 and cellpos(0)<9 and
     cellpos(1)>0 and cellpos(1)<8 and cellpos(0) < 11 and ((-1,-1,1,0)+ (-1,0,1,0)+(-
     1,1,1,0)+(0,-1,1,0)+(0,0,1,0) +(0,1,1,0)+(1,-1,1,0) +(1,0,1,0)+(1,1,1,0))=0 and ((-1,-
     1,0,0)+(-1,0,0,0)+(-1,1,0,0)+(0,-1,0,0) +(0,0,0,0)+(0,1,0,0)+(1,-1,0,0)
     +(1,0,0,0)+(1,1,0,0))>0}

%plate moving
rule : 1 100 {(0,0,0,1)=0 and cellpos(3)=1 and (0,0,0,0)=0 }
rule : 0 100 {(0,0,0,1) = 0 and cellpos(3)=1 and (0,0,0,0)=1} %alternate Margolus neighborhood
rule : 1 3000 { cellpos(3)=2 and (0,0,0,0)=0 }
rule : 0 100 {cellpos(3)=2 and (0,0,0,0)=1}
```

FIGURE 12.37 Cell-DEVS coupled model specification in CD++.

The definition of this Cell-DEVS coupled model using CD++ is illustrated in Figure 12.37 and found in *./3dFreeForm.zip*. This model uses three cubes of $10 \times 9 \times 12$ each. The cell's deformation phase uses the following rules:

1. Perform deformation if cell (0, 0, 0, 2) = 0 and this cell is on cube 0.
2. Perform compression if cell (0,0,0,2) = 1 and this cell is on cube 0.
3. Even/odd step alternates if cell (0, 0, 0,1) = 0 and this cell is on cube 1.
4. Deformation/compression control alternates if this cell is on cube 2.

The deformation stage involves activating different rules at odd and even steps, in which we decide the different neighbors from which the objective cell receives clay or to which it distributes clay. Only cube 0 (x, y, z, 0) performs the deformation transition; cube 1 (x, y, z, 1) helps to judge whether the cell in cube 0 changes in the odd or in the even step. Cube 2 (x, y, z, 2) identifies the deformation stage. In Figure 12.38 we show the mechanism to select the Margolus block. The origin cell is colored in gray and its neighbors are defined according to the coordinates shown in the figure. We repeat the procedure for other cells in the same Margolus block to obtain the neighbors of each objective cell in the same hyperplane.

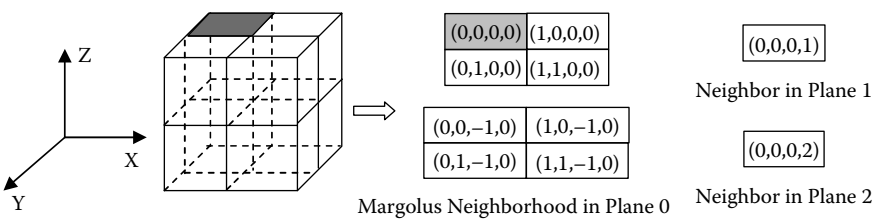


FIGURE 12.38 A cell and its neighborhood definition at the deformation stage.

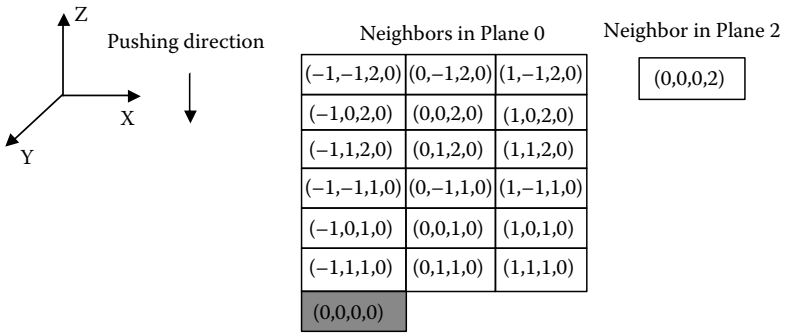


FIGURE 12.39 A cell and its neighbor definition at the compression stage.

The pair < (x,y,z,0), (x,y,z,1) > defines the step (odd or even). The neighbor on cube 0 is described in the figure, and its value, together with the values of all the neighbors in the same Margolus block, can be used to decide which transition rule should be applied. The deformation rules can be generalized as follows:

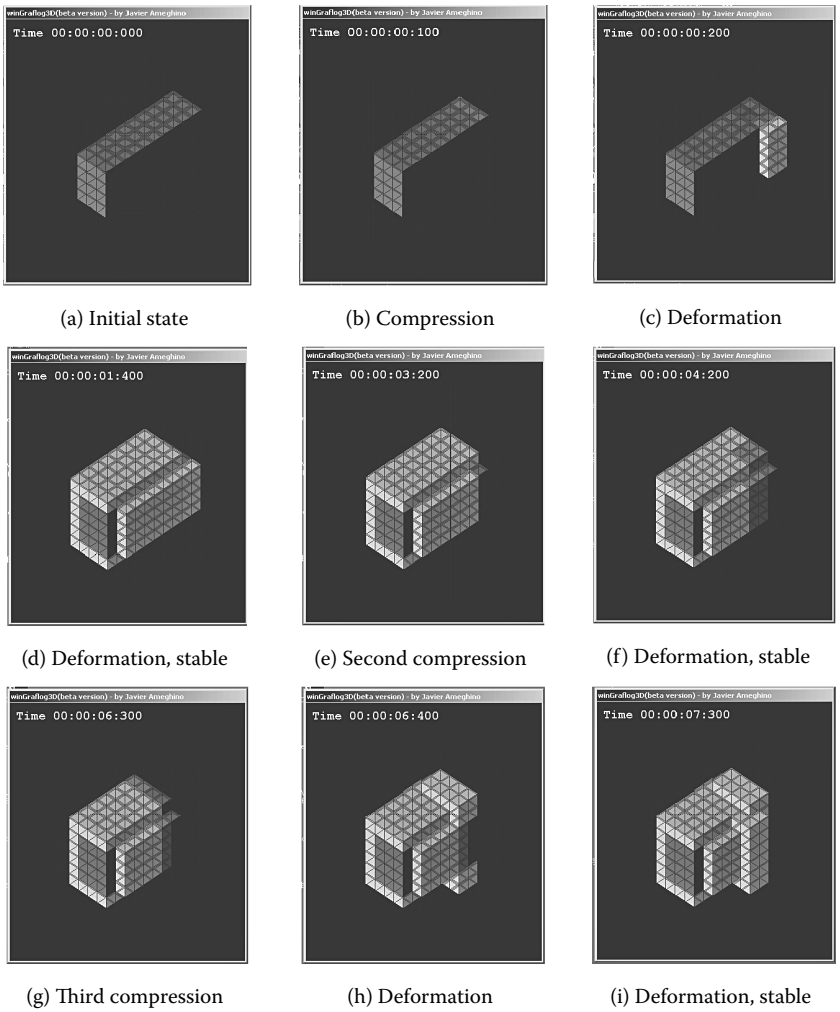
- Cells **gain** clay from neighbors on hyperplane (x,y,z,0) if (0,0,0,2) = 0, (0,0,0,0) is below the threshold and at least one neighbor is above the threshold.
- Cells **distribute** clay to neighbors on hyperplane (x,y,z,0) if (0,0,0,2) = 0, (0,0,0,0) is above the threshold and at least one neighbor is under the threshold.

Similar to deformation, compression only takes place on cube 0. Cube 2 controls when compression occurs: only when cell (x, y, z, 2) = 1. During compression, the clay in the cells right under the moving plate is transferred into the adjacent cells along the direction of pushing. The moving plate is represented by a set of cells with values of zero sitting on the top of the object. We assume the plate moves down along the z-axis as shown in Figure 12.39. For each cell (x, y, z, 0), if all neighboring cells (–1,–1,2,0), (–1,0,2,0), (–1,1,2,0), (0,–1,2,0), (0,0,2,0), (0,1,2,0), (1,–1,2,0), (1,0,2,0), and (1,1,2,0) are zero and at least one of the neighbor cells (–1,–1,1,0), (–1,0,1,0), (–1,1,1,0), (0,–1,1,0), (0,0,1,0), (0,1,1,0), (1,–1,1,0), (1,0,1,0), and (1,1,1,0) is greater than zero, the cell should gain all clay in its neighbor (0,0,1,0).

Cube 2 controls the deformation and compression stages. The value of each cell cube 2 switches between 0 (deformation) and 1 (compression). We assume that the transport delay of performing a compression step is 3,000 ms (30 times longer than a deformation). The transition rule for the control cube is as follows:

- S ← 1 if cell (0,0,0,1) = 0 and cell (0,0,0,0) = 0 and the cell itself is on cube 1.
- S ← 0 if cell (0,0,0,1) = 0 and cell (0,0,0,0) = 1 and the cell itself is on cube 1.
- S ← 1 if cell (0,0,0,0) = 0 and the cell itself is on cube 2.
- S ← 0 if cell (0,0,0,0) = 1 and the cell itself is on cube 2.

Figure 12.40 shows some of the results obtained. We studied each cell at different time steps (compression or deformation), as well as the total mass of the object. We found that the total mass (represented by cells in cube 0) was conserved for every transition. Figure 12.40 shows several steps during the transition process, which includes the initial state, first three compression steps, and some related deformation steps. Part (b) shows the immediate result after the first compression. Parts (c) and (d) show the object deformation. In part (d), a stable state is reached. Parts (e) to (i) show the repartition of clay after the second and third compression steps.



(a) Initial state (b) Compression (c) Deformation

(d) Deformation, stable (e) Second compression (f) Deformation, stable

(g) Third compression (h) Deformation (i) Deformation, stable

FIGURE 12.40 The deformation of the free-form object using Cell-DEVS.

12.7 SUMMARY

In this chapter, we have presented different models in chemistry and physics using Cell-DEVS. An advantage of this approach is that these real systems are often composed of continuous and discrete components interacting together. This dictates the need to integrate both models and simulate their global behavior. DEVS provides means for modeling discrete event systems, and Cell-DEVS enables modeling of different spatial systems. We described varied models on diffusion and reaction. We then introduced a model on snowflake formation and another one on binary solidification. We discussed how to model wave interference and a model of an FIA manifold. We also introduced a method for mapping problems modeled by partial differential equations and solved by finite differences, or FEM, into a Cell-DEVS specification. Finally, we introduced some models using Margolus neighborhoods: an HPP lattice gas model and a three-dimensional model of virtual clay.

Numerous other models can be found in the repository, including heat-spreading models (*./2dheat_diffusion.zip*), a model of power dissipation in circuits (*./powerdissipation.zip*), a model of the linear response of a truss system (*./Truss.zip*), a cellular model of a mass–spring–damper system (*./mks.zip*), and a quantum dot majority vote device (*./3inDelayedMajorityVoteGate.zip*).

REFERENCES

1. Wolfram, S. 2002. *A new kind of science.* Champaign, IL: Wolfram Media.
2. Wolfram, S. 1986. *Theory and applications of cellular automata,* vol. 1. Singapore: World Scientific.
3. Toffoli, T., and N. Margolus. 1987. *Cellular automata machines: A new environment for modeling.* Cambridge, MA: MIT Press.
4. Ding, W., X. Wu, L. Checiu, C. Lin, and G. Wainer. 2005. Definition of cell-DEVS models for complex diffusion systems. *Proceedings of Summer Computer Simulation Conference,* Philadelphia, PA.
5. Halsey, T. C. 2001. Diffusion-limited aggregation: A model for pattern formation. *Physics Today* 54.
6. Weimar, J. 2002. Three-dimensional cellular automata for reaction diffusion systems. *Fundamenta Informaticae* 52:275–282.
7. Checiu, L., and G. Wainer. 2005. Experimental results on the use of M/CD++. *Proceedings of Summer Computer Simulation Conference,* Philadelphia, PA.
8. Gaylord, R. J., and K. Nishidate. 1996. *Modeling nature.* New York: Springer–Verlag.
9. Reiter, C. 2005. A local cellular model for snow crystal growth. *Chaos, Solitons & Fractals* 23:1111–1119.
10. Kremeyer, K. 1997. Experimental and computational investigations of binary solidification. PhD thesis, Department of Physics, University of Arizona, Tucson, AZ.
11. Ameghino, J., and G. Wainer. 2004. Application of the cell-DEVS formalism for modeling cell spaces. *Proceedings of Artificial Intelligence, Simulation and Planning 2004, LNCS 3397,* Jeju Island, Korea.
12. Nutaro, J. 2006. A discrete event method for wave simulation. *ACM Transactions Model Computer Simulation* 16:174–195.
13. Andrade, F. J., F. A. Iñón, M. B. Tudino, and O. E. Troccoli. 1999. Integrated conductimetric detection: Mass distribution in a dynamic sample zone inside a flow injection manifold. *Analytica Chimica Acta* 379:99–106.
14. Troccoli, A., J. Ameghino, F. Iñón, and G. Wainer. 2002. A flow injection model using cell-DEVS. *Proceedings of 35th IEEE/SCS Annual Simulation Symposium,* San Diego, CA.
15. Wainer, G., and B. P. Zeigler. 2000. Experimental results of timed cell-DEVS quantization, AI and simulation. *AIS 2000*, Tucson, AZ, 203–208.
16. Zeigler, B. P. 1998. DEVS theory of quantization. Technical report, DARPA Contract N6133997K-0007, ECE Dept., the University of Arizona, Tucson, AZ.
17. Chandrupatla, T., and A. Belegundu. 1997. *Introduction to finite elements in engineering.* Upper Saddle River, NJ: Prentice Hall.
18. Brauer, J. 1988. *What every engineer should know about finite element analysis.* New York: Marcel Dekker, Inc.
19. Saadawi, H., and G. Wainer. 2003. Improving the finite element method using cell-DEVS. *Proceedings of 2003 SCS Summer Computer Simulation Conference,* Montreal, QC, Canada.
20. Saadawi, H., and G. Wainer. 2007. Defining models of complex 2D physical systems using cell-DEVS. *Simulation Modeling Practice and Theory* 15:1268–1291.
21. Saadawi, H., and G. Wainer. 2004. Modeling complex physical systems using 2D finite element cell-DEVS. *Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04),* Arlington, VA.
22. Hardy, J., O. de Pazzis, and Y. Pomeau. 1976. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Physical Review A* 13:1949–1961.
23. Rothman, D. A., and S. Zaleski. 2004. *Lattice-gas cellular automata: Simple models of complex hydrodynamics* (*Collection Alea-Saclay: Monographs and texts in statistical physics*). Cambridge: Cambridge University Press.
24. Kameyama, K. 1997. Virtual clay modeling system. *Proceedings of VRST '97: Proceedings of the ACM Symposium on Virtual Reality Software and Technology,* Lausanne, Switzerland, 197–200.
25. Arata, H., Y. Takai, N. K. Takai, and T. Yamamoto. 1999. Free-form shape modeling by 3D cellular automata. *Proceedings of SMI. International Conference on Shape Modeling and Applications,* Aizu, Japan, 242–247.
26. Druon, S., A. Crosnier, and L. Brigandat. 2003. Efficient cellular automata for 2D/3D free-form modeling. *Proceedings of WSCG. 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2003,* Plzen-Bory, Czech Republic.
27. Wu, P., X. Wu, and G. A. Wainer. 2004. Applying cell-DEVS in 3D free-form shape modeling. *Proceedings of Cellular Automata, 6th International Conference on Cellular Automata. ACRI 2004; Lecture Notes in Computer Science,* Amsterdam, 81–90.

13 Models of Artificial Systems, Networking, and Communications

13.1 INTRODUCTION

Discrete-event simulation methodologies were originally created to model artificial systems like the ones we will introduce in this chapter. As shown in Chapter 2, many existing artificial systems can be modeled as having I/O trajectories that are piecewise constant. Here, we will show a number of examples in this area. We first show a load-balancing system model. Then we introduce Alpha-1, a simulated digital computer based on the architecture of the SPARC processor. We then define models on robot path planning and a digital controller for a time-varying plant. We include a specialized library for modeling and simulation of networking and communications, including data generators, internetworking devices, and wireless ad hoc networks.

13.2 A LOAD-BALANCING SYSTEM

Our first example presents a simplified model of a database system in which the system distributes the workload among three servers accessing a common database. The load balancer receives jobs from clients and dispatches them to the servers for processing using a round-robin selection algorithm; they are dispatched using a fixed time for each incoming job. Each server processes a job from the balancer for a period (using an exponential distribution) and then sends the job to the database server for processing. The database takes a fixed time for processing each job and returns the response to the originating server. If the balancer, the servers, or the database server are busy, the job is queued. A job will also be queued if the server is waiting for a response from the database server.

The model in Figure 13.1 shows how to reuse predefined models in CD++. The *generator* model is the one introduced in Chapter 4, and the *server* model is based on the *queue* model presented in Chapter 4. The *dbserver* is also a modified version that queues job requests and, according to the input source, transmits the output to the corresponding output port. Figure 13.2 shows the implementation of the *balancer* model in CD++, found in *./loadbalancer.zip.*

The model in Figure 13.2 uses one input and three output ports. Initially, we clear the queue of jobs, and we obtain the *dispatchTime* for the component (which must be included by the user in the coupled model file definition) using the *getParameter* method. When a new job is received in the external transition function, we obtain its job ID and add it to the job queue. We then check to see if this is the only job in the queue. In such a case, we dispatch it immediately, taking it from the front of the queue and scheduling an internal event after the dispatch time. When that time is consumed, we first generate an output representing the chosen server by taking the ID of the first job in the queue and transmitting it through the corresponding output port (1–3). The internal transition checks whether more jobs are waiting and, in such a case, dispatches the new job. Otherwise, the model passivates, waiting for new jobs to arrive.

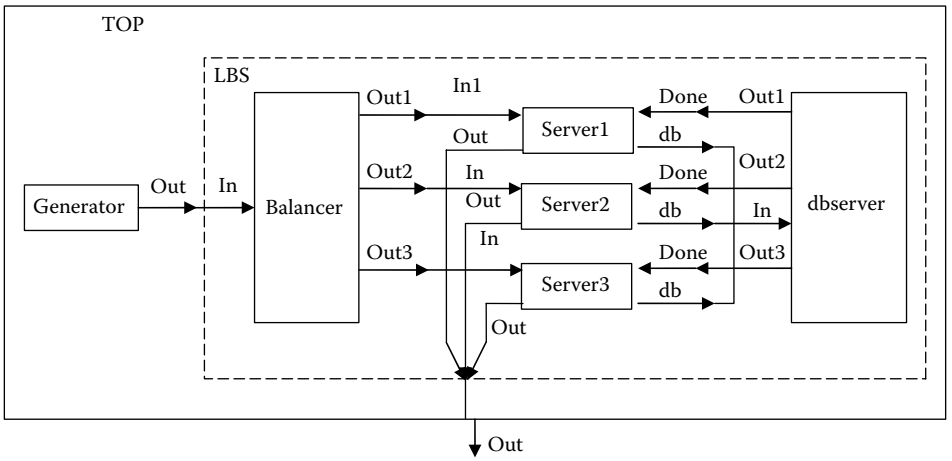


FIGURE 13.1 Structure of the load-balancing system.

Exercise 13.1

Change the dispatch policy. Use a priority queue (ports with higher numbers have higher priority) and a Last In, First Out policy. Compare the simulation results obtained in the three cases.

Exercise 13.2

Add a new input port carrying information about priority for each job. Transmit the job ID with a priority ID. Modify the models to allow this modification and run the simulation again, comparing with the results obtained in Exercise 13.1.

After each atomic model has been created, we define a coupled model using the description in Figure 13.1. The model can be formally defined as

$$\text{LBS} = < X, Y, \{\text{balancer, server1, server2, server3, dbserver}\}, EIC, EOC, IC, select > \quad (13.1)$$

where

X = { in }
Y = { out }
EIC = { (in, balancer.in) }
EOC = { (server1.out, LBS.out), (server2.out, LBS.out), (server3.out, LBS.out) }
IC = { (balancer.out1, server1.in), (balancer.out2, server2.in), (balancer.out3, server3.in), (server1.db, dbserver.in), (server2.db, dbserver.in), (server3.db, dbserver.in), (dbserver.out1, server1.done), (dbserver.out2, server2.done), (dbserver.out3, server3.done) }
select: ({balancer, server1, dbserver}) = dbserver; ({balancer, server2, dbserver}) = dbserver ({balancer, server3, dbserver}) = dbserver; ({balancer, server1}) = server1 ({balancer, server2}) = server2; ({balancer, server3}) = server3

Figure 13.3 shows a graphical representation for the top-level coupled model using CD++Modeler. When this model is executed, the results in Figure 13.4 are obtained. When job 1 is received at time 1:000, it is immediately dispatched. After four time units, it is transmitted to Server2, which processes it and transmits it to the database server. Five time units after that, the job is sent back to Server2, and it finishes. The second job arrives at 20:000, and it is transmitted to Server3. One time unit after that, a new job arrives (job 3). The balancer dispatches job 2 after receiving and queuing

```
Balancer::Balancer( const string &name ): Atomic( name ),
   in( addInputPort( "in" ) ), out1( addOutputPort( "out1" ) ), out2( addOutputPort(
"out2" ) ),
   out3( addOutputPort( "out3" ) ), dispatchTime( 0, 0, 0, 010 )      {

   string time( MainSimulator::Instance().getParameter( description(), "dispatch" ) ) ;
   if( time != "" ) dispatchTime = time ;
}

Model &Balancer::initFunction() {
   job_queue.erase( job_queue.begin(), job_queue.end() ) ;
   return *this ;
}

Model &Balancer::externalFunction( const ExternalMessage &msg ) {
   int new_pid = (int) msg.value();
   job_queue.push_back( msg.value() ) ;

   if( job_queue.size() == 1 ) {      // if this is the only job in the queue, start
dispatching
        int pid = (int) job_queue.front();
        holdIn( active, dispatchTime );
   }
}

Model &Balancer::internalFunction( const InternalMessage &msg ) {
    // if there is any job in the queue, start dispatching
    if  (job_queue.size() > 0 ) {
        int pid = (int) job_queue.front();
        holdIn( active, dispatchTime );
    }
    else
        passivate();
    return *this ;
}

Model &Balancer::outputFunction( const InternalMessage &msg )   {
   int pid = (int) job_queue.front();

   if (pid % 3 == 0)           // round-robin dispatching
        sendOutput( msg.time(), out1, pid ) ;
   else if (pid % 3 == 1)
              sendOutput( msg.time(), out2, pid ) ;
         else
             sendOutput( msg.time(), out3, pid ) ;

        job_queue.pop_front() ;
```

FIGURE 13.2 Load balancer atomic model.

job 3 (at 24:000). Running the load balancer simulation with different arguments produces the results shown in the following table:

Mean (s)	No. Jobs Generated	No. Jobs Finished	Throughput (job/second)
2	9031	1841	0.102
5	3645	1824	0.101
10	1870	1755	0.098
15	1157	1157	0.064
20	811	811	0.045

The table shows different results for various values of the mean interarrival time for the generator (which is exponentially distributed; that is, the lower the mean is, the faster the generator generates jobs). In this test, the dispatching time for the balancer is fixed at 4 s, and the processing time for the

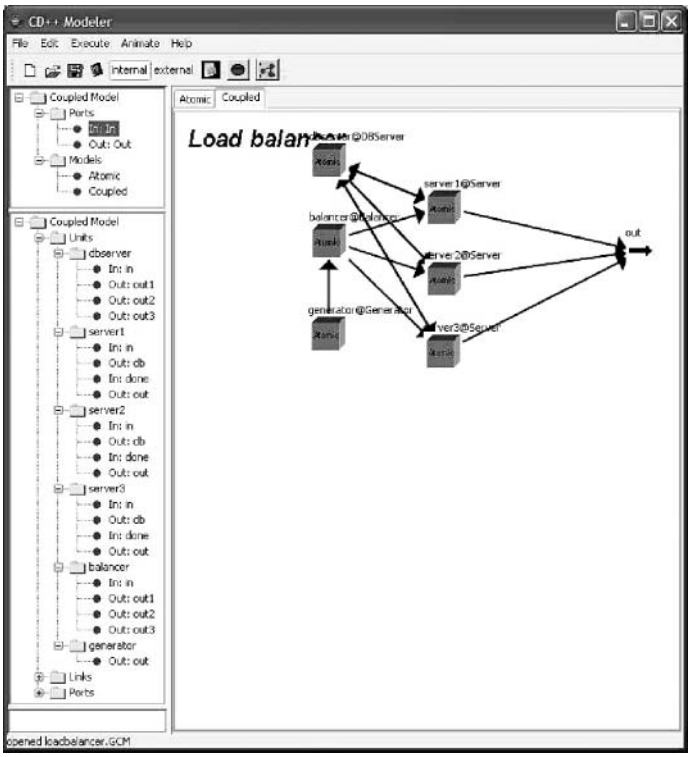


FIGURE 13.3 Load-balancing system top model.

```
00:00:01:000 / in /      1.00000
00:00:20:000 / in /      2.00000
00:00:21:000 / in /      3.00000
00:00:01:000   Balancer receives job# 1
00:00:01:000   Balancer starts dispatching job# 1
00:00:05:000   Balancer sends job# 1  to server 2
00:00:05:000   Server 2 receives job# 1
00:00:05:000   Server 2 starts processing job# 1
00:00:05:820          Server 2 sends job# 1 to database server.
00:00:05:820   DBServer receives a job from server 2
00:00:05:820   DBServer starts processing job from server 2
00:00:10:820   DBServer sends job back to server 2
00:00:10:820   Number of jobs done = 1  *******
00:00:10:820   Server 2 finishes job# 1
00:00:20:000   Balancer receives job# 2
00:00:20:000   Balancer starts dispatching job# 2
00:00:21:000   Balancer receives job# 3
00:00:24:000   Balancer sends job# 2  to server 3
00:00:24:000   Balancer starts dispatching job# 3
00:00:24:000   Server 3 receives job# 2
00:00:24:000   Server 3 starts processing job# 2
00:00:25:000   Balancer receives job# 4
```

FIGURE 13.4 Load-balancing system simulation results.

database server is fixed at 5 s. The processing time for each server is exponentially distributed with a mean of 20 s, and the simulation time is 5 h. The testing results indicate that the system throughput increases as the job arrival rates increase (for the means of 20, 15, and 10 s). However, as the rate increases further (for the means of 5 and 2 s), the system throughput remains almost stable, because

the servers become performance bottlenecks when the rate reaches a certain value. Such results are consistent with that of the real load-balancing system.

Exercise 13.3

Repeat the global test for the policies introduced in Exercises 13.1 and 13.2. Compare the simulation results obtained in the three cases.

13.3 THE ALPHA-1 SIMULATED PROCESSOR

This model was originally created as an educational tool to support the theoretical studies of computer architecture and organization. Although most of the bibliographies for these courses focus on the behavior of the logical subsystems of a computer (e.g., references 1–3), there is a lack of practice opportunities due to the complexity of the subsystems and their interactions. In order to support the learning experience, we defined a simulated computer with educational purposes [4–6]. To show the feasibility of the approach, the project was fully developed by a team of more than 20 undergraduate students as a part of their coursework. The model's architecture is mainly based in the specification of the *integer unit* (IU) of the SPARC processor by Sun Microsystems, with a few simplifications in the instruction set and the memory management unit (Figure 13.5).

The simulated memory is flat, and multiprogramming is not supported. Memory addressing uses two registers: *base* and *size,* which define the memory space for one program. The REGFILE component contains 520 general-use (integer) registers organized as a ring. They are divided into

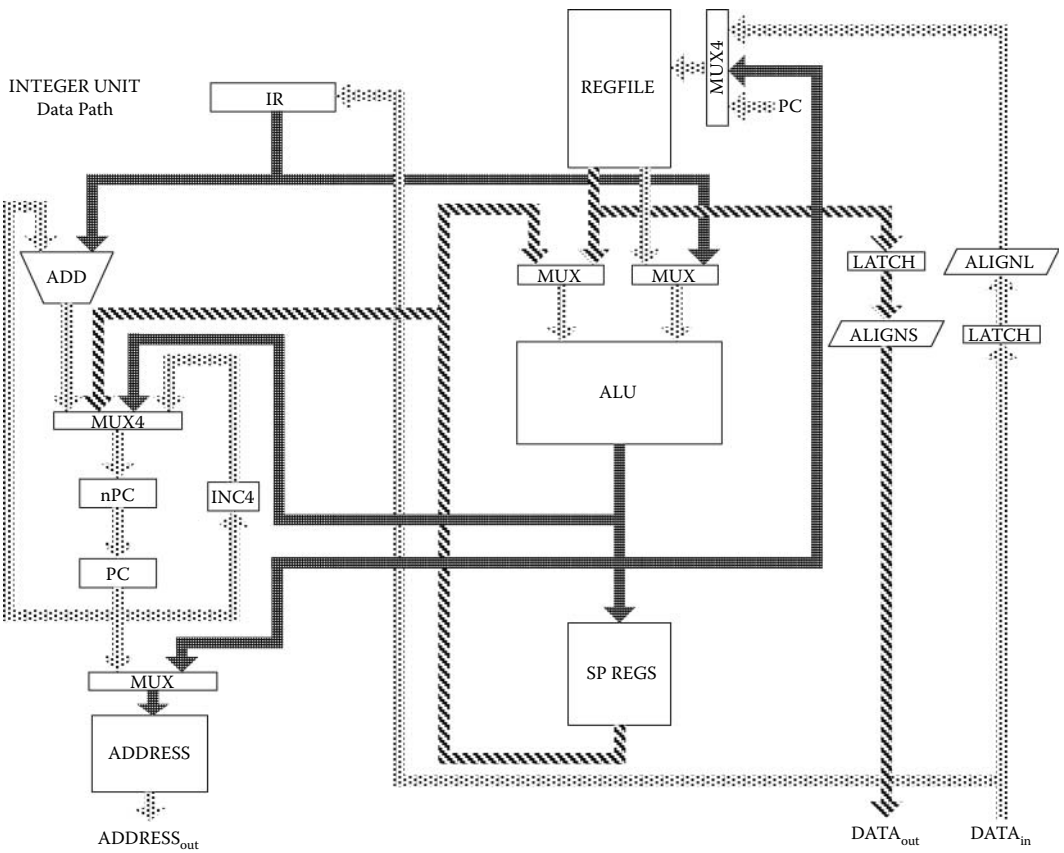


FIGURE 13.5 Sketch of the simulated integer unit.

three classes: 8 global (shared by all the procedures) and 512 organized in 64 windows of 24 input, output, and local registers for each procedure. When a procedure is started, 16 new registers are reserved (8 local and 8 output), and the 8 output records of the calling procedure are used as input registers, as seen in Figure 13.6.

A specialized 5-bit register, called the *CWP* (circular window pointer), marks the active window into the register array. The 32-bit *WIM* register (window invalid mask) is used to avoid the superposition with a register window already used by a procedure. When the *CWP* is decreased,

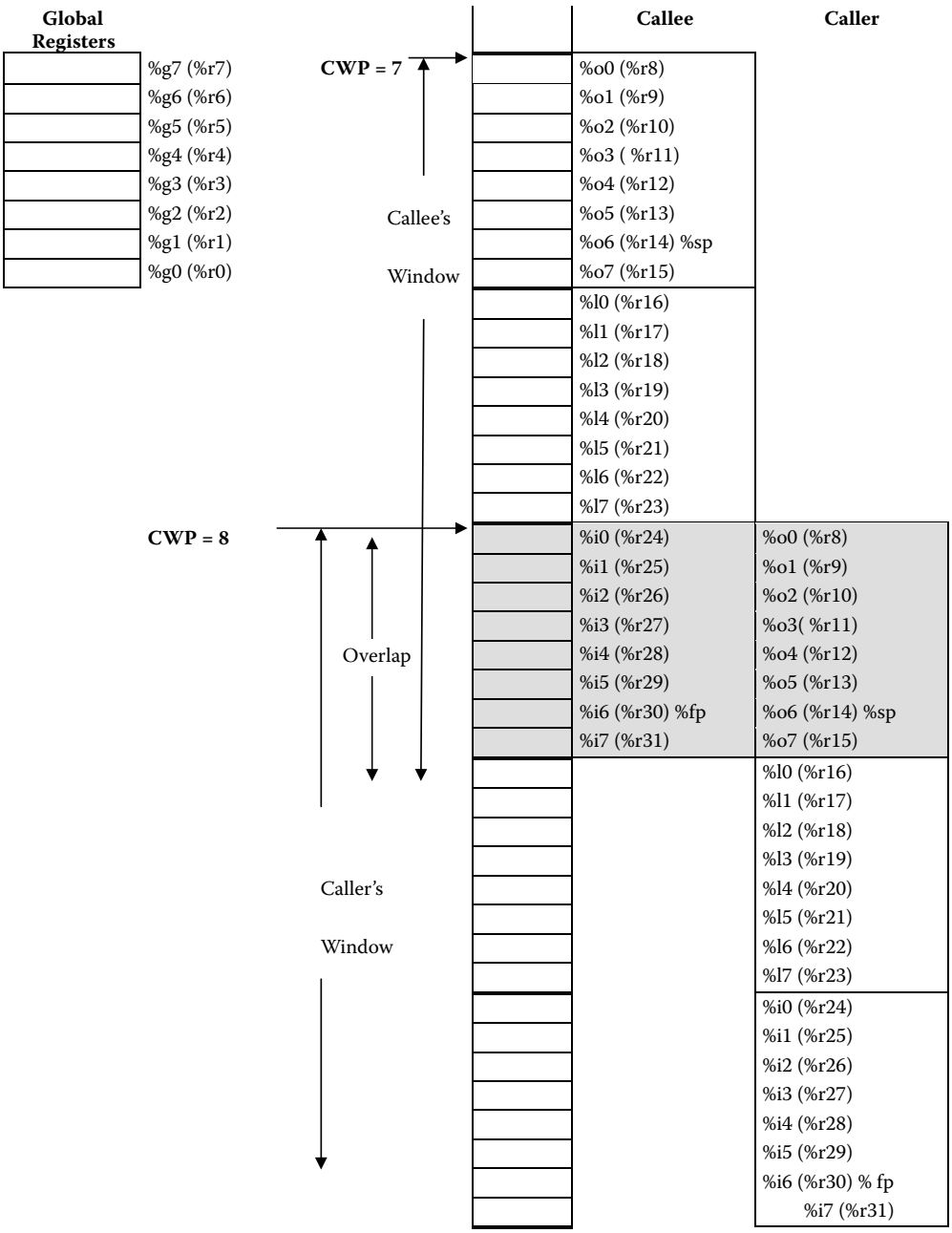


FIGURE 13.6 Register windows.

the hardware verifies whether the *WIM* of the new window is on. In this case, an interrupt is raised. Usually, the *WIM* has all the bits off, except for a bit in 1 that signals the oldest window.

Other registers include:

- **Y**: used by the product and division operations;
- **TBR** (trap base register): points to the memory address where the trap routine starts;
- **BASE** and **SIZE**: BASE points to the lower address that the program can access, while SIZE stores the length;
- **PSR** (processor status register): stores the present status for the program;
- **PC** (program counter): contains the address of the next instruction; and
- **nPC** (next program counter): stores the address for the following PC (nPC = PC + 4).

Each of the components of the integer unit architecture was defined as a DEVS model, and the complete IU was modeled as a coupled model. Two levels of abstraction were defined: the *functional* behavior using the atomic model transition functions, and the *digital logic* level (in some of the submodels) by developing the basic Boolean gates as atomic models and coupling them using digital logic concepts. For instance, the *WIMCHECK* model in Figure 13.7 is in charge of checking window overflow (underflow) on save (restore) operations in the register window.

WIMCHECK (presented in Wainer et al. [6] and found in *./alfa1.zip*) uses a five-line input decoder and a latch to keep the last result (as the present state should not be transmitted if it has not changed). We store the values received through the WIM and CWP ports (*WIM0-WIM31*; *CWP0-CWP4*).

The external transition function is in charge of setting/clearing the corresponding bit of the WIM and CWP registers according to the messages received through the model's input ports (Figure 13.8). The timing information for the circuit (i.e., the time needed to save a stable value on the registers) is used to schedule the next internal transition function. When this time is consumed, the output function calls the *wimResult* method, which returns the value of the *CWP*th position of the WIM register. *wimResult* uses a decoder over CWP, returning a 32-bit string with all the bits in zero, except for the *CWP*th bit. These bits are ANDed with the WIM register, obtaining all the bits in zero, if the *CWP*th was zero, or if the *CWP*th bit was one. The *wimResult* will be obtained by making an OR of these bits, which is subsequently used to compare with the previous output value of the model (stored in the *dlLastRES* latch). If they differ, the new result is transmitted through the RES port and the clock signal of the latch is set.

Figure 13.9 shows the results of two different tests. The first column shows a test in which we set the bits 0, 2, and 4 of the WIM. We then query bit 1 by setting the first bit of the CWP (bit 4). After the stabilization time, we check bit 2, and we obtain a value of 1 on port *Res*. On our second test, we

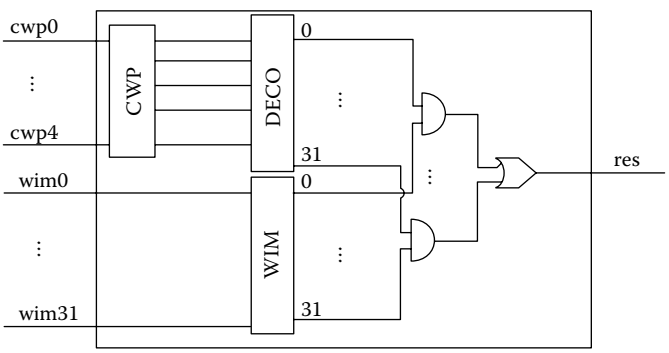


FIGURE 13.7 WIMCHECK model: basic sketch.

```
Model &WIMCHECK::externalFunction( const ExternalMessage &msg ) {

  int iPortNumber = getNumberFromString(msg.port().name(), 4);      //Get port number

  if( isWimPort(msg.port() )) //If port WIMxx
       setBitInReg ( rWIM, iPortNumber, (msg.value()==1 ? '1': '0'));
  else          //If port CWPxx
       setBitInReg ( rCWP, iPortNumber, (msg.value()==1 ? '1': '0'));

  m_tStabilizationTime = time ;

  holdIn(active, m_tStabilizationTime );
}

Model &WIMCHECK::internalFunction( const InternalMessage & ) {
   this -> passivate();
}

Model &WIMCHECK::outputFunction( const InternalMessage &msg ) {

   char cCurrent = wimResult();

   if (dlLastRES.output() != cCurrent) {
        dlLastRES.activate(cCurrent, CLOCK);
        sendOutput( msg.time(), RES,  (cCurrent == '1' ? 1 : 0));
   }
}
```

FIGURE 13.8 WIMCHECK model: transition functions.

```
INPUTS
00:00:00:001 / wim0 / 1.000
00:00:00:003 / wim2 / 1.000
00:00:00:005 / wim4 / 1.000
00:00:00:006 / cwp4 / 1.000
00:02:00:000 / cwp3 / 1.000
00:02:00:000 / cwp4 / 0.000

OUTPUTS
00:02:00:040 res 1
```

```
INPUTS
00:00:00:010 / cwp0 / 0.000
00:01:10:000 / cwp0 / 1.000
00:01:10:001 / cwp1 / 1.000
00:01:10:002 / cwp2 / 1.000
00:01:10:003 / cwp3 / 1.000
00:01:10:004 / cwp4 / 1.000
00:02:00:001 / wim0 / 1.000
00:02:10:000 / cwp0 / 0.000
00:02:10:001 / cwp1 / 0.000
00:02:10:002 / cwp2 / 0.000
00:02:10:003 / cwp3 / 0.000
00:02:10:004 / cwp4 / 0.000
00:03:00:001 / wim0 / 0.000
OUTPUTS
00:02:10:044 res 1
00:03:00:041 res 0
```

FIGURE 13.9 WIMCHECK simulation results.

check bits 0 and 31 without modifying the WIM register values. We then set bit 0 of the WIM and query the register. Finally, we clear the bit and repeat the query. As a result, we first obtain 1 and then 0, showing that when we change the state of a bit, the circuit changes accordingly.

Every circuit in the integer unit was modeled and tested using a similar approach. We then redefined a number of the models at a lower level of abstraction (using digital logic), providing a multiresolution model. These models were built as a set of basic components representing the Boolean gates AND, OR, NOT, and XOR, as shown in Figure 13.10(a). These gates were incorporated as coupled models representing the structure of the circuit. For instance, Figure 13.10(b) shows a 3-bit comparator.

Figure 13.11 shows the definition of the comparator coupled model (CMP), which is a part of the Address Unit [6]. The Address Unit uses an adder (to update the PC) and CMP to check whether

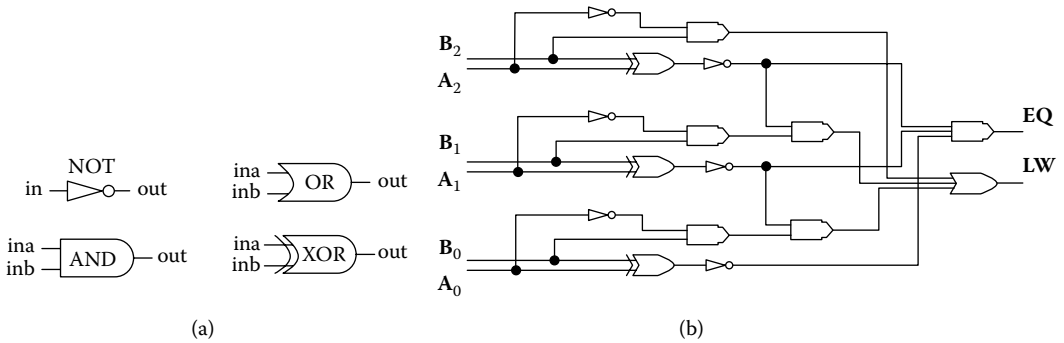


FIGURE 13.10 Modeling a comparator using Digital Logic.

```
[top]
components : NOT_n_1@NOT NOT_n_2@NOT XOR_n@XOR AND_n_1@AND AND_n_2@AND
in : OPAn OPBn
out : LW EQ

Link : OPAn in@NOT_n_1
Link : OPBn ina@XOR_n
Link : OPAn inb@XOR_n
Link : OPBn inb@AND_n_1
Link : out@NOT_n_1 ina@AND_n_1
Link : out@XOR_n in@NOT_n_2
Link : out@AND_n_2 EQ
Link : out@NOT_n_2 LW
```

FIGURE 13.11 CMP coupled model. (From Wainer, G. et al. 2001. *ACM Journal on Educational Resources in Computing* 1:111–151.)

the address is outside the limits. The CMP was built as a 32-bit extension of the 3-bit comparator presented in Figure 13.10. The model receives two inputs (through latches *OPA* and *OPB*), and it returns the signal *EQ* if both values are equal or *LW* if A is lower than B.

Exercise 13.4

Design and implement a main memory unit (MMU) for Alpha-1. This MMU must implement different memory access techniques, including: (1) segmentation, (2) paging, (3) segmented paging, and (4) paged segmentation.

Exercise 13.5

Design and implement a translation lookaside buffer model, which should mimic the translation of virtual addresses into real addresses. (This exercise can be done independently of the rest of the simulated computer.)

The simulated computer can be used to run SPARC executable code. To do so, the user must first write an assembly language program. Figure 13.12 shows one of the test examples found in *./alfa1-test.zip,* which adds two registers with carry (*addcc*) and moves the results into memory. The contents of the register and the register numbers used are chosen at random.

We need to create a memory map from the assembler source by assembling the source code. Then we need to save the object file into the `memory.map` file, which contains the memory image. We should also configure the memory size of the machine (in this case, 256 bytes) and choose where the results are going to be saved (`memory.dmp`). This file will have the size in bytes that we had used at `[mem]` definitions (Figure 13.13).

```
            set 7541504,  %r30     !7541504 to register 30
            set 1862488, %r14      !1862488 to register 14

            addcc %r14, %r30, %r11  !Add, result to register 11
            st   %r11, [dest]      !Save the result in memory

            unimp

               .align 4
value:      .ascii "VALUE:"
dest:        .word  FFFFFFFF  !result of the Test 1
```

FIGURE 13.12 Simple example to run on the simulated computer.

```
…
[mem]
preparation: 0:0:0:50
memsize: 256
memfile: memory.map
dumpfile: memory.dmp
…
```

FIGURE 13.13 Editing the memory source.

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000:81 03 01 0d 0a 00 00 20 00 00 00 20 00 00 00 00    □...... ... ....
00000010:00 00 00 00 90 00 00 00 20 00 00 00 00 00 00 00    ....□... .......
00000020:00 11 00 17 08 90 12 20 5e 33 00 06 d0 73 13 00    .....□. ^3..Ðs..
00000030:58 1c 6b 58 08 d4 20 20 42 00 00 00 00 56 41 4c    X.kX.Ô  B....VAL
00000040:55 45 77 62 e2 00 00 00 00 00 00 00 34 00 00 00    UEwbâ.......4...
00000050:0d 0a 00 00 00 22 00 00 00 42 00 00 00 82 00 00    .....".../B.../..
00000060:00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00    ................
00000070:00 00 00 00 00 00 ff ff ff ff 00 00 00 00 00       ......ÿÿÿÿ.....
```

FIGURE 13.14 An initial memory map.

Figure 13.14 shows the initial memory map for the execution of this program. We can see that the first instruction is on address `2B` (`73 13 00h = 7541504d`, which is the data used by the instruction as seen in Figure 13.12). The map can be used to run the simulation, and the memory contents will change accordingly during execution. The result of the simulation is stored in the file `memory.dmp` with the state from the memory when concluding the execution.

Detailed examples of execution can be found in Daicz, Troccoli, and Wainer [4] and Wainer et al. [6]. For instance, when we check the contents of the `memory.dmp` file for this example, we will find `VALUE:int32,9403992`, which is the right result for the addition of the two values in the program.

If we study the simulation log, we can see the detailed execution behavior of the simulated computer, including all the subcomponents activated to carry out the execution of the program, the values of the registers, and the memory changes.

Exercise 13.6

Execute other existing tests and analyze the simulation results obtained.

13.4 ROBOT PATH PLANNING

Path planning for robotic applications is used to determine how a robot can reach a destination, avoiding obstacles on the way. In this section, we will present different mechanisms to do this, including a variety of sample models. Most complex cases include multiple robots in cooperative environments (in which mobile agents interact, trying to achieve a common goal).

In order to follow the movement of robots in an area, we need a spatial planner, which must find a path free of obstacles to follow a predefined trajectory. Depending on the obstacle information available, this planning can be static or dynamic. In general, this consists of two phases:

- Route planning: A route is defined as a sequence of subgoals that the robots must reach before the final goal.
- Path generation: Once the plan has been created, different heuristics (for instance, the shortest path) could be used to reach the predefined goal.

13.4.1 Fixed-Route Paths

The model found in *./robots.zip* presents the movement of robots in an industrial plant using fixed paths (i.e., tags are distributed throughout the plant, and the robots have detectors for finding their path using these tags). Robots carry a load from the source point, where it is produced, to a destination point, where it is consumed. Routes are one way; once the load is delivered, robots are taken off the floor back to their starting point. The robots move at different speeds, and there may be more than one robot on each route [7].

Figure 13.15 shows a sample floor plant represented by a 20 × 20 Cell-DEVS. This cellular model is linked to four different DEVS models, each devoted to generating a load at the source points (12,19), (0,10), (9,0), and (19,6), as seen in Figure 13.16. The coupled model uses five components (*Floor*—a Cell-DEVS—and *Source1–Source4,* load generator models built as DEVS). Then we link all the models (generators output ports are connected to *floor* input ports) and we define the Cell-DEVS *floor* coupled model parameters. In this case, the input ports *in1* to *in4* will be coupled to the cell space: events arriving on port *in1* should be sent to the *in* port of the cell at position (12,19).

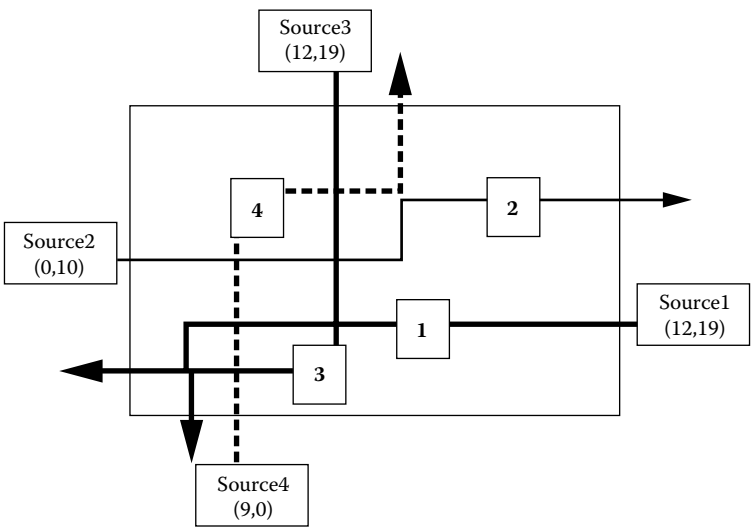


FIGURE 13.15 Floor plan with robot routes.

```
[top]
components : Floor Source1@Generator Source2@Generator Source3@Generator
Source4@Generator
link : out@Source1 in1@Floor
link : out@Source2 in2@Floor
link : out@Source3 in3@Floor
link : out@Source4 in4@Floor

[Floor]
type : cell            dim : (20,20)
delay : inertial       border : nowrapped
neighbors : (-1,0) (0,-1) (0,0) (0,1) (1,0)
in : in1 in2 in3 in4
link : in1 in@Floor(12,19)
link : in2 in@Floor(0,10)
link : in3 in@Floor(9,0)
link : in4 in@Floor(19,6)
localtransition : RobotsMov

[RobotsMov]
...
% ------ Robot 2 -----------------------------------
rule : 30 1000 {(0,1)=3 and (0,0)=0 and cellpos(1)!=4}
rule : 31 1000 {(0,1)=3 and (0,0)=0 and cellpos(1)=4}
rule : 0 0 {(0,-1)=30 and (0,0)=3}
rule : 0 0 {(0,-1)=31 and (0,0)=3}
rule : 4 0 {(0,0)=31}
rule : 3 0 {(0,0)=30}

rule : 40 2000 {(-1,0)=4 and (0,0)=0 and cellpos(0)!=17}
rule : 41 2000 {(-1,0)=4 and (0,0)=0 and cellpos(0)=17}
rule : 0 0 {(1,0)=40 and (0,0)=4}
rule : 0 0 {(1,0)=41 and (0,0)=4}
rule : 4 0 { (0,0)=40 }
rule : 3 0 { (0,0)=41 }

% ------ Robot 3 -------------------------------
...
```

FIGURE 13.16 Model definition for robot routes.

We also show an excerpt of the Cell-DEVS model. We use a value of zero for an empty cell. A cell containing a route 2 robot uses values 3, 30, or 31 if the robot is moving horizontally, and 4, 40, or 41 if the robot is moving vertically (the same applies for cells containing robots in other routes; valid values for a cell containing a route 1 robot are 1, 10, 11; 2, 20, 21; for cells containing a route 3 robot, they are 5, 50, 51; 6, 60, 61; and for cells containing a route 4 robot, they are 7, 70, 71; 8, 80, 81). The *cellpos()* function is used to see if the robot is on the predefined path.

Robot movement is done in three steps, as seen in Figure 13.17. For example, a route 1 robot at the source is indicated by a 1 in cell (12,19), which indicates the robot is ready to move horizontally. After a delay of 1000 ms, the next cell on the route will receive a neighbor change event indicating that cell (12,19) has just changed to 1, producing a change to 10 (or 11) and indicating the cell is

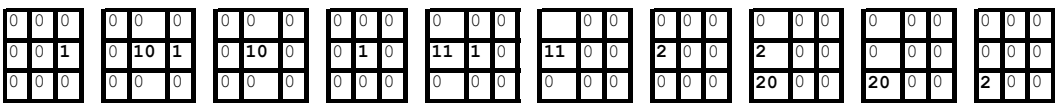


FIGURE 13.17 Route 1 movements.

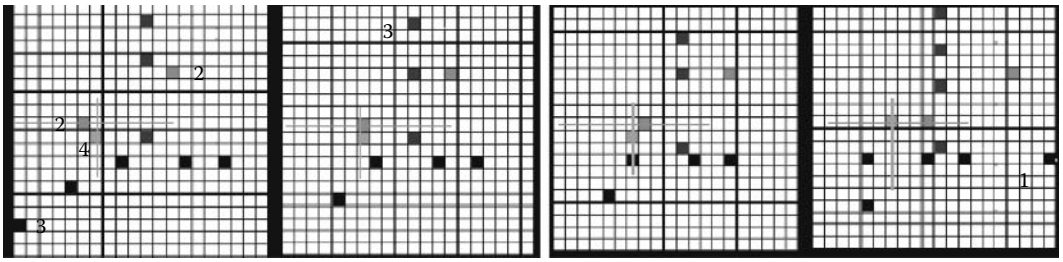


FIGURE 13.18 Executing the robot model (showing two robots reaching an intersection point).

prepared to receive the robot (10 is used if the robot continues horizontally and 11 if the robot must turn). Then the origin cell changes to 0 (the robot is no longer present). Finally, the value 10 (or 11) will change to 1 or 2, respectively (1 indicates the presence of a robot that is about to move horizontally and 2 a robot that is about to move vertically). Collisions are avoided by allowing the first step to take place only if the destination cell is empty.

Figure 13.18 shows a simulation scenario for this example. The robots follow the paths defined in Figure 13.15. New robots arrive at the floor through route 3 (from the top of the graph), and a robot in path 2 can be seen in the NE of the figure. Likewise, we can follow those arriving at route 1 from the E.

Robots run at different speeds (according with their delays), and collision is avoided between a robot in route 2 and another in route 4 (marked in the middle of the figure). The robot in route 2 advances, while the one in route 4 waits until there is a safe distance to continue.

Exercise 13.7

Change the rate of generation of robots and repeat the tests.

Exercise 13.8

Modify the path used and the robot speed in the Cell-DEVS model definition.

13.4.2 Route Planning Models

We present a Cell-DEVS model for route planning, which, based on the obstacles, finds the different paths available and creates a Voronoi diagram (Figure 13.19). Voronoi diagrams use the idea of proximity to a finite set of points in the plane $P = \{p_1 \ldots p_n\}$ ($n \geq 2$). The diagram associates every point p_j to its closest points p_i ($i \neq j$). The resulting sets define a tessellation of the plane into regions (exhaustive because every point belongs to a set and they are mutually exclusive), and points equidistant to two elements in P define the *border* of the regions. Voronoi diagrams can be used to describe the paths surrounding the obstacles for a robot of a given size and to indicate the distance to them. These indicators allow a robot to determine whether the path is feasible to pass through [8,9].

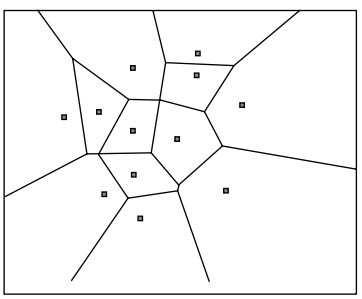


FIGURE 13.19 Voronoi diagram.

The path-planning model presented here is based on Behring et al. [8], where a cellular model was used to process a top-down bitmap including a robot of arbitrary shape and a number of obstacles. Because cellular models only use local rules, any proposed algorithm can be applied to objects of arbitrary size or shape without computing distances or intersections or explicitly modeling objects.

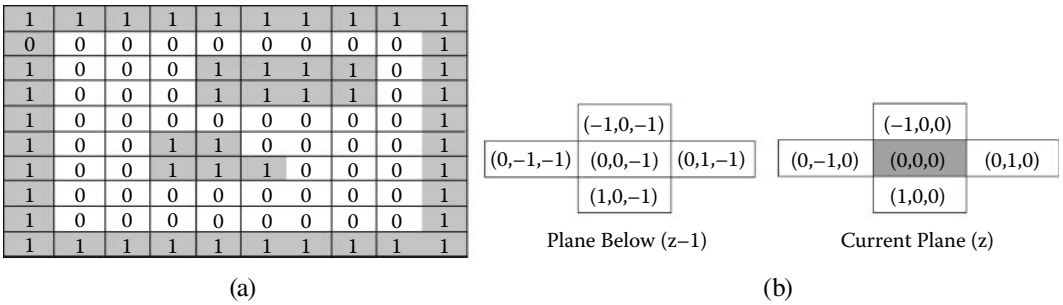


FIGURE 13.20 (a) Input bitmap; (b) three-dimensional neighborhood.

```
[Path-Finding]
dim : (10, 10, 4)  delay : transport localtransition : nothing-rule
neighbors:(-1,0,0)(0,-1,0)(0,0,0)(0,1,0)(1,0,0)(0,-1,-1) ... (0,1,-1)
zones : bound-rule { (0,0,1)..(9,9,1) } plane2-rule { (0,0,2)..(9,9,2) }
        plane3-rule { (0,0,3)..(9,9,3) }
[nothing-rule]
rule: { (0,0,0) } 10 { t }
[bound-rule]
rule: 1 10 { (0,0,-1)=1 and (0,-1,-1)=1 and (-1,0,-1)=1 and (0,1,-1)=1 and (1,0,-1)=1 }
...
rule: 12 10 { (0,0,-1)=1 and (0,-1,-1)=1 and (-1,0,-1)=0 and (0,1,-1)=0 and (1,0,-1)=1 }
[plane2-rule]
rule: {(0,0,-1)+0.1} 10 { (0,0,-1) >4 and (0,0,-1)<13 }
...
rule: {(0,1,0)} 10 { fr((0,1,0))=0.1 and isint((0,-1,0)) and isint((-1,0,0)) and
isint((1,0,0)) }
[plane3-rule]
rule: {(time)} 10 { (0,0,0)=0 and %check and (-1,0,-1)!=(0,1,-1) }
...
rule: {(time)} 10 { (0,0,0)=0 and %check and (0,-1,-1)!=(0,1,-1) }
```

FIGURE 13.21 Cell-DEVS model definition in CD++.

The algorithm produces a Voronoi diagram that can be used to determine a path equidistant from obstacles in the space. Paths are calculated by marking the intersections of expanding wavefronts propagated from given starting points. The model executes in two stages:

1. Object boundary detection: Each cell is examined and compared to a set of 12 *edge codes*. Each cell matching a configuration in this template uses the corresponding code (1–12) for the second stage. For instance, in Figure 13.20(a), we will use different codes for the top/bottom of the obstacles, the different corner areas, empty spaces, and the center of the obstacles.
2. Cells with edge codes are expanded in the space, and the cells in the intersections are considered as a part of the final Voronoi diagram.

The cellular model stores the original encoding for obstacles (0 or 1), the calculated edge code (1–12), a flag used during wavefront expansion, and the Voronoi diagram. We use a three-dimensional Cell-DEVS, in which each plane contains each of these values. The model definition, found in *./PathPlanning2.zip,* is as shown in Figure 13.21.

The model defines a 10 × 10 × 4 Cell-DEVS with four sets of rules (one for each plane). The model is effectively divided into four two-dimensional sections by using separate zones consisting of four plane regions:

- **Nothing rule** is used by the original data plane to keep the values from being changed.
- **Bound rule** encodes the edge directions in the data plane using 12 templates.

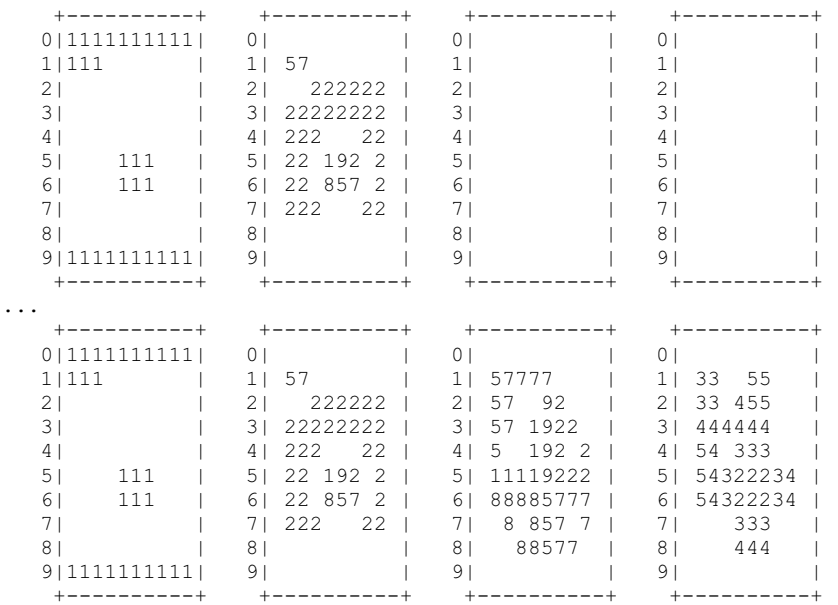


FIGURE 13.22 Partial boundary and one obstacle.

- **Plane2 rule** discards cells with edge codes 1–4 (they do not belong to an edge). Cells with edge codes 5–12 are given a flag value for propagation. The rules in this section carry over the values from the second plane that satisfy the criteria.
- **Plane3 rule** creates the Voronoi diagram. In the previous plane, cells receive data values from their immediate neighbors and propagate the data out from the starting point (i.e., points where these data wavefronts collide are those farthest away and equidistant from the starting obstacles; these are the points of interest when plotting a path for a robot). The Voronoi diagram stores the iteration number at which the cell was added to the diagram.

Figure 13.22 shows the execution of the model using a partial boundary and one obstacle. The initial values for the cell space contain a boundary on the upper and lower horizontal edges, as well as one small obstacle. The input values in the first plane remain unchanged, and, after one cycle, the edge codes in the second plane are generated. The third plane is initially populated with edge codes > 4, and these values are successively propagated across their neighborhoods (note the holes where cells were out of reach of their neighbors). Propagation stops when cells have no more nonflagged neighbors. The final plane contains the Voronoi diagram. Values in this diagram are derived from the simulation time divided by 10. The first values that appear in this plane are twos, just under 30 ms into the simulation. Because the first values on the diagram are twos, one should add that offset to find the desired values. In this case, for a robot of diagonal size 2, the points on the graph of value four or five represent viable travel paths, which can be used by the robot of 2 to travel, avoiding the obstacles.

13.4.3 Shortest Path Selection

The Voronoi diagram provides a number of possible paths; we need to find the shortest one. To do so, we built a Cell-DEVS model based on a flooding technique described in Tzionas, Thanailakis, and Tsalides [9]. In this model, a *valid* cell is one considered part of a valid path if its value is larger than or equal to the robot size. A cell with more than two valid neighbors is called a *node*. An *output*

node is a cell where the robot is located before moving, and an *end node* is the destination. The shortest path to the end node is based on the Manhattan distance.

The algorithm consists of two phases: *flooding* and *selection*. The flooding algorithm explores all possible paths starting on the output node, choosing only valid cells in parallel. When a node is found, the path is divided. If, during the exploration, two paths are crossed, only the one with the best value continues. Selection starts when we get to the end node: we backtrack, looking for the minimum cost according to the chosen criteria.

The Cell-DEVS implementation presented in Wainer [10] and found in *./PathPlanning.zip* uses a two-dimensional Cell-DEVS model, in which we initially include the Voronoi diagram encoded as the distances to the objects, as in Figure 13.22. The algorithm is based on the one presented in Tzionas et al. [9] (Figure 13.23).

Each cell encodes the information using the integer part for some states and the fractional for others, as follows:

0—obstacle;
1–10—distance to obstacles;
100–5000—distance covered (100 is the minimum distance, i.e., 0);
5001—final path;
.0—no flooding agent;
.1—search agent; and
.2—marking within the minimum path.

```
[path]
type : cell            dim : (32,32)          delay : inertial
border : noWrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1)  (1,0) (1,1)
localtransition : normal

[normal]
%Flooding rules: E, N, W, S
rule : {trunc(0,-1)+1+0.1} 10 {fractional(0,-1)) = 0.1 and (0,0) > 1.99 and (0,0) <100}
rule : {trunc(1,0)+1+0.1} 10  {fractional(1,0)) = 0.1 and (0,0) > 1.99 and (0,0) <100}
rule : {trunc(0,1)+1+0.1} 10  {fractional(0,1)) = 0.1 and (0,0) > 1.99 and (0,0) <100}
rule : {trunc(-1,0)+1+0.1} 10 {fractional(-1,0)) = 0.1 and (0,0) > 1.99 and (0,0) <100}

%Agent change to "no flooding" status
rule : {trunc(0,0)} 10 {fractional (0,0) = 0.1}

%Node rules
rule : {trunc(0,-1)+1+0.1} 10 {fractional(0, -1) = 0.1 and (0,0) > (0,-1)+1 and
             (0,0)<=5000}
rule : {trunc(0,0)}       1000 {fractional(0, -1) = 0.1 and (0,0) <= (0,-1)+1 and
             (0,0)>100}
rule : {trunc(1,0)+1+0.1} 10 {fractional(1, 0) = 0.1 and (0,0) > (1,0)+1 and
             (0,0)<=5000}
rule : {trunc(0,0)} 1000 {fractional(1, 0) = 0.1 and (0,0) <= (1,0)+1 and (0,0) >100}
rule : {trunc(0,1)+1+0.1} 10 {fractional(0, 1) = 0.1 and (0,0) > (0,1)+1 and
             (0,0)<=5000}
...

%Bactracking: W, S, E, N
rule : {trunc(0,0)+0.2} 10 {fractional(0,1)=0.2 and (0,0)+1=trunc(0,1) and (0,0)>=100 and
             (0,0) <5000}
rule : {trunc(0,0)+0.2} 10 {fractional(-1,0)=0.2 and (0,0)+1=trunc(-1,0) and (0,0) >=100
             and (0,0)<5000}
rule : {trunc(0,0)+0.2} 10 {fractional(0,-1)=0.2 and (0,0)+1=trunc(0,-1) and (0,0)>=100
             and (0,0)<5000}
rule : {trunc(0,0)+0.2} 10 {fractional(1,0)=0.2 and (0,0)+1=trunc(1,0) and (0,0)>=100 and
             (0,0) <5000}

%Final marking
rule : {5001} 10 {fractional(0,0)=0.2}
```

FIGURE 13.23 Minimum path search.

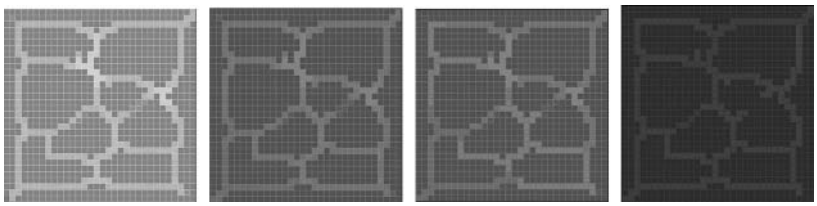


FIGURE 13.24 (a) Initial Voronoi diagram; (b and c) flooding; (d) selection.

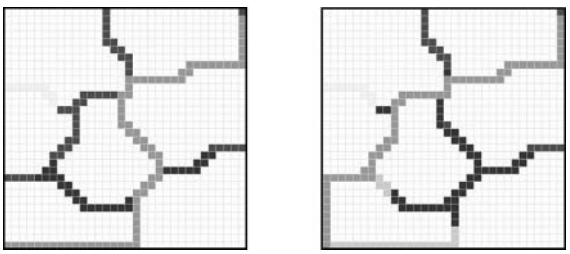


FIGURE 13.25 (a) Shortest path; (b) shortest path with modified Voronoi diagram.

We start by putting the value 100.1 in the output node; that is, we locate a search agent on the cell with the minimum distance to the destination. The flooding algorithm modifies each cell with the distance to the output node. The exploration rules check for an exploration agent in the neighborhood (0.1). They then check to ensure the cell is on exploration mode (i.e., it is occupied, and a distance is stored). In this case, the agent is moved to the cell, and one more is added to the distance covered. We then decide in which direction the nodes should continue the flooding. The backtracking rules check for nodes within the minimum distance path. The idea is to see if the neighbors belong to the shortest path and, in such a case, incorporate the cells to the minimum distance path (marking them with the value *5001*). We can find a minimal path, as seen in Figure 13.24.

Our implementation encodes the distance to the objects at the beginning of the process (in the Voronoi diagram). Figure 13.25 shows two examples in which we change the original Voronoi diagram by adding an extra connection in the bottom-left part of the diagram (which affects the shortest path found).

Exercise 13.9

Modify the initial Voronoi diagram and repeat the tests. Discuss the results obtained.

13.4.4 Self-Reconfiguring Robots

Self-reconfiguring robots are versatile in both their structure and the tasks they perform [11]. They are usually composed of a number of modules that can reshape according to the task to be carried out. Each robot is independent of the rest, and the robots act in parallel. This ability of reconfiguration leads to flow-based locomotion algorithms (allowing the robots to adapt to the terrain on which they have to travel).

In this section, we will show a model presented in Wainer [10] and found in *./reconfig.zip* that is based on Butler et al. [11], in which we study robotic locomotion in a two-dimensional plane. The model follows a flow-like locomotion pattern and is capable of (1) linear motion on the plane where modules move, (2) convex transitions into a different plane, and (3) concave transitions into

```
[reconfig]
type : cell            dim : (15,45)          delay : transport     border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-2) (1,-1) (1,0)(1,1)
localtransition : reconfig

%%% LEGEND
%%%  _: any cell value
%%%  a: robot not moving or obstacle
%%%  b: any cell moving
%%%  c: empty cell or obstacle

[reconfig]
% 001       001
% 011   =>  031
rule : 3 0 {(-1,-1)=0 and (-1,0)=0 and (-1,1)=1 and (0,-1)=0 and (0,0)=1 and (0,1)=1}
% ___       ___
% 001   =>  011
% 031       031
rule : 1 100 {(0,-1)=0 and (0,0)=0 and (0,1)=1 and (1,-1)=0 and (1,0)=3 and (1,1)=1}
% 001   =>     001
% 031          001
% ___          ___
rule : 0 100 {(-1,-1)=0 and (-1,0)=0 and (-1,1)=1 and (0,-1)=0 and (0,0)=3 and (0,1)=1}
...
```

FIGURE 13.26 Reconfigurable robots definition.

a different plane. The control algorithm uses local rules and is constructed as a cellular model. We will show the behavior of a self-reconfiguring robot, avoiding obstacles in a nonstructured space.

The cells use 11 different states: empty (0), occupied by a nonmoving module (1), occupied by an obstacle (2), or occupied by a robot moving in north (N), south (S), east (E), and west (W) directions (3–10). We use a modified Moore's neighborhood and 27 rules controlling the full behavior of a cell. Locomotion is produced in two phases. The first phase determines if the cell has to change its state and the new state it will reach; in the second phase, depending on the state of each neighbor, a cell might decide to cancel its decision or to go ahead as planned. The rules shown in Figure 13.26 define the different steps needed to execute the first configuration in Figure 13.27(b). We start with three moving modules (black cells), and the one in the bottom left decides to move up. In the next step, it moves and the third rule deletes the original one, obtaining the final configuration. Figure 13.27(b) shows the different movement rules available.

Figure 13.28 shows some of the execution results obtained when using a square topology. Particularly noteworthy is the fact that the robot climbs obstacles with a relative height of three units, and when it climbs down, it follows the shape of the terrain. The model was extended to a hexagonal topology, resulting in the same two-phase mechanism but fewer rules (21) and states (8). Figure 13.29 shows a graphical representation of the model, showing the local rules.

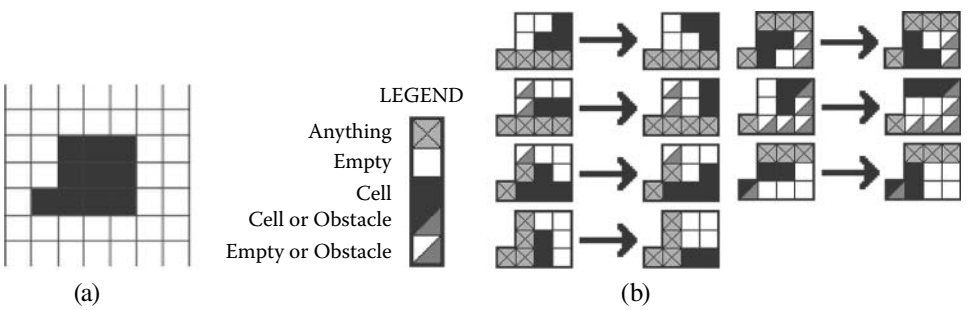


FIGURE 13.27 (a) Neighborhood shape; (b) model's rules.

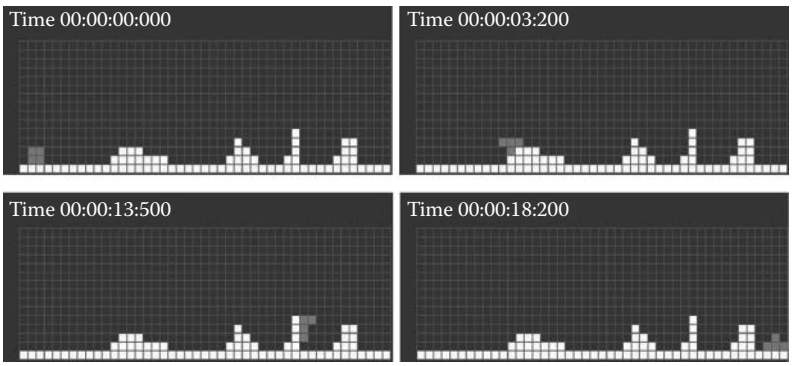


FIGURE 13.28 Model execution.

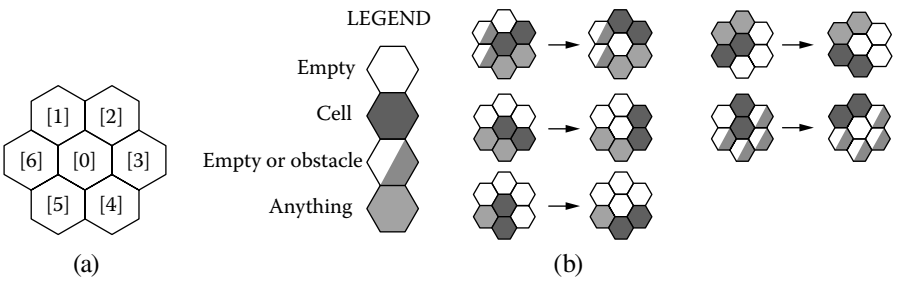


FIGURE 13.29 (a) Hexagonal neighborhood definition; (b) model's rules.

Figure 13.30 shows the model representation using hexagonal Cell-DEVS in CD++. The first rule checks whether a mobile robot on position [4] can move to the origin (with direction 1). To do so, we first check for possible collisions with other cells (first rule in Figure 13.30). Considering that the robot moves to the right, we first check whether cell [5] or cell [6] cannot move. In that case, we can move to the N. The next rules are used to empty the cell in that case. Figure 13.31 shows the model's execution. The results obtained are similar to those presented in Figure 13.28 but using the hexagonal topology. Using a square topology required 18.2 s to travel across all obstacles, while using the hexagonal topology required only 15.8 s.

```
[reconfig-robot-hexa]
dim : (15,45)          delay : transport   border : wrapped
neighbors : (-1,-1)(-1,0)(-1,1)(0,-1)(0,0)(0,1) (1,-2) (1,-1) (1,0) (1,1)

[reconfig]
rule: 1 100 {[0]=0 and [4]=1 and [5]=3 and ([6]=0 or [6]=2)}
rule: 1 100 {[0]=3}
rule: 4 0   {[0]=1 and [1]=0 and [2]=0 and [3]=0 and [4]=1}
rule: 0 100 {[0]=4 and [1]=0 and [2]=0 and [3]=0 and [4]=1}
rule: 1 100 {[0]=0 and [1]=0 and [5]=1 and [6]=4}
rule: 1 100 {[0]=4}
rule: 5 0   {[0]=1 and [1]=0 and [2]=0 and [3]=0 and [4]=0 and [5]=1}
rule: 0 100 {[0]=5 and [1]=0 and [2]=0 and [3]=0 and [4]=0 and [5]=1}
rule: 1 100 {[0]=0 and [1]=5 and [2]=0 and [6]=1}
rule: 1 100 {[0]=5}
...
```

FIGURE 13.30 Hexagonal model's rules.

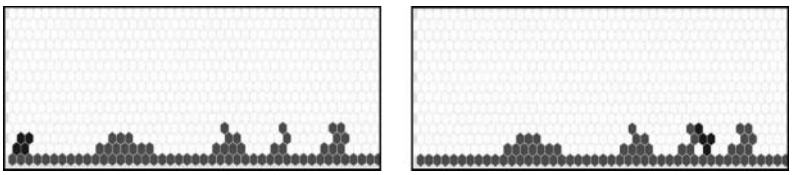


FIGURE 13.31 Hexagonal neighborhood execution.

Exercise 13.10

Modify the terrain topology for both models and repeat the tests.

13.5 DISCRETE-EVENT CONTROL OF A TIME-VARYING PLANT

Conventional adaptive control of unknown time-varying plants can be defined as in Figure 13.32. The goal of the controller we present here is to have the plant's output match the reference signal y^*, with zero control error.

The plant output $y(k)$ and the reference signal y^* are fed into a controller C_i. The control signal $u(k)$ is generated by the controller. The adaptive controller uses a plant model M_i, which tries to identify the plant behavior. The output $y(k)$ is compared to the estimated y_i, and the identification error e_i is obtained. Using a single identification model is efficient when the initial parameter estimation error is small and plant parameters are slowly varying over time. Nevertheless, when either of these conditions is not satisfied (i.e., for subsystem failures or changes in the operating environment), the use of multiple models is more appropriate. As seen in Figure 13.32, a finite number of models is evaluated by an index of performance, and the most suitable controller is applied to the plant. This approach is beneficial for maintaining control of a plant when there are parameter jumps [12].

Multiple model control demands a union of high-level decision making with mathematically complex algorithms. In this section, we present an implementation of such algorithms using QDEVS,

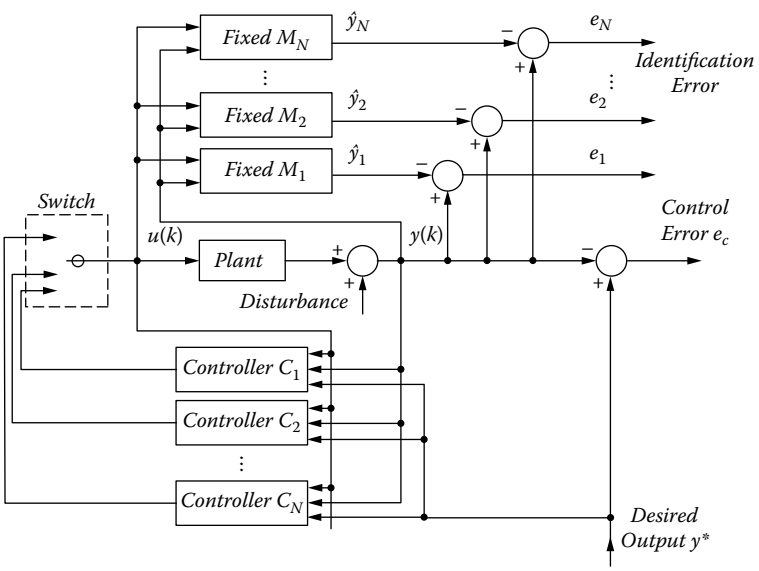


FIGURE 13.32 Multiple fixed models for control of an unknown plant. (Adapted from Narendra, K. S. et al. 2003. *International Journal of Adaptive Control and Signal Processing* 17:87–102.)

based on the work presented in Kofman [13,14]. Using this approach, the reference signal $y(k)$ (a continuous analog signal) is discretized using quantized DEVS with hysteresis. The discrete event solution permitted us to integrate discrete event and continuous components seamlessly in the plant, avoiding the common problems of controller wind-up (a saturation condition of the controller that avoids the error to get close to zero) and bursting during the parameter estimation process.

The first step was to create DEVS atomic models of both the adaptive controller (which uses three models) and the plant. The *plant* atomic model is defined by a second-order difference equation in discrete time:

$$y_p(k) = p_1(s)y_p(k-1) + p_2(s)y_p(k-2) + p_3(s)u_c(k-1) \tag{13.2}$$

where a piecewise constant parameter vector is defined as

$$\theta(s)^{\mathrm{T}} = \begin{bmatrix} p_1(s) & p_2(s) & p_3(s) \end{bmatrix} \tag{13.3}$$

Here, the *plant state* input $s = \{1, 2, 3\}$ determines what set of parameters the plant should operate on. This model also requires using its most recent outputs as inputs (a new plant output is made when the trigger is enabled).

The next step is to define the three plant identification models, M_i. Each M_i creates outputs as

$$y_i(k) = q_{i,1}y_p(k-1) + q_{i,2}y_p(k-2) + q_{i,3}u_c(k-1) \tag{13.4}$$

where the model's parameter vector is defined as

$$\theta_i^{\mathrm{T}} = \begin{bmatrix} q_{i,1} & p_{i,2} & p_{i,3} \end{bmatrix} \tag{13.5}$$

A second output is the modeling error, defined as

$$e_i(k) = \left(y_i(k) - y_p(k)\right)^2 \tag{13.6}$$

This error is used by the controller to determine which plant-identifying model's parameters are best for controlling the system (i.e., θ_1^{T}, θ_2^{T}, or θ_3^{T}).

We also defined a *unit delay* model, which is in charge of delaying the signal's propagation for one time unit in order to ensure that the plant-identifying models will be updated after a plant output is generated.

Finally, we built a model of the *controller,* whose goal is to have the plant's output match the reference signal y_r with zero error. This controller must analyze the available modeling errors (from the different plant-identifying models) and decide which is the most suitable. The parameters associated with the best-fit model are used to generate a control signal for the system. The certainty equivalence principle [15] is used with the chosen plant's parameters to calculate the control signal as follows:

$$u_c(k) = \frac{y_r(k+1) - \theta_i^{\mathrm{T}}\tilde{\phi}(k)}{q_i}, \text{ where } \tilde{\phi}(k) = \begin{bmatrix} -y_p(k) & -y_p(k-1) & 0 \end{bmatrix} \tag{13.7}$$

The following shows the DEVS definition of one of the models (in this case, *GenControl,* which represents a generic controller for the plant; a detailed specification for each of the models can be found in Campbell and Wainer [16]):

GenControl
X = {Yrin Ypin Ypdin Em1in Em2in Em3in} **Y** = {Uout, modelSelect}
S = {haveYr haveYp haveYpd}

δ_{int}(s) { passivate }

δ_{ext}(s, x, e) {
```
    switch (port) {
        case Yrin:
                if (haveYr == 0) haveYr = 1;
                Yr = Yrin.value;
        case Ypin:
                if (haveYp == 0) haveYp = 1;
                Yp = Ypin.value;
        case Ypdin:
                if (haveYpd == 0) haveYpd = 1;
                Ypd = Ypdin.value;
        case Em1in: em1 = Em1in.value;
        case Em2in: em2 = Em2in.value;
        case Em3in: em3 = Em3in.value;
    holdIn( active, Time( 0.001) ) ;
}
```

λ (s) {
```
    if (haveYr == 1 && haveYp == 1 && haveYpd==1) {
        haveYr = haveYp =haveYpd = 0;
        bestModel = 2; // initial guess
        U = (Yr-q21*Yp-q22*Ypd)/q23; // calc U as if Model 2 was best
        if (em1<em2 && em1<em3) { // Model 1 is best
                bestModel = 1;
                U = (Yr-q11*Yp-q12*Ypd)/q13;
        if (em3<em2 && em3<em1) { // Model 3 is best
                bestModel = 3;
                U = (Yr-q31*Yp-q32*Ypd)/q33;    }
                send output U to Port Uout
    send output bestModel to Port modelSelect
}
```

The variables *haveYxx* are used to ensure that we have all the arguments needed before computing the control value using Equation (13.7). The ports *Emxx* are used to determine which estimation model to use. The output function computes the next state, based on Equation (13.7).

Figure 13.33 shows the implementation of the plant model in CD++, as found in *./DEController.zip.* In this case, the plant model also waits for all the inputs needed (*U, Ypdin,* etc.), and it then computes the current value, according to the plant state model chosen. Figure 13.34 shows the structure of a coupled model integrating the previously presented models, based on the generic idea presented in Figure 13.32.

```
Model &Plant::externalFunction( const ExternalMessage &msg ) {
        if (msg.port() == Uin) U = (double)msg.value();
        if (msg.port() == Ypdin) Ypd = (double)msg.value();
        if (msg.port() == Ypddin) Ypdd = (double)msg.value();

        if (msg.port() == Trigger) {
                if (createOutput == 0) {
                        createOutput = 1;
                        scrap = (double)msg.value();
                }
        }
        if (msg.port() == plantState) pState = (int)msg.value();

    holdIn( active, Time( static_cast< float >(.1) ) ) ;
}

Model &Plant::internalFunction( const InternalMessage & ) {
        passivate();
}

Model &Plant::outputFunction( const InternalMessage &msg ) {
        if (createOutput == 1) {
                createOutput = 0;
                if (pState == 1) Yp = p11*Ypd+p12*Ypdd+p13*U;
                if (pState == 2) Yp = p21*Ypd+p22*Ypdd+p23*U;
                if (pState == 3) Yp = p31*Ypd+p32*Ypdd+p33*U;
                sendOutput( msg.time(), Ypout, Yp ) ;
        }
}
```

FIGURE 13.33 Plant model definition in CD++.

In order to study the plant behavior, we introduced a quantized version of each of the models. The following example used a reference signal defined as

$$y_{ref}(k) = \sin\left(2\pi k/20\right) + \sin\left(2\pi k/10\right) + 2 \tag{13.8}$$

Using this signal as an input, we applied quantized DEVS with hysteresis ($Q = 0.1$, $n = 2$). Given that the signal being quantized is $\in (0, 2)$, the normalized quantum size can be considered $\overline{Q} = 0.05$. The resulting quantized signal is a discrete time signal that contains discrete event changes. To remove the discrete time component, we use a list that contains the signal's event changes and their associated event times. Figure 13.35 shows the quantized signal.

Figure 13.36 shows the results obtained when we used single model adaptive control with time-invariant parameter $p^{\mathrm{T}} = [0.6\ 0.2\ 0.1]$. A discrete event controller was implemented using RLS and certainty equivalence control [15]. The adaptive model uses initial parameter estimates defined as $\theta_{init}^{\mathrm{T}} = [1.1 \quad -0.3 \quad -0.4]$.

The control error remains roughly the same, despite the difference in quantum size for discretization of the reference signal. The RLS adaptive algorithm was able to converge more quickly when the quantum size was smaller. This is inherent because increased excitation increases performance of adaptive algorithms. It is worth noting that this discrete event implementation of adaptive control overcomes the issue of controller wind-up. Controller wind-up, or the parameter burst phenomenon, occurs in discrete time when long periods pass without excitation while adaptation continues. Using the discrete event notation, adaptation does not occur unless there are event changes.

Our next example uses multiple model control with plant states $p_1^{\mathrm{T}} = [0.6 \quad 0.2 \quad 2.0]$, $p_2^{\mathrm{T}} = [0.1 \quad 0.8 \quad 2.5]$, and $p_3^{\mathrm{T}} = [0.2 \quad 0.5 \quad 1.0]$. The controller uses parameter estimates $\theta_1^{\mathrm{T}} = [0.6 \quad 0.2 \quad 2.0]$, $\theta_2^{\mathrm{T}} = [0.1 \quad 0.8 \quad 2.5]$, and $\theta_3^{\mathrm{T}} = [0.2 \quad 0.5 \quad 1.0]$. The simulation results presented in Figure 13.37 show a multiple model controller forced to always use the first plant identification

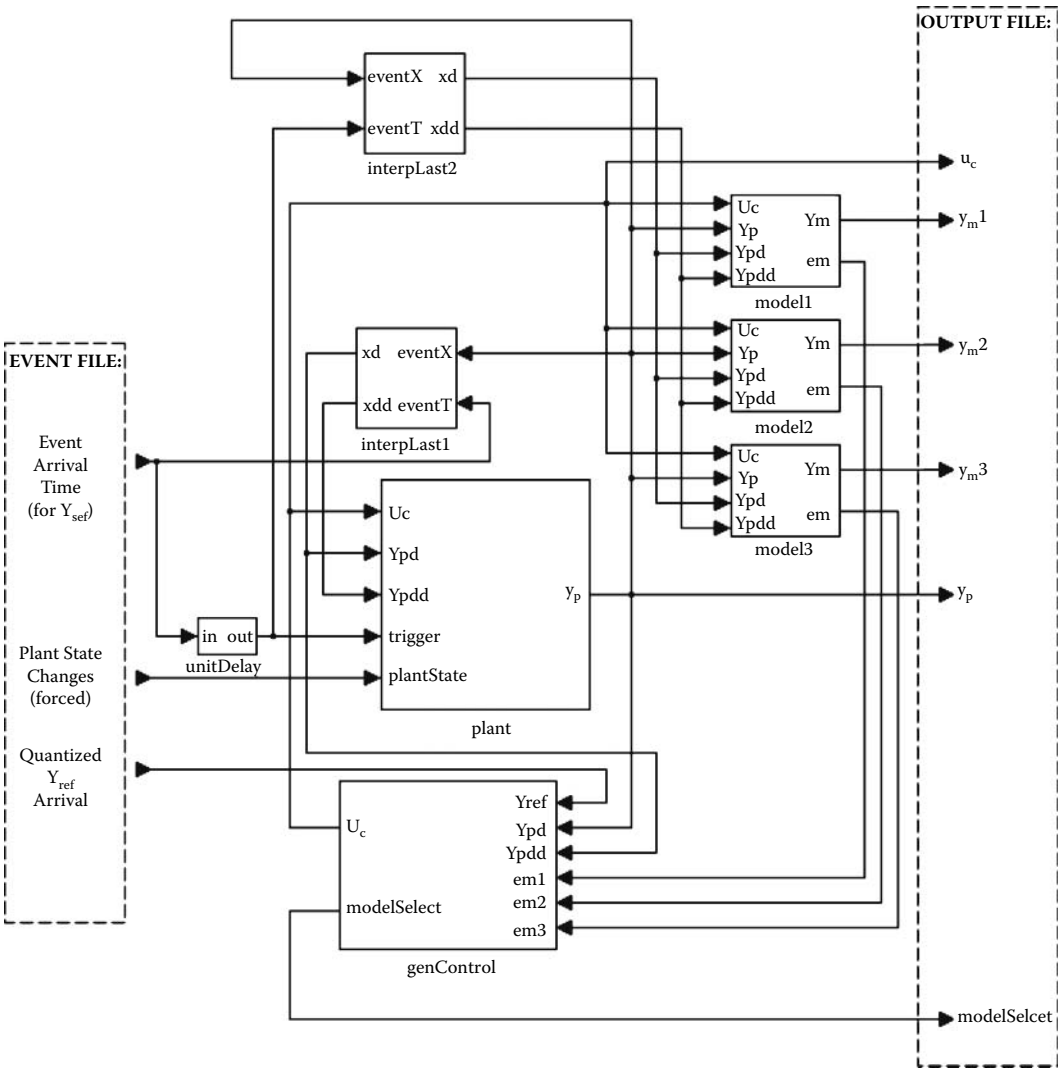


FIGURE 13.34 Conceptual coupled model of multiple model controller.

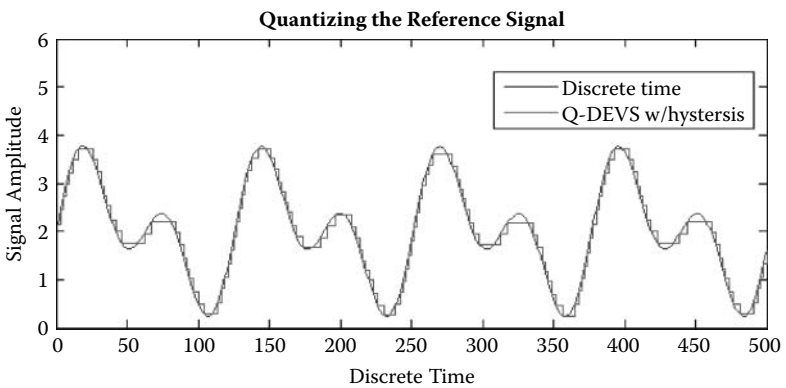


FIGURE 13.35 Discretization of reference signal ($n = 2$, $Q = 0.10$).

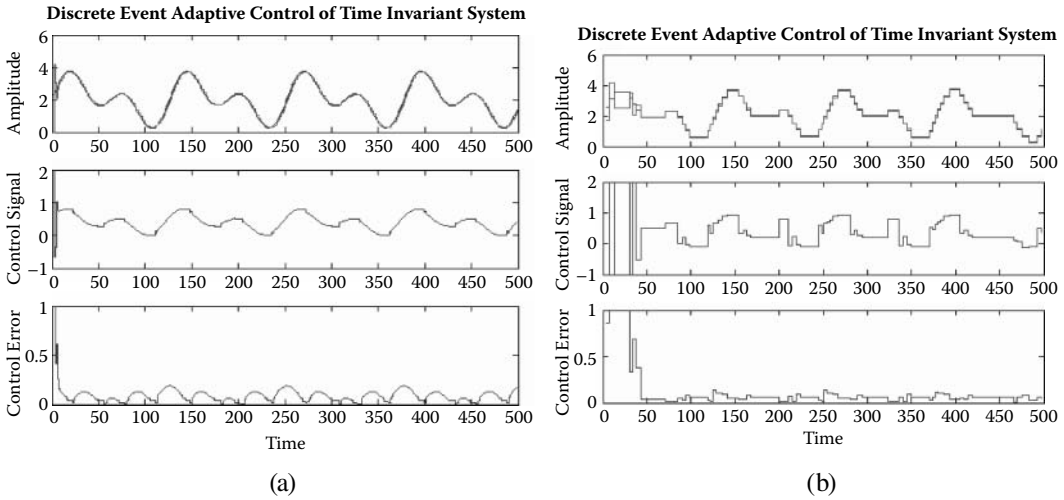


FIGURE 13.36 Adaptive control using: (a) $Q = 0.02$; (b) $Q = 0.2$.

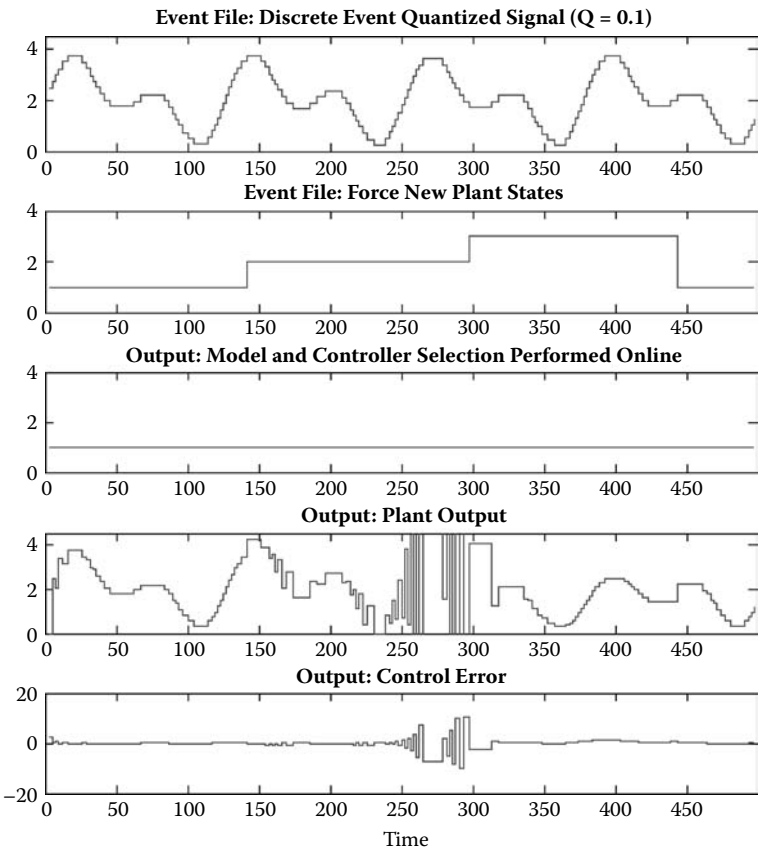


FIGURE 13.37 CD++ simulation with parameter jumps, using only one plant-identifying model.

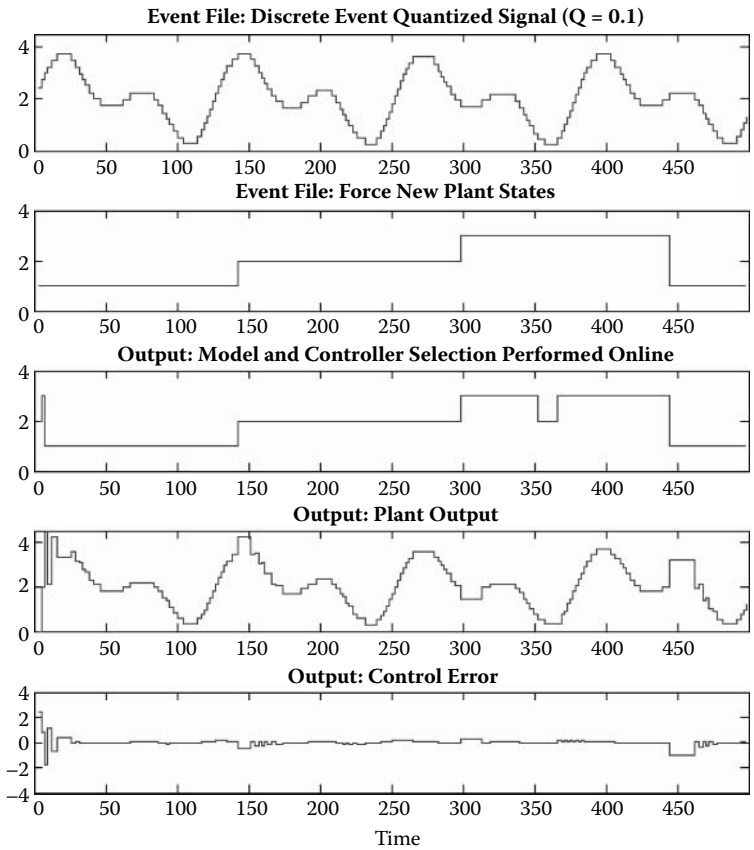


FIGURE 13.38 CD++ simulation containing parameter jumps, using a fixed controller.

model θ_1^T. Using the fixed parameter controller, stable control was achievable for plant states P_1 and P_3. At plant state P_3, the closed loop system becomes unstable, eventually yielding unbounded plant outputs.

In Figure 13.38, the multiple model controller is allowed to operate as designed, switching among its plant-identifying models. The simulation displays the advantages of multiple model control. Because a matching identification model was designed a priori, the controller was able to find it and use its parameters. For this deterministic scenario, control error existed only at the time coinciding with each jump in plant parameters. During the simulation initialization, the instantaneous error of each model was zero. This required modeling of several reference signal events and their corresponding triggered plant outputs in order to identify which controller was the most suitable. During operation, only at time 355 did a false model switch occur. The source of the false switch was due to two models' having almost zero modeling error.

13.6 NETWORKING PROTOCOLS FOR LOCAL AREA NETWORKS

One of the main areas of application of discrete-event modeling and simulation is the analysis of networking protocols. The current scale of these networks and their high level of heterogeneity make it very difficult to face the design of new protocols [17]. Various discrete-event simulators are readily available (both academic and commercial, such as NS-2 [18] and its successor NS-3, GloMoSim [19], OPNET [20], and OMNeT++ [21]).

In this section, we introduce different models for simulating user-defined topologies to assess network functionality; modular design allows the addition of new models easily, while the models themselves are flexible to permit future enhancements. The use of a DEVS-based network simulator provides facilities to carry out formal tests, model sharing between different DEVS-based toolkits, the ability to execute the models on varied middleware and hardware, and the possibility to define models using different techniques interacting within the same environment. This could allow including non-network entities that affect network operation (like those introduced in previous chapters in this book), providing results that are more realistic. Most models introduced in this section can be found at *./NetworkIP.zip*.

13.6.1 Hub

Hubs are simple layer 1 devices that regenerate received data to all connected devices. The hub model presented in this section allows us to show how to define a very simple model for networking applications while permitting interconnecting of other models and creating complex topologies with which to experiment. The model architecture is as shown in Figure 13.39.

The hub atomic model specification can be defined as

$$Hub = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D> \tag{13.9}$$

X = { *In*: receives data from interconnected devices; *Set*: sets hub-specific information };
S = {Sigma, X, Preparation Time}
Y = { *Out1..n*: 1st…nth connected device };
$\delta_{int}(e, s)$: {
  case phase:
    active: passivate
}

$\delta_{ext}(s, x, e)$: {
  case port:
    In: set localvalue to msg.value
    Set: set local data field (hub identifier) to msg.value
}

$\lambda(s)$: {
  Output data to all output ports
}

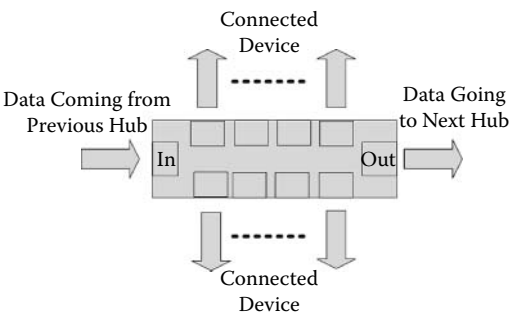


FIGURE 13.39 Hub structure.

```
hub::hub( const string &name ): Atomic( name ), set( addInputPort( "set" ) ),
in( addInputPort( "in" ) ), out1( addOutputPort( "out1" ) ), out2( addOutputPort( "out2"
) ) ,
out3(addOutputPort("out3") ), out4(addOutputPort( "out4" ) ), out5(addOutputPort( "out5" ) ),
out6( addOutputPort("out6" ) ), out7( addOutputPort( "out7" ), out8( addOutputPort( "out8" ) ),
out9( addOutputPort( "out9" ) ) , preparationTime( 0, 0, 0, 0 ) {
        string time( MainSimulator::Instance().getParameter( description(), "preparation"
) ) ;

        if( time != "" ) preparationTime = time ;
}

Model &hub::initFunction() {
        localValue = 0;
        send = false;
        MAC = 0;
}

Model &hub::externalFunction( const ExternalMessage &msg ) {
        if( msg.port() == in )          {
                localValue = msg.value();
                send = true;
        } else  if( msg.port() == set ) MAC = msg.value();

        holdIn( active, preparationTime);
}

Model &hub::internalFunction( const InternalMessage & ) {
        passivate();
}

Model &hub::outputFunction( const InternalMessage &msg ) {
        if(send)        {                       // regenerating traffic to all connected ports

                sendOutput( msg.time(), out1, localValue);
                sendOutput( msg.time(), out2, localValue);
                sendOutput( msg.time(), out3, localValue);
                sendOutput( msg.time(), out4, localValue);
                sendOutput( msg.time(), out5, localValue);
                sendOutput( msg.time(), out6, localValue);
                sendOutput( msg.time(), out7, localValue);
                sendOutput( msg.time(), out8, localValue);
                sendOutput( msg.time(), out9, localValue);
                send = false;
        }
}
```

FIGURE 13.40 Hub model definition.

Figure 13.40 shows the model implementation in CD++ following the model's specification. The model includes two input ports (*in,* which receives input data, and *set,* to set configuration information for the hub) and nine output ports. When an input is received on the *in* port, its value is stored. If the value arrives through the *set* port, the MAC variable is set. A delay representing the circuit latency is programmed. When consumed, the output function retransmits the current value. The internal transition function then passivates the model.

Any connected device can send data onto the hub to be broadcast to all other devices. The model proved successful in linking multiple hosts together, providing simple local networks, and proved useful in creating subnets.

Exercise 13.11

Test the hub model with varied input data.

13.6.2 Alternating Bit Protocol (APB)

This communication protocol provides a simple mechanism to ensure reliable transmission through an unreliable network. The general behavior of the protocol is as follows [22]:

- The sender sends a packet and waits for an acknowledgment.
- If the acknowledgment does not arrive within a predefined time, the packet is re-sent.
- When the expected acknowledgment is received, the next packet is sent.
- In order to distinguish two consecutive packets, the sender adds an additional bit on each packet (called an *alternating bit* because the sender uses zero and one alternatively).

The ABP model we created, found in *./alternatebitprot.zip,* consists of three components: a sender, the network, and a receiver. The network is decomposed further to two subnets corresponding to a sending and receiving channel, respectively, as seen in Figure 13.41.

The receiver accepts data and sends back an acknowledgment extracted from the received data after some time. The subnets forward the data received after a delay. However, in order to simulate the unreliability of the network, only 95% of the data is passed on to each subnet (i.e., 5% of the data is lost).

$$\text{Receiver} = <S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta> \tag{13.10}$$

S = {passive, active}; X = {in}; Y = {out}
δ_{int}(active) = passive
δ_{ext}(in, passive) = active
δ_{ext}(in, active) = active
λ(active) { send *in* % 10 to port *out* } //extract the alternating bit and send back
ta(passive) = INFINITY
ta(active) = receiving_time

The receiver and subnets have two phases: *passive* and *active.* Whenever they receive an input, they will become *active* and send out an output with a probability of 95% after a delay representing the latency of the network. The state will then be changed back to *passive.* The *receiving_time* of the receiver is a constant; the *delay* value in subnets is a nondeterministic value (expressed by a normal distribution with a given mean and standard deviation) chosen by the user. Figure 13.42 shows the implementation of the sender.

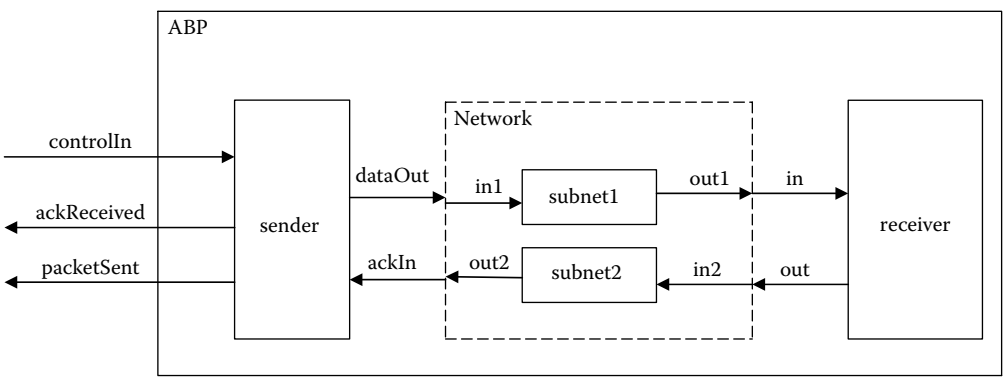


FIGURE 13.41 Structure of the ABP model.

```
Sender::Sender( const string &name ): Atomic( name ), controlIn( addInputPort( "controlIn") )
  , ackIn( addInputPort( "ackIn" ) ), dataOut( addOutputPort( "dataOut" ) )
  , packetSent( addOutputPort( "packetSent" ) ), ackReceived( addOutputPort( "ackReceived") )
  , preparationTime( 0, 0, 10, 0 ), timeout(0, 0, 20, 0){

   alt_bit = 0;
}

Model &Sender::externalFunction( const ExternalMessage &msg ) {
   if( msg.port() == controlIn  && state() == passive)       {
        totalPacketNum = static_cast < int > (msg.value());
        if (totalPacketNum > 0)       {
               packetNum = 1;
               ack = false;
               sending = true;
               alt_bit = packetNum % 2;      //set initial alt_bit
               holdIn(active, preparationTime );
        }
   }
   if( msg.port() == ackIn  && state() == active)    {
        if (alt_bit == static_cast < int > (msg.value()))           {
               ack = true;
               sending = false;
               holdIn(active, Time::Zero) ;
        }
   }
}

Model &Sender::internalFunction( const InternalMessage & ){
if (ack)       {
   if (packetNum < totalPacketNum)           {
        packetNum ++;
        ack = false;
        alt_bit = (alt_bit + 1) % 2;
        sending = true;
        holdIn( active, preparationTime );
        }
   else
        passivate();
   }
else    {
   if (sending)        {
        sending = false;
        holdIn(active, timeout);
        }
   else         {
        sending = true;
        holdIn( active, preparationTime );
        }
   }
}

Model &Sender::outputFunction( const InternalMessage &msg ){
if (sending)   {
   sendOutput( msg.time(), dataOut, packetNum * 10 + alt_bit ) ;
        sendOutput(msg.time(), packetSent, packetNum ) ;
   }
   else
        if (ack)  sendOutput( msg.time(), ackReceived, alt_bit ) ;
}
```

FIGURE 13.42 ABP sender atomic model.

A packet sent out by the sender is just the packet sequence number and an alternating bit (e.g., 11 for the first packet, 100 for the 10th packet). The packet sequence number is sent to the *packetSent* port, and the packet sequence number plus the alternating bit (e.g., 11 for the first packet, 100 for the tenth packet) are sent to the *dataOut* port. The *controlIn* signal is a positive integer indicating how many packets should be sent in a session. When a *controlIn* signal is received, the sender changes

INPUT	OUTPUT
00:00:10:00 in 11	00:00:20:000 out 1
00:00:30:00 in 20	00:00:40:000 out 0
00:00:45:00 in 31	00:01:02:000 out 1
00:00:52:00 in 31	00:01:45:000 out 0
00:01:25:00 in 40	00:02:05:000 out 1
00:01:35:00 in 40	
00:01:55:00 in 51	

(a)

INPUT	OUTPUT
00:00:10:00 in 11	00:00:12:987 out 11
00:00:20:00 in 20	00:00:21:796 out 20
...	...
00:02:10:00 in 120	00:02:13:687 out 120
00:02:20:00 in 131	00:02:21:055 out 131
00:02:30:00 in 140	00:02:43:679 out 151
00:02:40:00 in 151	

(b)

FIGURE 13.43 ABP execution: (a) receiver execution; (b) subnet execution.

from the initial phase *passive* to *active*. It then switches to the *sending* mode, transmitting a packet plus an alternating bit. When the *sending_time* is consumed, the packet is assumed to be sent out, and the sender starts waiting for an acknowledgment during a *timeout* delay. If it expires, the sender will re-send the previous packet [with the alternating bit]. If the expected acknowledgment is received before the *timeout,* the sender will send the next packet. It will change back to *passive* phase when all packets have been sent out successfully. An output will be generated when a packet is sent out (*packetSent, dataOut*) or an expected acknowledgment is received (*ackReceived*).

The coupled model network and ABP models are defined as in Figure 13.41. Figures 13.43 and 13.44 show the simulation results of the receiver, the subnet, the sender, and the whole network. In Figure 13.43(a), the input of the receiver represents the packet number, and the last digit is zero or one (alternating bit). The output of the receiver (acknowledgment) is the alternating bit extracted from the input (*receiving_time* = 10 s). If a new packet arrives while the receiver is already processing one, the older packet should be discarded. For instance, if packet 1 arrives at 45 s and again at 52 s, the first one is discarded. If the duration between two consecutive inputs is less than or equal to the *receiving_time,* the first input is discarded. The output of the subnet should be exactly as the input with some packets lost (with a probability of 95% according to a random function). The latency is given by a normal distribution (with mean of 3 s and standard deviation of 1 s). The output is not deterministic due to the random function in the subnet model (and several inputs will not generate outputs). The example in Figure 13.43(b) shows an output in which the event with value 140 is lost in the subnet.

As discussed earlier, the sender has two inputs: *controlIn* and *ackIn*. *controlIn* (a positive integer) indicates how many packets should be sent in a session. *ackIn* is the acknowledgment received from the receiver (zero or one). When the sender receives an acknowledgment from *ackIn,* it compares the acknowledgment with its current alternating bit. If they are equal, it generates an output to the *ackReceived* port and sends the next packet. If an expected acknowledgment is not received and the *timeout* expires, the previous packet is re-sent. Figure 13.44 shows the execution of this model,

INPUT	OUTPUT
00:00:00:00 controlIn -1	00:00:25:000 dataout 11
00:00:05:00 controlIn 0	00:00:25:000 packetsent 1
00:00:10:00 ackIn 0	00:00:30:000 ackreceived 1
00:00:15:00 controlIn 5	**00:00:40:000 dataout 20**
00:00:30:00 ackIn 1	**00:00:40:000 packetsent 2**
00:01:30:00 ackIn 0	**00:01:10:000 dataout 20**
00:01:55:00 ackIn 1	**00:01:10:000 packetsent 2**
00:02:20:00 ackIn 1	00:01:30:000 ackreceived 0
00:02:45:00 ackIn 0	00:01:40:000 dataout 31
	00:01:40:000 packetsent 3
	00:01:55:000 ackreceived 1
	00:02:05:000 dataout 40
	00:02:05:000 packetsent 4
	00:02:35:000 dataout 40
	00:02:35:000 packetsent 4
	00:02:45:000 ackreceived 0

(a)

INPUT	OUTPUT
00:00:10:00 in1 11	00:00:12:987 out1 11
00:00:15:00 in2 1	00:00:16:796 out2 1
00:00:20:00 in1 20	00:00:21:957 out1 20
00:00:25:00 in2 0	00:00:28:035 out2 0
00:00:30:00 in1 31	00:00:32:182 out1 31
00:00:35:00 in2 1	00:00:38:160 out2 1
00:00:40:00 in1 40	00:00:48:655 out2 0
00:00:45:00 in2 0	

(b)

FIGURE 13.44 APB execution: (a) sender and (b) network.

```
00:00:20:000 packetsent 1
00:00:34:783 ackreceived 1
00:00:44:783 packetsent 2
00:00:59:775 ackreceived 0
00:01:09:775 packetsent 3
00:01:25:117 ackreceived 1
00:01:35:117 packetsent 4
00:01:52:621 ackreceived 0
00:02:02:621 packetsent 5
00:02:32:621 packetsent 5
00:02:47:851 ackreceived 1
00:02:57:851 packetsent 6
```

FIGURE 13.45 APB execution: top level.

including the behavior of the model upon reception of illegal events.

The first three events in the first column are ignored. When the sender receives the order of transmitting five packets, it starts doing so (`00:00:25:000 dataout 11`). We then receive an ACK (`00:00:30:00 ackIn 1`), and we start transmitting packet 2. The next event (`00:01:30:00 ackIn 0`) simulates a lost packet. We retransmit the packet (`00:01:10:000 dataout 20`), which is acknowledged (`00:01:30:00 ackIn 0`). The event `00:02:20:00 ackIn 1` simulates a wrong acknowledgment. In this case, the packet is retransmitted. The network coupled model execution can be seen in columns 3 and 4. Each packet is transmitted after the transmission delay. The event at 40:000 is lost during the test.

The coupled model for the ABP integrates the previous components. The input of the top model is just *controlIn,* a positive integer indicating the number of packets that needs sending in a session. The outputs indicate when a packet is sent out (*packetSent*) and when an expected acknowledgment is received (*ackReceived*). Figure 13.45 is an example of the outputs, which are nondeterministic due to the randomness in the network. In this test, packet 5 is sent twice due to the packet loss.

In the top model, if the second *controlIn* input comes before the first *controlIn* finishes, the second input will be discarded as shown in the sender atomic model. If another *controlIn* input comes after the first *controlIn* finishes its session, it will be executed normally. The ABP model generates the expected results according to the specifications. This is a good example showing how to define multiresolution models in CD++. We can see how the outputs at the level of the sender and receiver are hidden when we simulate the whole network (which is only considering the execution at the top level).

Exercise 13.12

Reproduce the tests presented in this section. Repeat the simulations with new input data.

Exercise 13.13

Modify the model and simulate the transmission of real data. To do so, create a text file with a text to be transmitted and use the characters in the file as the data transmitted. Check the correctness of the data transmitted and the simulation results (considering the possibility of data lost). Ensure that the text transmitted is received completely.

13.6.3 A Cellular Model for Cryptography

Cryptography applications are widely used in different contexts, ranging from hiding passwords in computers to advanced mechanisms for transmitting information that cannot be detected by external users. Gutowitz [23] considered the use of one-dimensional cellular automata (CA) as a possible cipher mechanism. Wolfram [24] discusses a variety of models in the application of CA for cryptography. If the CA are nonreversible, they are simple to encrypt but very difficult to decrypt (making them candidates for public keys).

In one-dimensional CA, there are 256 different rules that can be implemented, based on the neighborhood pattern. The following table shows a binary representation of the rules to be applied. The first row shows the precondition for triggering the rule (i.e., the values of the neighborhood pattern). The second row shows the cell postcondition after triggering the rule. The table shows rule 30, which is proposed in Gutowitz [23] as a candidate for use in cryptography applications. If we take

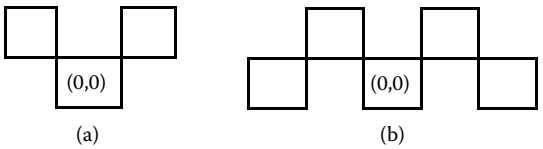


FIGURE 13.46 Encryption/decryption neighborhoods.

```
rule : 1 100 { (0,0) = 1 }
rule : 0 100 { (0,0) = 3 }
rule : 2 100 { (-1,-1) = 2 or (-1,1) = 2 }
rule : 1 100 { (0,0) = 2 and (-1,-1) = 1 and (-1,1) = 1 }
rule : 1 100 { ( 0, 0) = 2 and (-1,-1) = 0 and (-1, 1) = 0}
```

FIGURE 13.47 Encryption rules.

the bottom row bits (00011110) and consider the binary value represented, we obtain the number 30 (thus, it is called "rule 30").

Neighbor pattern	111	110	101	100	011	010	001	000*
State	0	0	0	1	1	1	1	0

We have defined a modified version of this rule using a two-dimensional Cell-DEVS to encrypt binary messages and another one for decrypting. We used a cell space of 20 × 20 cells and a neighborhood defined as in Figure 13.46(a). A cell can use the following values:

0—a bit to be encoded;
1—a bit to be encoded;
2—a cell ready to take the binary value that results from encrypting its neighbors; and
3—a cell that must not be encrypted but, rather, become a zero because the encrypted message needs to be padded in zeros.

The implementation of these rules in CD++ can be seen in Figure 13.47.

Figure 13.48 shows the encryption of the message 101100111000111101. The second line is used as a scratchpad for computing the value of the next row, and computation ends after 20 cycles. The result is that, after 20 iterations, the value becomes `1011010100100100000`.

The decryption model also uses a 20 × 20 space, the neighborhood shape in Figure 13.46(b), and the rules in Figure 13.49.

Exercise 13.14

Use new input data to encrypt and obtain the encrypted version of the message.

Figures 13.50 and 13.51 show a decryption test that uses the model to decrypt the encoded message that resulted from the preceding encryption process (101100111000111101 → 1011010100100100000) in 20 iterations, as we can see in Figure 13.50.

The final version in Figure 13.51 shows the message after 171 iterations.

Exercise 13.15

Execute the decryption model to ensure the original message presented in Exercise 13.14 can be decrypted.

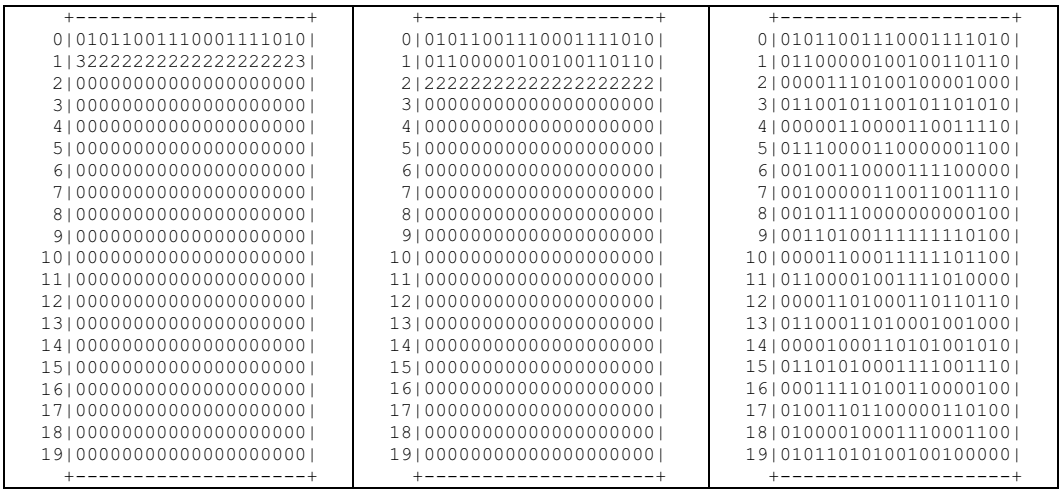


FIGURE 13.48 Encryption simulation.

```
rule : 1 100 { (0,0) = 1 }
rule : 0 100 { (0,0) = 0 }
rule : 1 100 { (0,0) = 2 and (-1,-1) = 0 and (0,-2) = 0 }
rule : 0 100 { (0,0) = 2 and (-1,-1) = 0 and (0,-2) = 1 }
rule : 0 100 { (0,0) = 2 and (-1,-1) = 1 and (0,-2) = 0 }
rule : 1 100 { (0,0) = 2 and (-1,-1) = 1 and (0,-2) = 1 }
rule : 1 100 { (0,0) = 2 and (-1,1) = 0 and (0,2) = 0 }
rule : 0 100 { (0,0) = 2 and (-1,1) = 0 and (0,2) = 1 }
rule : 0 100 { (0,0) = 2 and (-1,1) = 1 and (0,2) = 0 }
rule : 1 100 { (0,0) = 2 and (-1,1) = 1 and (0,2) = 1 }
```

FIGURE 13.49 Decryption rules.

13.6.4 Host

We defined a model simulating a computer host, which includes the different layers of the TCP/IP protocol stack: application, transport, network, data link, and physical. The structure of the coupled host model is shown in Figure 13.52. A version of this model was introduced in Ahmed et al. [25].

13.6.4.1 The Application Layer

The front end of the host model is the application layer according to the TCP/IP protocol stack, illustrated in Figure 13.53. In our host, it is designed as a simple atomic model. The layer manipulates the data received from the user in a way to identify the application type sending the data. This step is done to facilitate creating a connection manager. We defined a DEVS model in CD++ using the following specification:

$$\text{AL} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D> \tag{13.11}$$

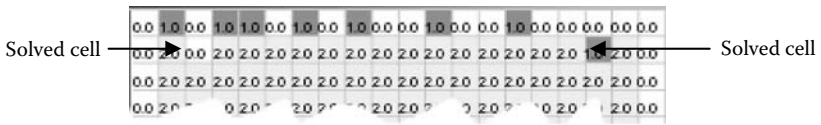
X = {HTTP_In $\in N$, FTP_In $\in N$, TelNet_In $\in N$, SMTP_In $\in N$, SNMP_In $\in N$, TransportLayer_In $\in N$ };
S = {Sigma, X, Preparation Time}
Y = { ApplicationOut, parsedApplicationLayer $\in N$ };

0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0
0.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.0

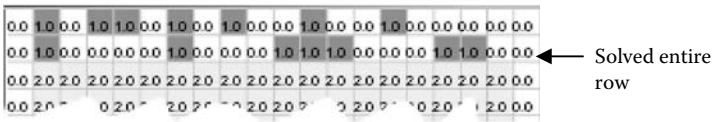
FIGURE 13.50 Initial state of the decryption model.

```
δext(s,e,x) {
   if (passive)
   Case msg.port
      HTTP_In:
      FTP_In:
      SMTP_In:
      SNMP_In:
      TelNet_In:
              Identify protocol port and save data, signal output function to send to transport
      In: signal output function to output to user
      Else
              continue
}

δint(s,e) {
      Case phase
              active: passivate
              Default: continue
}

λ(s) {
      Send application value to application out
}
```

After one iteration of the decoding process, we obtain:



and after 9 iterations of the process, we get



The final version shows the message after 171 iterations.

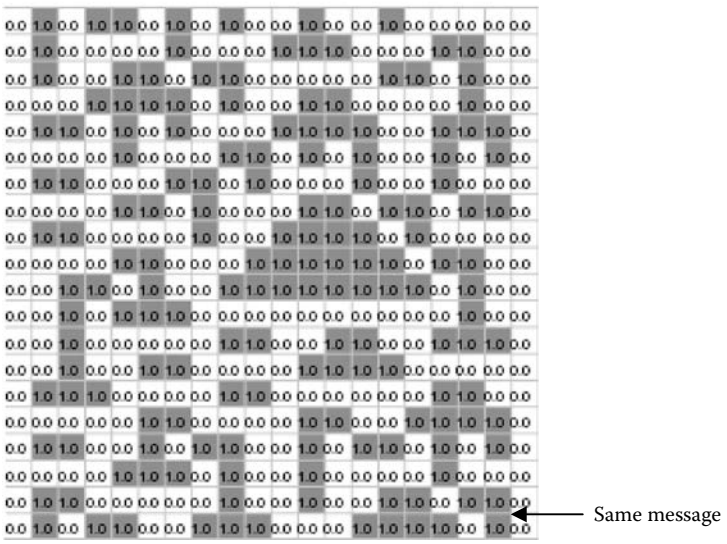


FIGURE 13.51 Final state of the decryption model.

The layer model parses the input data, uses the port value, and forwards the newly created value to the transport layer. This layer models a host generating and receiving data routed from various internetworking devices and different services and protocols. Figure 13.54 shows a simulation execution for the application layer model when it receives data.

According to the specification, the model adds the application port variable, and then it outputs the information to the transport layer. The simulation starts with two HTTP requests, which are serviced in 500 ms (port 80 is added to the packet). A similar result is obtained when we activate the SMTP service in port 25.

13.6.4.2 The Transport Layer

This layer is responsible for reliable, end-to-end data transfer throughout the network. There are many protocols functioning in this layer; however, as a starting point, TCP was chosen. Figure 13.52 (as well as Figure 13.55, which shows a subset of the original figure) shows how the transport model was broken in two (full-duplex communications). The transmitter model is responsible for receiving data from the application layer model, parsing it, and adding sequence and acknowledgment numbers, a window size, and a checksum to the original data received (fields required for service level agreement simulations). The data received are transformed to the format shown in Figure 13.56, according to the protocol requirements [26].

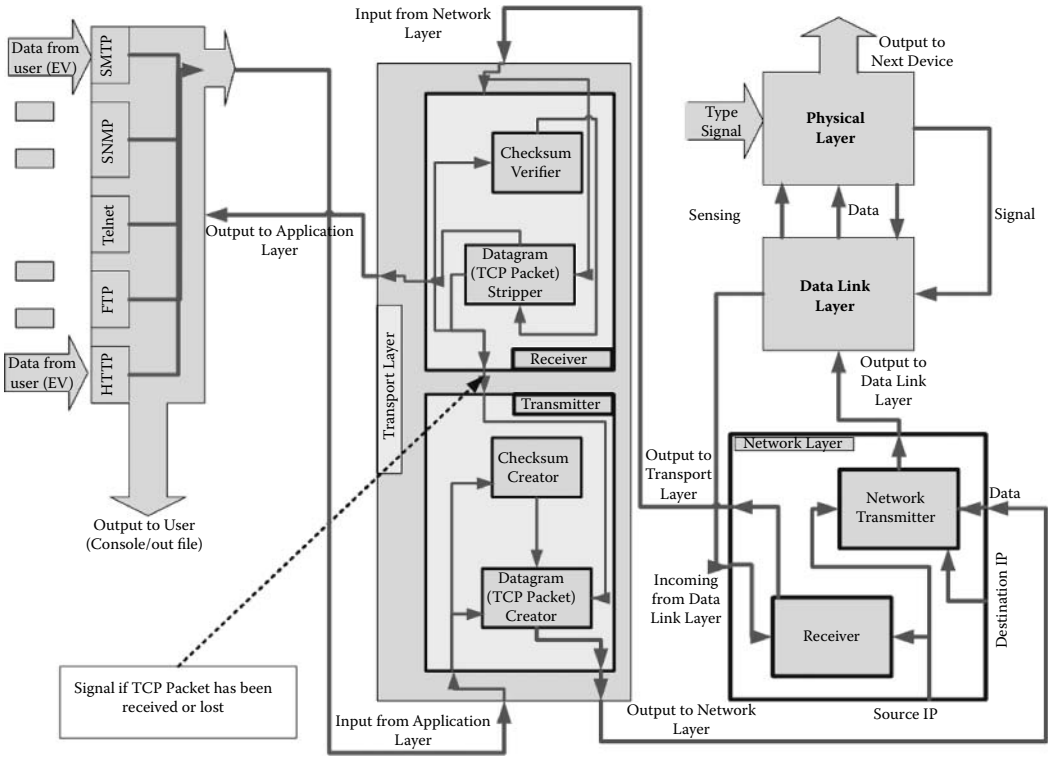


FIGURE 13.52 Host coupled model [25].

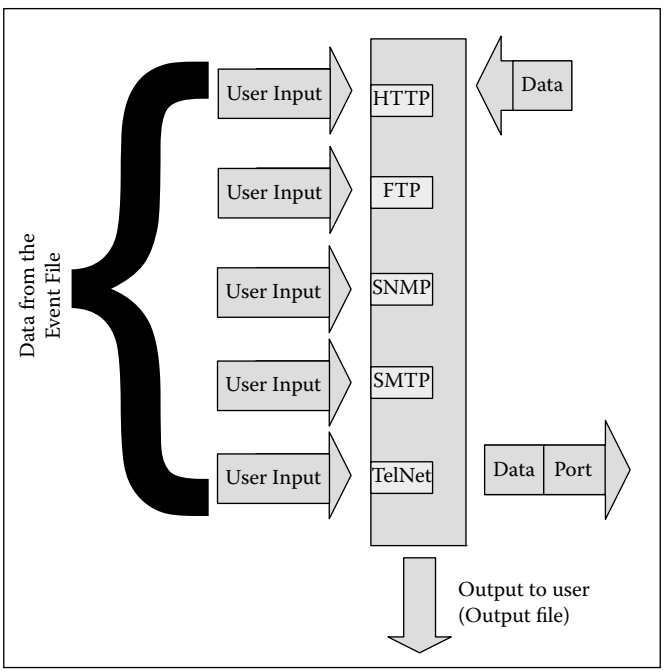


FIGURE 13.53 Application layer architecture.

```
INPUTS
00:00:10:00 infromHTTPuser 11   // data input on HTTP input port
00:00:16:00 infromHTTPuser 12
00:00:22:00 infromHTTPuser 13
00:00:28:00 infromHTTPuser 14
00:00:35:00 infromHTTPuser 15
---
00:02:50:00 infromSMTPuser 11
00:02:60:00 infromSMTPuser 12
00:03:00:00 infromSMTPuser 13
00:03:10:00 infromSMTPuser 14
00:03:20:00 infromSMTPuser 15
```

```
OUTPUTS
00:00:15:000 outtotransport 1180  //Application data sent on HTTP Port
00:00:21:000 outtotransport 1280
00:00:27:000 outtotransport 1380
00:00:33:000 outtotransport 1480
00:00:40:000 outtotransport 1580
---
00:02:55:000 outtotransport 1125   // Application data sent on Port 25
00:03:05:000 outtotransport 1325
00:03:15:000 outtotransport 1425
00:03:25:000 outtotransport 1525
```

FIGURE 13.54 Application input/output.

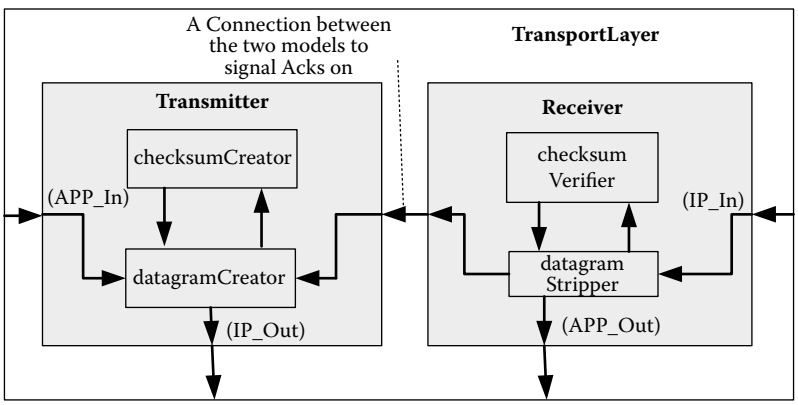


FIGURE 13.55 TCP coupled model structure.

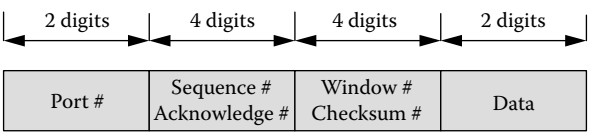


FIGURE 13.56 TCP packet format.

The creation of such packets is split between two atomic models: *datagramCreator* and *checksumCreator.* The data received (from the application layer) is routed to the *datagramCreator,* which will create an initial packet and forward it to the *checksumCreator* to compute a checksum. Then the completed packet will be forwarded to the *datagramCreator* and sent to the next layer in the protocol stack (through the *IP_Out* port). Before the packet is sent, a copy is saved to

```
checksumCreator::checksumCreator( const string &name ): Atomic( name )
, in( addInputPort( "in" ) ) , checksumCreatorOut( addOutputPort( "checksumCreatorOut" ) ) )
                             , preparationTime( 0, 0, 5, 0 ) {
}

Model &checksumCreator::externalFunction( const ExternalMessage &msg ) {
       if( msg.port() == in ) value = msg.value();
       holdIn( active, preparationTime );
}

Model &checksumCreator::internalFunction( const InternalMessage & ) {
       passivate();
}

Model &checksumCreator::outputFunction( const InternalMessage &msg ) {
       double result = checksum(value);
       sendOutput( msg.time(), checksumCreatorOut, result );
       return *this;
}

double checksumCreator::checksum (double data) {
       /*
              ddssaawwccccpp
              1100000000000
                22000000000
                  330000000
                    4400000
                      555X0
                         88
              1122334455588          */
       int dd = data/1E11;
       int ss = (data-(dd*1e11))/1E9;
       int aa = (data-dd*1e11-ss*1E9)/1E7;
       int ww = (data-dd*1e11-ss*1E9-aa*1E7)/1E5;
       int ccc = (data-dd*1e11-ss*1E9-aa*1E7-ww*1e5)/100;
              // since we are creating the checksum this should be zero.
       int pp = (data-dd*1e11-ss*1E9-aa*1E7-ww*1e5-ccc*100);
       // after all field are isolated.
       ccc = dd+ss+aa+ww+pp;
       double result =(dd*1E11)+(ss*1E9)+(aa*1E7)+(ww*1E5)+(ccc*1e2)+pp;
       return result;
}
```

FIGURE 13.57 *checksumCreator* model.

accommodate the connection manager, which will resend packets in case they are not received. For instance, the model *checksumCreator* is defined as shown in Figure 13.57.

As we can see, when we receive a packet, and the checksum method adds redundancy check information. We first isolate the different packet components. After all of them are isolated, we use the cyclic redundancy check (CRC) algorithm to compute the checksum. On the receiver side, we use two atomic models: a *datagramStripper* and a *checksumValidator*. The *datagramStripper* receives the data from the network layer. The *checksumValidator* checks the checksum field of the packet. If it is valid, then the *datagramStripper* is notified that the packet is not corrupted. Then *datagramStripper* checks the packet type to see if it is data or an acknowledgment. In the case of data, the packet headers are stripped, and the data is forwarded to the application layer. The *datagramStripper* also requests the *datagramCreator* to send an ACK to the packet source. On the other hand, if it is an ACK (data field is zero), the datagram stripper forwards the message to the *datagramCreator* to check whether the ACK is expected in order to delete the saved packet or resend it. If the checksum is incorrect, the packet is simply discarded.

Figure 13.58 shows the execution log of the transport layer model, in which we present the data being manipulated by its various models. Data output from the application layer are received by the

```
X / 00:00:10:000 / Root / infromapplication /   1280 to top
X / 00:00:10:000 / top / in /   1280 to datagramcreator1
D / 00:00:10:000 / datagramcreator1 / 00:00:00:005 to top
* / 00:00:10:005 / top to datagramcreator1
Y / 00:00:10:005 / datagramcreator1 / gocheck / 1200000000080 to top
D / 00:00:10:005 / datagramcreator1 / ... to top
X / 00:00:10:005 / top / in / 1200000000080 to checksumcreator1
D / 00:00:10:005 / checksumcreator1 / 00:00:00:005 to top
* / 00:00:10:010 / top to checksumcreator1
Y / 00:00:10:010 / checksumcreator1/checksumcreatorout/120000000 9280 to top
D / 00:00:10:010 / checksumcreator1 / ... to top
X / 00:00:10:010 / top / checkin / 1200000009280 to datagramcreator1
D / 00:00:10:010 / datagramcreator1 / 00:00:00:005 to top
* / 00:00:10:015 / top to datagramcreator1
Y / 00:00:10:015 / datagramcreator1/datagramcreatorout/1200000009280 to top
```

FIGURE 13.58 Transport layer log file.

transport layer, using the input: `00:00:10:00 infromApplication 1280`, which was generated in Figure 13.54.

The first event (`X/00:00:10:000`) is an input carrying the value 12 through the HTTP port 80. This is transmitted to *datagramCreator,* which executes the external transition function (adding the window size, acknowledgment, and sequence number to the data; for testing purposes, these values are zero). Then it schedules an internal transition (`D`) in 5 ms (reflecting the delay of the circuit). When this time is consumed, an internal transition (`*`) is fired. The first step involves executing the output function (`Y`), which transmits the packet through the *gocheck* port. The model then passivates (as discussed earlier "…" represents time = ∞). This event is converted into an input (`X`) for *checksumCreator,* which receives the application data and computes the checksum (also taking 5 ms). The *checksum-Creator* responds at time `00:00:10:010` with the same packet sent to it, with the addition of the checksum values (highlighted). Once the checksum is computed, the data are returned back to the *datagramCreator* through the *checkin* port (highlighted), where they are sent to the network layer through the *datagramCreatorOut* port at `00:00:10:015`.

13.6.4.3 The Network Layer

The third layer in the TCP/IP stack is the network layer, which is responsible for end-to-end communication through the network. The network layer is usually where most of the delay and stochastic operation occur. The layer adds source and destination IP fields to the packet to enable routing and creating subnets, local networks, and many network artifacts, as shown in Figure 13.59.

The headers for the Internet protocol are based on RFC (Request For Comments) no. 791 [26]. They contain the full addressing information (source and destination IP) as well as other quality-of-service parameters such as Time To Live (TTL), identification, and a checksum. The traffic packets are made of the source address, the destination address, and the TCP field. The options in each field are chosen from the IPV4 packet format. As seen in Figure 13.52, the network model consists of a transmitter and a receiver, which add or extract the corresponding information using the header format just discussed. Figure 13.60 shows an input example for this model.

The network layer is divided into two coupled models: a receiver and a transmitter (Figure 13.61). The transmitter receives data from the transport layer, and it is converted to the format illustrated in Figure 13.59. The transmitter also saves the destination IP in case of a resend request. The receiver is a coupled model that receives data from the data link layer, removes all IP headers associated with the packet, and forwards it to the transport layer.

The information sent to the network layer is used to create a *checksum* value, which is used to verify the data sent over the network. The model outputs the required four fields, as shown in Figure 13.62.

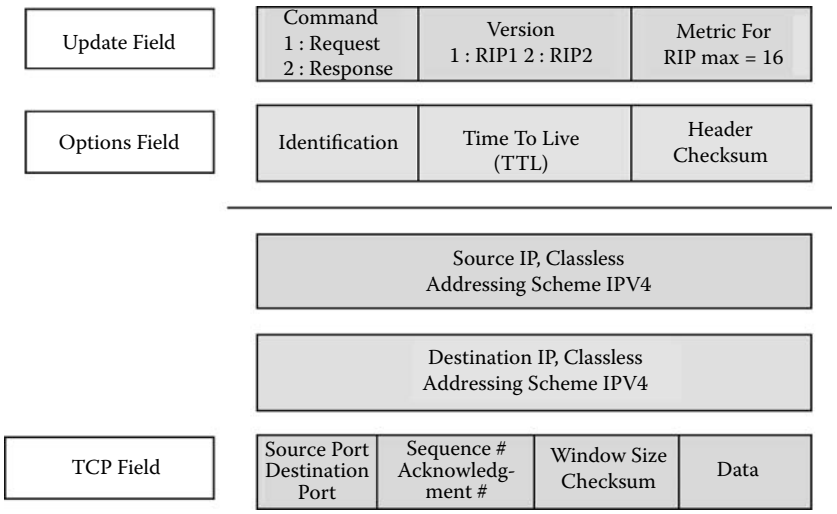


FIGURE 13.59 Header format.

```
00:00:00:010 infromTransport 1122334455580  // data to send
00:00:00:020 DestinationIP 192168111223     // destination IP value
```

FIGURE 13.60 IP test values.

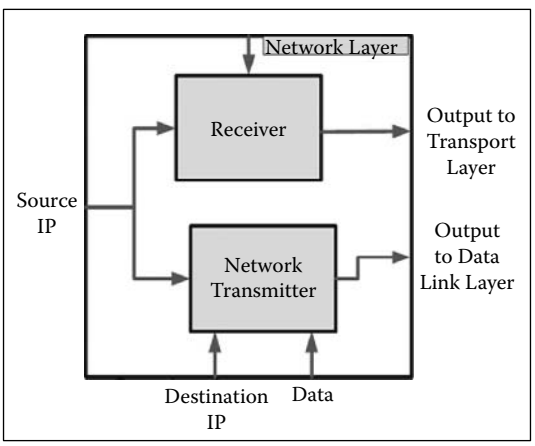


FIGURE 13.61 IP coupled model structure.

13.6.4.4 The Data Link Layer (DLL)

Modeling the data link requires dividing it into two parts. The first is to code the CRC operations of the logical link control (LLC) sublayer, which is the upper layer of the DLL (in charge of multiplexing and demultiplexing, flow control, etc.). The second is to model the carrier sense multiple access with collision detection (CSMA/CD) algorithm in the media access control (MAC) sublayer. The MAC layer is an interface between the LLC and the physical layer that provides channel access control (allowing multiple nodes to communicate in a multipoint network). All of these parts are combined into one atomic model called *dataLink*.

```
Y / 00:00:13:020 / networktransmitter1 / out / 485000015500 to top
Y / 00:00:13:020 / networktransmitter1 / out / 192168116224 to top
Y / 00:00:13:020 / networktransmitter1 / out / 192168116224 to top
Y / 00:00:13:020 / networktransmitter1 / out / 12223334318080 to top
```

FIGURE 13.62 Network layer log file.

FIGURE 13.63 Frame message format.

The CRC operations involve calculating the frame check sequence before sending a frame and detecting for errors when a frame is received. These operations are implemented similarly to the representation of the CRC-16, as shown earlier in Figure 13.57. The format of the frame sent and received over the physical layer is shown in Figure 13.63.

When a packet is received from a higher-level protocol (such as the *networkTransmitter* in the host), the CRC function appends a Frame Correction Sum (FCS) field into the packet in order to create a frame, and those ready to be sent are pushed into a queue. Before transmitting a frame, the *dataLink* senses the carrier by sending a *senseCarrier* port message to the physical layer and waits for a response. Figure 13.64 shows the model definition for this layer.

Eventually, the physical layer sends its status, which could be *idle, busy, jammed,* or *collision*. If the carrier is busy, *dataLink* sends another *senseCarrier* message and waits for another response (repeating it until the carrier is idle) and then outputs the frames in the queue. However, after every frame sent, the *dataLink* sends a *senseCarrier* message to the physical layer to ensure that there is no collision. If there was a collision, the *dataLink* sends a jamming signal to the physical layer via the *senseCarrier* port and waits for a response from the carrier. The carrier responds by sending a jamming signal to all connected devices. Upon receiving this response, the device that had its frames lost will resend the frame that was stored in the queue after a random delay of 0–10 ms. In contrast, if there was no collision after the frame was sent, the frame stored in the queue is deleted.

Figure 13.65 shows the execution of this model. The inputs presented show the reaction of the *dataLink* model when we send a frame with no errors and an IP packet. The output file displays the packet that was part of the frame received. The output file also displays the frame created when the packet was received at 20 s.

13.6.4.5 Simulation Results

In this section, we show the execution results of an example integrating all the components in the library. In Figure 13.66, data are received through the same set of layers in the reverse direction, with each layer stripping the extra variables added by its counterpart. Figures 13.66–13.68 show the results of one of the examples for a host whose source IP address is `111222333`. The test in Figure 13.66 shows the transmission of FTP data from the host to another end on the network (simple data values were chosen in order to ease the reviewing of the results). Figure 13.67 shows the host's reaction to these events.

Figure 13.68 shows the data link execution upon reception of two events: the first represents the host sending the data received throughout the network (after adding the appropriate headers), forwarding it to the data link layer. We can also see that the data link layer has actually stored the data until it checks the physical layer. As the response arrives from the physical layer, data are sent to the other host.

```
Model &dataLink::externalFunction( const ExternalMessage &msg ) {

// A packet has been received from the network layer
if( msg.port() == getPacket ) {
      switch(pcount++) {         //to synchronize the field received of the packet
      case 0: other.push_back(msg.value());       break; // other field of the packet received

      case 1: destination.push_back(msg.value()); break; // IP destination address
      case 2: source.push_back(msg.value());      break; // IP source address
      case 3: data.push_back(msg.value());   // data field of the IP packet
              pcount=0;
              sense0=true;                   //send a sensing signal
              break;
      default: pcount=0;
      }
      holdIn( active, preparationTime);  return *this;
}

// A frame has been received from another device via physical layer
if( msg.port() == getFrame)   {
   switch(fcount++) {
      case 0:  temp.other = msg.value(); break; //other fields in packet section of the
               frame
      case 1:  temp.destination = msg.value(); break; //destination address
               ...
      case 4:  //FCS field of the frame
               temp.fcs = msg.value();
               fcount=0;
               if(CRC((temp.destination+temp.other+temp.data+temp.source),temp.fcs))
                   packetOut=true;    //check if the frame has any errors by using CRC algorithm

      }
    holdIn( active, preparationTime);  return *this;
}

if( msg.port() == status)     {   // physical layer sends the status of the carrier
      if(msg.value()==1)                     //Carrier is idle
           if(collision)  {   //check if there was a collision that happened before
                 frameOut=sense0=true;                     //resend previous frame
                 collision=false;
           }
      else {
               if(frameSent) {     //last frame successfully sent
                   other.pop_front();
                   destination.pop_front();
                   source.pop_front();
                   data.pop_front();
                   frameSent=false;
               }
               f(other.size()!=0)       //sense carrier before sending the next frame
                    frameOut=sense0=frameSent=true;
            }
}
else {
     if(msg.value()==-1)    //there is a jam in carrier
        { if(other.size()!=0) jamming=true; }   //check if there is information to send
     else if(msg.value()==0) busy=true;   //Carrier is busy
          else if(msg.value()==2) collision=sense1=true;   // Collision: jam physical layer

     }
  holdIn( active, preparationTime);  return *this;
}
else
    passivate();
}

Model &dataLink::internalFunction( const InternalMessage & msg) {

   if(busy)   {  //wait for five milliseconds approximately for 1 frame to be sent in carrier

               Time adder2(0,0,0,5);         //then check the status of the carrier
```

FIGURE 13.64 DLL model.

```
                sendOutput((msg.time()+adder2), senseCarrier, 0);
                busy=false;
    }
    passivate();
}

Model &dataLink::outputFunction( const InternalMessage &msg ) {

  if(frameOut)   {            //send frame
   sendOutput( msg.time(), sendFrame, other.front());
   sendOutput( msg.time(), sendFrame, destination.front());
   sendOutput( msg.time(), sendFrame, source.front());
   sendOutput( msg.time(), sendFrame, data.front());
   sendOutput( msg.time(), sendFrame,
   FCS(source.front()+destination.front()+data.front()+other.front()));
   frameOut=false;
  }

  if(sense0) {  //sense the carrier before transmission of data. a value of 0 means sensing

   sendOutput( msg.time() , senseCarrier, 0);
   sense0=false;
  }

  if(packetOut) {             //send packet to internet layer
     sendOutput( msg.time(), sendPacket, temp.other);
     sendOutput( msg.time(), sendPacket, temp.destination);
     sendOutput( msg.time(), sendPacket, temp.source);
     sendOutput( msg.time(), sendPacket, temp.data);
     packetOut=false;
  }

  if(jamming) {    //sense carrier again after random delay to resend frame
       x = (int)other.front()%10;           //create random delay from 0 to 10 msec
       Time adder(0,0,0,x);
       sendOutput((msg.time()+adder), senseCarrier, 0);         //sense carrier again
       jamming=false;
  }

  if(sense1) sendOutput( msg.time() , senseCarrier, -1);   //send a jamming signal to the carrier


}
```

FIGURE 13.64 (continued).

00:00:10:00 getFrame 101	00:00:10:000 sendPacket 101
00:00:10:00 getFrame 102	00:00:10:000 sendPacket 102
00:00:10:00 getFrame 103	00:00:10:000 sendPacket 103
00:00:10:00 getFrame 104	00:00:10:000 sendPacket 104
00:00:10:00 getFrame 410	00:00:20:000 sendFrame 201
00:00:20:00 packetIn 201	00:00:20:000 sendFrame 202
00:00:20:00 packetIn 202	00:00:20:000 sendFrame 203
00:00:20:00 packetIn 203	00:00:20:000 sendFrame 204
00:00:20:00 packetIn 204	00:00:20:000 sendFrame 810

FIGURE 13.65 Data link layer inputs/outputs.

13.6.5 Router

A router is a device used to interconnect networking devices. In general, routers contain buffers to store packets to be transmitted and a routing table to decide where to route them upon reception. Routing is a function of layer 3, and it is in charge of distributing packets across an internetwork. The model presented here, introduced in Ahmed et al. [25], uses an abstract look in the routing

```
00:10:00 FTP_In 11
00:10:00 Destination 192168111
00:10:01 statusCarrier 1
00:40:02 FTP_In 1001214
00:40:02 Destination 192168001
00:40:03 statusCarrier 1
00:80:04 FTP_In 1001215
00:80:04 Destination 192168001
00:80:06 statusCarrier 1
01:90:07 Telnet_In 1001216
01:90:07 Destination 192168001
01:90:11 statusCarrier 1
```

FIGURE 13.66 Simulation execution test.

```
Y / 00:00:49:010 / networktransmitter1 / out / 2000000000 to top
Y / 00:00:49:010 / networktransmitter1 / out / 111222333 to top
Y / 00:00:49:010 / networktransmitter1 / out / 0 to top
Y / 00:00:49:010 / networktransmitter1 / out / 0 to top
D / 00:00:49:010 / networktransmitter1 / ... to top
Y / 00:00:49:010 / top / outtodatalink / 2000000000 to Root(00)
Y / 00:00:49:010 / top / outtodatalink / 111222333 to Root(00)
Y / 00:00:49:010 / top / outtodatalink / 0 to Root(00)
Y / 00:00:49:010 / top / outtodatalink / 0 to Root(00)
```

FIGURE 13.67 Host log file section.

```
Y / 00:00:06:000 / internet / outtodatalink / 20000 to top
Y / 00:00:06:000 / internet / outtodatalink / 192168116224 to top
Y / 00:00:06:000 / internet / outtodatalink / 0 to top
Y / 00:00:06:000 / internet / outtodatalink / 0 to top
D / 00:00:06:000 / internet / ... to top
X / 00:00:06:000 / top / getpacket / 20000 to datalink
X / 00:00:06:000 / top / getpacket / 192168116224 to datalink
X / 00:00:06:000 / top / getpacket / 0 to datalink
X / 00:00:06:000 / top / getpacket / 0 to datalink
```

FIGURE 13.68 Data link interaction.

process, considering three main functionalities: receiving/forwarding traffic, processing IP packets, and maintaining a routing table, as seen in Figure 13.69.

In order to simulate these functions, two models were created: the *RouterInterface* and the *RouterProcessor,* which, in turn, are composed of the *ProcessingUnit* and a *ripTable*. Every router has a number of interfacing cards to receive or forward traffic from or to the network. The *RouterInterface* model receives and sends packets with the format discussed in Figure 13.59. After packets are received by the *RouterInterface,* they are processed to see if they are messages to the router (requests or updates) or just data packets to be forwarded to their destinations. The *ProcessingUnit* is responsible for reading in the packets from the interfaces, processing them, and making routing decisions regarding their destinations. Upon receiving a packet, it looks at the packet's header, extracts from it the TTL value, and checks whether it is valid. In this case, it will read the packet's type, and it will react according to the type.

Three types of packets are accepted: *request, respond,* and *data*. *Request* packets carry the destination address of the requesting router that wants the update, following the routing information protocol [27]. This address value is extracted from the packet's header, and it is sent along with the requesting router reply information to the *ripTable* model (so proper reply information can be prepared and sent to the requesting router). The *respond* packets carry a network address and a metric

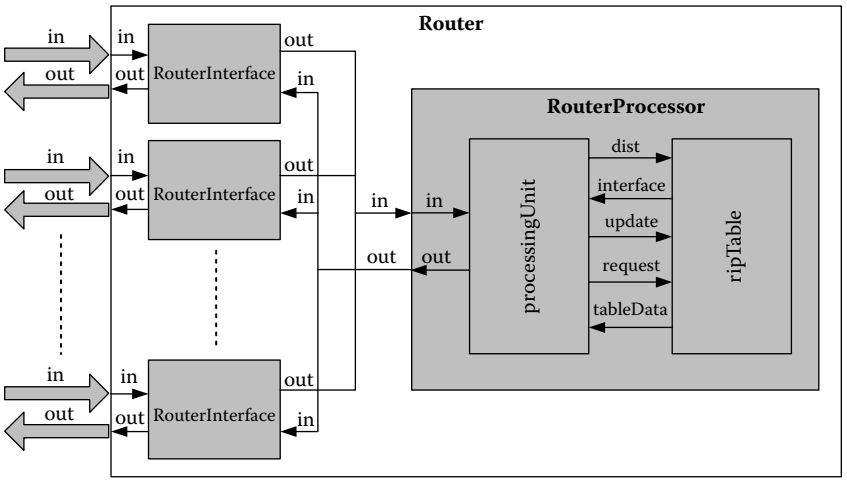


FIGURE 13.69 *Router* coupled module.

value (cost) associated with that route. These packets are used to update other routers or to respond to other routers' requests for updates. The router extracts both the address and the metric, and it forwards this information, along with the sending router's data, to the *ripTable* model. When a *data* packet is received, the *processingUnit* extracts its destination address and forwards it to the *ripTable* (which maintains the routing information for forwarding packets). Once the *ripTable* returns the value of the output interface chosen, the *RouterProcessor* will forward the data packet through it. If the destination address is not found in the routing table, the value 0 is returned (and a request packet is issued through all interfaces, except the one that the packet was received through, requesting an update on that destination). Figure 13.70 shows the definition of the *ripTable* atomic model.

The *ripTable* model is in charge of maintaining the routing information for the router. The entries in the table have the format <*Address, Metric, Interface*>, where *Address* is a destination for the packet, *Metric* represents the cost of getting to that destination, and the output *Interface* is the one through which the router must forward the packet to be at least one hop closer to the destination. The *ripTable* can receive different events: *update request* and request for forwarding information (*Address*). In the case of *updates,* the model will be receiving the address that the update is about, together with a new metric value. The address is searched in the RIP table. If the address does not exist, the information will be added. Otherwise, the metric in the table is compared with the newly received one, and if the new value is smaller than the one in the table, the output interface number is replaced by the new update. For *request* events, the model prepares the required information (the whole table) and redirects it as it responds. Finally, for the *Address* request, the model will search its table for the destination address and send the output interface that should be used to forward the packet. Figure 13.71 shows different testing scenarios for the model subcomponents.

The events injected represent update, request, and destination values. We first send four updates, simulating the values that will be passed from the *RouterProcessor* to the *ripTable* (and the table did, in fact, receive the messages and stored them). For instance, the first message will activate the external transition function, and the value 1.1 will be stored on *hostemp.address.* When the second message arrives, it will be stored in *hostemp.metric* and the third message on *hostemp.gatewayIP* (that is, a value of five). At this point, the host's table is empty. Therefore, we initialize the *hosts[0]* entry with the temporary data just stored, and *hostsize* is now 1. The *senddone* flag is set. Thus, at this time, a *done* message is generated. The same occurs with the next messages. At 100, we request an interface for address 1.3, and we can see that the model did output the right interface number associated with that address. When the address message arrives, we look for the value 1.3 on each entry. It is found, the *sendPort* flag is set, and the output function sends the *temp* value found on the

```
Model &ripTable2::externalFunction( const ExternalMessage &msg) {

     if ( msg.port() == Address )     {     //input address of an intended destination
        for(temp=0,i=0;i<hostSize;i++)        //find port number of the destination host
               if(hosts[i].address==msg.value()) break;
        if(i != hostSize) temp=hosts[i].gatewayIP;   // destination is in the table

        sendPort=true;      //send the next interface
        holdIn (active, preparationTime);   return *this;
     }

     if( msg.port() == update )  { // update input to for the table
        if(receivecnt++ == 0)  //receive destination address first. save temporarily
          hostTemp.address=msg.value();
        else if(receivecnt++==1)   //second: receive metric of desired destination
          hostTemp.metric=msg.value();
          else if(receivecnt==2)  {       //last: interface prt of the destination
           hostTemp.gatewayIP = msg.value();
           i = receivecnt = 0;     // reset message sequence and counter
           sendDone=true;

           while(i<hostSize)  //find the destination if it is in the table already
                 if(hosts[i++].address==hostTemp.address) break;

                   //if destination not in table, add the new destination and its
                     information
           if(i == hostSize) {
                hosts[hostSize].address   = hostTemp.address;
                hosts[hostSize].metric    = hostTemp.metric;
                hosts[hostSize].gatewayIP = hostTemp.gatewayIP;
                hostSize++;  // ->increment table size
             }
           else
             // destination already exists. Check if new metric cost < metric stored
              if(hosts[i].metric > hostTemp.metric) hosts[i].metric=hostTemp.metric;

           holdIn (active, preparationTime);   return *this;
         }  // if receivecnt==2
     } // port == update

     if( msg.port() == request )   {      //a request message to output table information

        temp=msg.value(); // -> store temporarily the address of the requestor
        sendTable=true;
        holdIn (active, preparationTime); return *this;
     }
     passivate();      //unknown input port will passivate the model
}


Model &ripTable2::internalFunction ( const InternalMessage &)  {
     passivate();
}

Model &ripTable2::outputFunction( const InternalMessage &msg) {

     if(sendPort)      {      //send interface port of the desired destination
        sendOutput(msg.time(), interfacePort, temp);
        sendPort=false;
     }

     if(sendDone) {    //send an acknowledgement that the table has been updated
        sendOutput(msg.time(), done, 0);
        sendDone=false;
     }

     if(sendTable)     {     //send the RIP table information in response to a request
        for(int i=0;i<hostSize;i++) {
          sendOutput(msg.time(), respond, (commandVersion + hosts[i].metric) );
          sendOutput(msg.time(), respond, hosts[i].address);
          sendOutput(msg.time(), respond, 0 );
```

FIGURE 13.70 *RipTable* atomic model.

```
                    sendOutput(msg.time(), respond, 0 );
                    sendOutput(msg.time(), respond, temp );
                  }
                  sendTable=false;
                }
              }
```

FIGURE 13.70 (continued).

```
INPUTS
00:00:00:010 update 1.1          //sending update data
00:00:00:010 update 1            //metric 1
00:00:00:010 update 5            //interface 5
00:00:00:011 update 1.2          //sending update data
00:00:00:011 update 2            //metric 2
00:00:00:011 update 6            //interface 6
00:00:00:012 update 1.3          //sending update data
00:00:00:012 update 3            //metric 3
00:00:00:012 update 7            //interface 7
00:00:00:013 update 1.4          //sending update data
00:00:00:013 update 4            //metric 4
00:00:00:013 update 8            //interface 8
00:00:00:100 address 1.3         //requesting interface for address 1.3
00:00:00:110 address 1.5         //requesting interface for address 1.5
00:00:00:120 request 1           // request data, address 0 (all table)
00:00:00:120 request 0
00:00:01:010 update 1.3          //update data (address 1.3)
00:00:01:010 update 1            //metric 1
00:00:01:010 update 3            //interface 3

OUTPUTS
00:00:00:101 out_interface 7      //out interface 7
00:00:00:111 out_interface 0      //out interface 0 (unknown)
00:00:00:121 out 222255202002    //start of response messages. 1st entry (option)
00:00:00:121 out 1.1              //(address)
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1                // (interface to respond through)
00:00:00:121 out 222255202002  // 2nd table entry (option filed)
00:00:00:121 out 1.2              //(address)
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1
00:00:00:121 out 222255202002
00:00:00:121 out 1.3
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1
00:00:00:121 out 222255202002
00:00:00:121 out 1.4
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1
00:00:00:121 done 0              //responding completed
```

FIGURE 13.71 *RipTable* simulation.

destination field (in this case, seven) through the *out_interface* port. We then request the address 1.5, which does not exist in the table; therefore, the model sends a value of zero for the output interface. We then send a request with the address value zero. The model responds with all of the table entries to that interface port, followed by a *done* signal to the router processor. Finally, the model accepts updates for an existing address in its table, replacing the output interface associated with that address.

```
INPUTS
00:00:010 in1 2000001  // update with metric 1
00:00:010 in1 111101101  // address
...
00:00:100 in1 3010012 // data, ttl=10, CRC=12
00:00:100 in1 121117001 // source address
00:00:100 in1 133303303 // destination address
00:00:100 in1 15
00:01:010 in1 2000000   // update metric 0
00:01:010 in1 133303303
...
00:02:000 in1 3008011 // data,ttl=8, CRC = 11
00:02:000 in1 114124201
00:02:000 in1 123456789 // unknown destination 00:02:010 in2 2000007   // update metric 7
00:02:010 in2 122202202
00:02:010 in1 3000007  // data, TTL = 0
00:02:010 in1 122202202

OUTPUTS
00:00:018 out2 2000001  // update
00:00:018 out2 111101101  // address
...
00:00:109 out2 3010012 // data forward
00:00:109 out2 121117001
00:00:109 out2 133303303
00:00:109 out2 15
00:01:018 out2 2000000   // update
00:01:018 out2 133303303
...
00:02:009 out2 1000000// request
00:02:009 out2 123456789
```

FIGURE 13.72 Router input/output events.

Figure 13.72 shows a complete simulation for the router model. The first packet is an update. The router passes the related values to its table and the table is updated. The message arrived at the router from interface 1, and a corresponding update message was created and sent through interface 2 (*out2*). For every update packet, an update to the neighbor nodes is sent through the other router interface. Then we show a packet representing data injected into the router. The packet option field shows a TTL value of 10. The router knew the address because it received an update on it before. The router forwards the packet using the right output interface. Afterward, another update with a smaller metric for an address that the router has in its table is sent through interface 1. We can see that the router did update its table with the better metric value and sent an update through interface 2.

No output was sent in response to the last two packets. The reason is that the first one was an update with a metric higher than the existing one in the routing table. The second was a data packet with a TTL value of zero (expired). In both cases, the router discarded the packets (Figure 13.73).

In Figure 13.74, we show the behavior of the model when we simulate a signal coming from the *RouterProcessor* queue, considering packets sent from the different interfaces. Packets with different types were fed to *PacketProcessor,* and the outputs were analyzed, as shown in Figure 13.74.

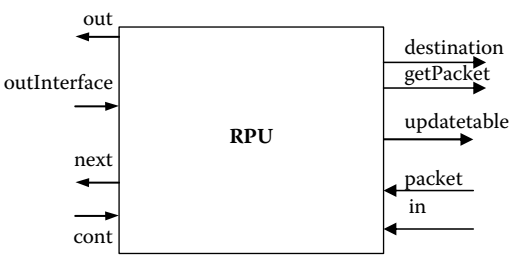


FIGURE 13.73 RPU input/output ports.

```
INPUT
00:00:00:001  in  1              // send a flag signal
00:00:00:010  packet  3000000001 // send a data packet
00:00:00:010  packet  1.2
00:00:00:010  packet  1.3
00:00:00:010  packet  1.4
00:00:00:050  outInterface  2   // send output interface (as the table responds)
00:00:01:001  in  2              // send a flag signal
00:00:01:010  packet  3000000002// send 2nd data packet
00:00:01:010  packet  2.2
00:00:01:010  packet  2.3
00:00:01:010  packet  2.4
00:00:01:050  outInterface  0   // table responds, address not found
00:00:02:001  in  3                  // send a flag signal
00:00:03:010  packet  2000000005     // send an update packet
00:00:03:010  packet  3.2
00:00:03:010  packet  0
00:00:03:010  packet  0
00:00:03:050  cont  0                // confirmation from ripTable
00:00:04:001  in  4                  // send a flag signal
00:00:04:010  packet  1000000000     // send a request signal
00:00:04:010  packet  4.2
00:00:04:010  packet  0
00:00:04:010  packet  0
00:00:04:050  cont  0                // confirmation from ripTable

OUTPUT
00:00:00:001  getpacket  1       //requesting a packet from interface 1
00:00:00:030  destination  1.3   //requesting output interface for destination
00:00:00:070  out  3000000001    //forwarding packet through interface
00:00:00:070  out  1.2
00:00:00:070  out  1.3
00:00:00:070  out  1.4
00:00:00:070  out  2
00:00:00:070  next  0           //request next flag
00:00:01:001  getpacket  2      //requesting a packet from interface 2
00:00:01:030  destination  2.3  //requesting output interface for destination
00:00:01:070  out  1000000000
00:00:01:070  out  2.3
00:00:01:070  out  0
00:00:01:070  out  0
00:00:01:070  out  -2
00:00:01:070  next  0            // request next flag
00:00:02:001  getpacket  3       //requesting a packet from interface 3
00:00:03:030  updatetable  3.2   // sending update information to table (address)
00:00:03:030  updatetable  5     // (metric)
00:00:03:030  updatetable  3     // (interface)
00:00:03:050  next  0            // request next flag
00:00:04:001  getpacket  4       //requesting a packet from interface 4
00:00:04:030  requ  4            //forward request info to table (interface)
00:00:04:030  requ  4.2           // (address)
00:00:04:050  next  0            // request next flag
```

FIGURE 13.74 *PacketProcessor* model execution.

The messages in the figure show the process of sending flag signals from the *queue* to *PacketProcessor* and then responding to the processor's requests for packets. The model requested the output interface every time it received a data packet—as in the first two packets sent. The model used the interface ID to forward the packet and issued a request to be updated when the interface value received was zero. Using the event file, we also simulated an update packet (the third packet) and a request packet (the fourth packet); for both types, the processor output the right messages to the *ripTable* model.

13.7 MODELING MOBILE AD HOC NETWORKS (MANETS)

Wireless networks use radio or electromagnetic waves as the physical layer within a networked environment. Modeling mobile ad hoc networks (MANets) are self-configuring networks (i.e., every node in the system can become a router) with varying topology. In this section, we will show how to use CD++ to build models to test routing algorithms for MANets, as discussed in Farooq, Wainer, and Balya [28]. We will present a model, found in *./AD-HOCRouting.zip,* that implements the ad hoc on-demand distance vector (AODV) algorithm [29]. This on-demand algorithm was one of the first ad hoc routing algorithms chosen by the Internet Engineering Task Force as an experimental RFC standard. AODV has low processing and memory overhead and offers quick adaptation to dynamic link conditions.

AODV assumes bidirectional links and creates routes on demand. Our model considers a network plane in which nodes are spread at random. Data movement between two cells represents one hop, as routing takes into account the shortest hop count. Each node can communicate with the nodes to the N, S, E, and W. However, if each neighbor is not a node, we have a *dead cell* (representing physical obstacles or simply the absence of a link). Two nodes with a dead cell between them cannot communicate directly.

If we assume that the communication cost between any two nodes is the same, modeling AODV using Cell-DEVS involves finding the shortest path between two nodes. To do so, we used a variant of the classical Lee's algorithm [30]. Figure 13.75 shows a simple example of a network plane. Here, S represents a sender node and D a destination node; black cells represent dead cells. In order to find a route from S to D, the node S broadcasts an RREQ message to all its neighbors (called the *wave* nodes). The wave nodes rebroadcast the message to their neighbors and set up a reverse path to the sender. These nodes further rebroadcast this message and set up a reverse path to the nodes from which they received the message.

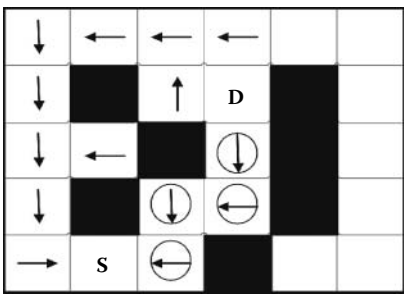


FIGURE 13.75 AODV routing in Cell-DEVS.

This process continues until the message reaches the destination node D. Because there is more than one path, D may receive multiple RREQ messages for the same sender. However, the first route through which D receives the RREQ message is the shortest path between sender and destination. The destination thus ignores all RREQ messages except for the first one, and it replies by sending an RREP message using the reverse path. All the wave nodes on this route become *path* nodes (represented with circles containing arrows in the figure). All other wave nodes move to a *clear* state (not shown in the figure). Using these ideas, we built a model in CD++, which is presented in Figure 13.76.

The model uses the following values to represent the cell's states:

S = 0: **dead** (dead cell);
1: **init** (initial state of the nodes);
2: **initD** (initial state of the destination node);
3: **DR** (destination ready; state of the destination node after it has received a send request from the sender);
4: **InitS** (initial state of the sender node);
5: **WaveU** (wave up);
6: **WaveD** (wave down);
7: **WaveR** (wave right);
8: **WaveL** (wave left);
9: **PathU** (path up);

```
[path]
type : cell   dim : (15,15)   delay : transport    border : nowrapped
neighbors :    (-1,0)  (0,-1)  (0,0)  (0,1)   (1,0)
localtransition : path-rule

[path-rule]
rule :  3 100 { (0,0)=2 and stateCount(9)>0 }  ; DR <- (0)=InitD and stateCount(PathU)>0
rule :  3 100 { (0,0)=2 and stateCount(10)>0 }
rule :  3 100 { (0,0)=2 and stateCount(11)>0 }
rule :  3 100 { (0,0)=2 and stateCount(12)>0  }
rule :  5 100 { (0,0)=1 and (-1,0)>3 and (-1,0)<9 } ; WaveU <- (0,0)=Init &
                            DR<(-1,0)<PathU
rule :  6 100 { (0,0)=1 and (1,0)>3 and (1,0)<9 }
rule :  7 100 { (0,0)=1 and (0,1)>3 and (0,1)<9 }
rule :  8 100 { (0,0)=1 and (0,-1)>3 and (0,-1)<9 }
rule :  9 100 { (0,0)=5 and stateCount(2)=1 } ; PathU <- (0,0)=WaveU &
                            stateCount(InitD)=1
rule : 10 100 { (0,0)=6 and stateCount(2)=1 }
rule : 11 100 { (0,0)=7 and stateCount(2)=1 }
rule : 12 100 { (0,0)=8 and stateCount(2)=1 }
rule :  9 100 { (0,0)=5 and (0,-1)=11 }      ; PathU <- (0,0)=WaveU and (0,-1)= PathR
rule :  9 100 { (0,0)=5 and (0,1)=12 }
rule :  9 100 { (0,0)=5 and (1,0)=9 }
rule : 10 100 { (0,0)=6 and (0,-1)=11 }
rule : 10 100 { (0,0)=6 and (0,1)=12 }
rule : 10 100 { (0,0)=6 and (-1,0)=10 }
rule : 11 100 { (0,0)=7 and (0,-1)=11 }
rule : 11 100 { (0,0)=7 and (-1,0)=10 }
rule : 11 100 { (0,0)=7 and (1,0)=9 }
rule : 12 100 { (0,0)=8 and (0,1)=12 }
rule : 12 100 { (0,0)=8 and (-1,0)=10 }
rule : 12 100 { (0,0)=8 and (1,0)=9 }
rule : 13 100 { (0,0)=1 and stateCount(13)>0 }  ; clear <- (0,0)=Init and
                            stateCount(clear)>0
rule : 13 100 { (0,0)>4 and (0,0)<9 and stateCount(13)>0 }
rule : 13 100 { (0,0)>4 and (0,0)<9 and stateCount(3)>0 }
rule : 13 100 { (0,0)>4 and (0,0)<9 and stateCount(14)>0 }
rule : 13 100 { (0,0)>4 and (0,0)<9 and (-1,0)>8 and (-1,0)<13 and (-1,0)!= 10 }
rule : 13 100 { (0,0)>4 and (0,0)<9 and (1,0)>8 and (1,0)<13 and (1,0) != 9 }
rule : 13 100 { (0,0)>4 and (0,0)<9 and (0,-1)>8 and (0,-1)<13 and (0,-1)!= 11 }
rule : 13 100 { (0,0)>4 and (0,0)<9 and (0,1)>8 and (0,1)<13 and (0,1) != 12 }
rule : 14 100 { (0,0)=4 and stateCount(9)>0 }
rule : 14 100 { (0,0)=4 and stateCount(10)>0 }
rule : 14 100 { (0,0)=4 and stateCount(11)>0 }
rule : 14 100 { (0,0)=4 and stateCount(12)>0 }
```

FIGURE 13.76 Implementing AODV routing in CD++.

10: **PathD** (path down);
11: **PathR** (path right);
12: **PathL** (path left);
13: **Clear** (final state of the nodes that received a wave message but are not going to become a path node); and
14: **Found** (destination found; final state of the sender).

Each cell evolves through different states. The first rules are used to define when the destination is ready (i.e., the destination node has received an RREQ). In order to check this, we verify whether any of the four neighbors is a *path*. In that case, the search has finished and we have found the destination. The next set of rules is used to determine when a cell should become a wave cell (i.e., a cell neighboring the RREQ). To be converted into a wave, the cell should be in an *init* state, and one of its neighbors should be a *wave* (i.e., it should have a value between four and eight). According to the current value of the neighboring cell, the origin cell should become a wave pointing to the N, S, E, or W. The next step is to determine when a wave cell will become part of the path. This will happen if the cell is a wave and a neighbor in the right direction is in the path. For instance, the first rule in

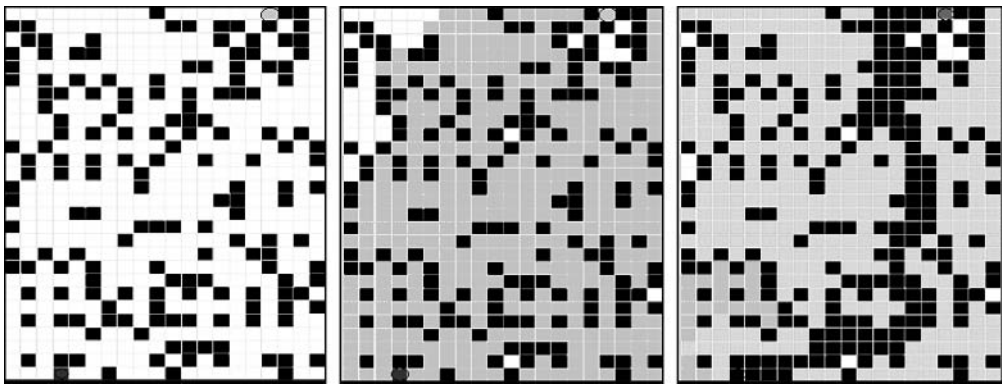


FIGURE 13.77 AODV simulation: (a) initial distribution; (b) state after 50 steps; (c) state after 100 steps.

Figure 13.76 says that if the current cell is a wave with direction N and the cell to the W is a path cell with direction E, then the current cell should become a path cell with direction N. The following rules determine how to change into a path cell with the right direction.

After the path has been set, we need to clear the unused cells. There are several conditions for clearing a cell:

1. The cell is in the *init* state and there are neighbors that have been cleared.
2. The cell is a *wave* and there is a neighbor that has been cleared that is the destination node, or that is in the *found* state (i.e., a source that has found the path to the destination).
3. The cell is a *wave* and there is a *path* node in the neighborhood, but not in the right direction (e.g., this is a wave to the W, and we have a path neighbor with direction to the N).

Finally, we define how to transform the source node into a *found* node. To do so, the cell should be in an initial state for a sender, and at least one of the neighbors should be a *path* node.

A number of tests were conducted on the model. For instance, Figure 13.77 shows a case with 20×28 cells (dead cells are in black and cells that have not received any message are white). Initially, the sender (shown in gray in the SW) broadcasts an RREQ message to the destination (shown in the top-right part of the figure). After 50 steps, we see the light gray nodes representing those nodes that have received an RREQ. The dark gray node (close to the source) carries an RREP (in accordance with the rules for defining path cells).

After 100 steps, a route has been established. Clear cells are represented in light gray. The final state after 106 steps shows the shortest route between the sender and the receiver, and all wave nodes end up in clear state.

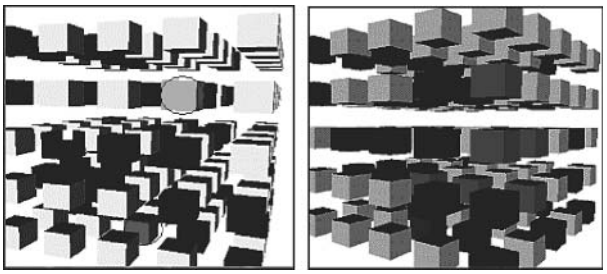
Exercise 13.16

Modify the source and destination position and repeat the global test.

Exercise 13.17

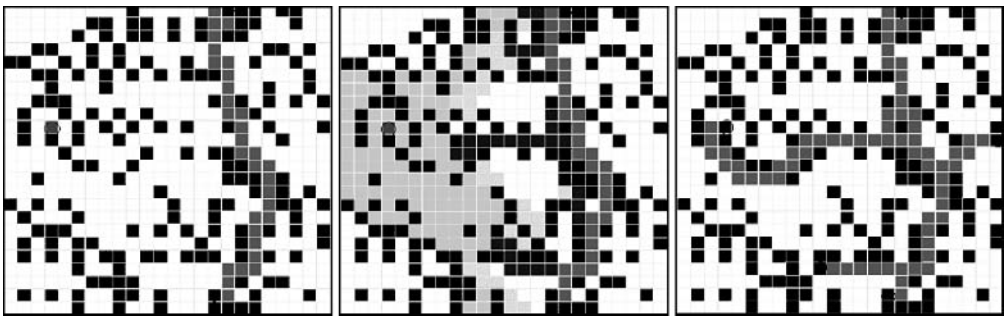
Modify the number and position of dead cells and repeat the tests. Discuss the results obtained.

We extended the preceding model to three dimensions (Figure 13.78), which can be used to model the transmission of messages in a three-dimensional ad hoc network; it can also be used, for example, to represent interworking. That is, if one of the planes represents a wireless ad hoc network and the other a wired network, it would make sense to transmit the data in the ad hoc plane to the wired plane through the nearest gateway because the cost in a wired network is generally less than in a wireless one.



(a) (b)

FIGURE 13.78 Three-dimensional AODV simulation: (a) initial state; (b) final state.



(a) (b) (c)

FIGURE 13.79 AODV multicast modeling: (a) 93 steps; (b) 125 steps; (c) final state after 217 steps.

The model also includes a few other extensions:

- Multicasting in AODV: The construction of multicast trees on *ad hoc* networks is complex, and Hochberger and Hoffmann [31] have shown that as the number of receiver or sender nodes increases, the number of states for each cell goes beyond practical limits, particularly if we consider optimality (i.e., the tree should duplicate data as little as possible). We proposed an algorithm based on this model, which constructs optimal multicast trees [28]. The nodes can join to broadcast by finding the shortest optimal route and constructing a multicast tree. The final state of the model after 217 steps of execution is shown in Figure 13.79(c): Three new nodes have been successfully added to a multicast tree.
- We have also defined models for routing among multiple pairs of senders and receivers. Lee's algorithm fails for multiple pairs of senders and receivers; it generates deadlocks and may prevent the generation of a routing path between pairs of nodes that can communicate [31]. To overcome this problem, we have exploited the inherent parallelism in Cell-DEVS and have found a solution to the problem: each pair of senders/receivers is allocated a plane in a three-dimensional Cell-DEVS. On each plane, we run a variant of Lee's algorithm that permits routing multiple pairs of senders and receivers without having to define more states. Because each pair is routed separately in each plane, routing messages for each pair do not interfere with each other. By avoiding this interference, we can successfully prevent the generation of deadlocks. Moreover, the approach exploits the inherent parallelism in Cell-DEVS as multiple pairs are routed simultaneously.
- Mobility behavior of the ad hoc nodes (nodes move diagonally and bounce back when they reach the edges of the plane; collision avoidance is also implemented): The model, found in *./MobileNode.zip,* contains both static and mobile nodes and one or more gateways. The coupled model presented in Figure 13.80 has 20 × 20 cells, and the surrounding 25 cells will form the neighborhood. The *mobilenode* model implements all the desired behavior:

```
[mobilenode]
type : cell    width: 20   length : 20       height : 3
delay : transport             border : nowrapped
neighbors : (-2,-2,0) (-2,-1,0) (-2,0,0) (-2,1,0) (-2,2,0) (-1,-2,0) (-1,-1,0) (-1,0,0)
            (-1,1,0)
...
(1,-2,-1) (1,-1,-1) (1,0,-1) (1,1,-1) (1,2,-1) (2,-2,-1) (2,-1,-1) (2,0,-1) (2,1,-1) (2,2,-1)


[mobility_CA-rule]
rule: 1 1000 {(0,0)=4 and ((-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2)}
rule: 4 1000 {(0,0)=1 and ((1,1) != 0 or (0,2)=3 or (2,2)=4 or (2,0)=2 )}
rule: 3 1000 {(0,0)=2 and ((-1,1) != 0 or (-2,0)=1 or (-2,2)=3 or (0,2)=4)}
rule: 2 1000 {(0,0)=3 and ((1,-1) != 0 or (2,-2)=2 or (2,0)=4 or (0,-2)=1)}

[coverage-rule]
rule: 7 1000 { statecount(6) > 0 }
rule: 7 1000 { statecount(10) > 0 }
rule: 7 1000 { statecount(20) > 0 }
rule: 7 1000 { statecount(30) > 0 }
rule: 7 1000 { statecount(40) > 0 }
rule: 0 1000 {statecount(40)=0 and statecount(30)=0 and statecount(20)=0 and
statecount(10)=0 and statecount(6)= 0 }
```

FIGURE 13.80 Implementing mobility model in CD++.

mobility, routing, and coverage. Numerical values are used to represent the model's state variables as follows: S = 0 (empty), S = 1 (moves to SE), S = 2 (moves to NE), S = 3 (moves to SW), S = 4 (moves to NW), S = 5 (static node), S = 6 (gateway), S = 10 (1 hop), S = 20 (2 hops), S = 30 (3 hops), S = 40 (4 hops), S = 50 (5 hops), S = 60 (cannot reach the gateway), and S = 7 (within coverage).

Nine different collision scenarios are created. Four of them are between static and mobile nodes, three are between two mobile nodes, and two are between a mobile node and a gateway. All mobile nodes change their directions at the next time unit in order to avoid collision. We also incorporate a hop-count submodel in which every node determines the next neighbor that can reach the gateway with the smallest number of hops.

Figure 13.81 shows the coverage values for the hop-count values. It can be seen that two areas are totally out of coverage. As service demand increases in these areas, network engineers should install more gateways in these regions.

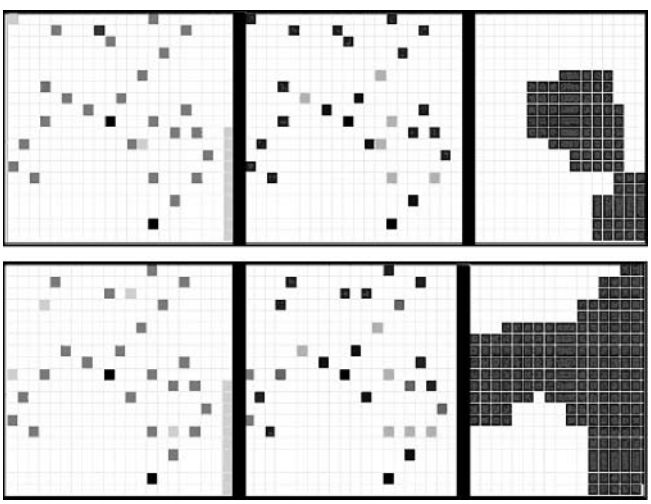


FIGURE 13.81 Mobility, hop count, and coverage.

13.8 SUMMARY

This chapter introduced multiple models of artificial systems, one of the main applications of discrete event modeling and simulation. We presented various simple models in varied areas (ranging from a load-balancing model for a database server to varied models for networking, including routing in MANets). We also introduced the application of these techniques to the field of robotic path planning and control systems. We showed how to build a complete set of tools used to simulate a simple computer. The tools can be used in computer organization courses to analyze and understand the basic behavior of the different levels of a computer system.

The benefit of using discrete-event modeling for these applications was demonstrated thoroughly. The discretization provides a drastic reduction in data volume, while having the benefit of mixing continuous and discrete time models.

The model repository includes a variety of other models in this field. The *./AdHocNetwork.zip* model contains the definition of a controller for an ad hoc networking device integrated to a two-dimensional interface. *./BluetoothZimulator.zip* includes analysis of the registration in a Bluetooth network, and the *./WSN.zip* model includes the description of a wireless sensor network. *./cruisecontrolsystem.zip* contains a detailed model of a controller for a cruise control system. The *./RoutingIP.zip* model presents a small network with IP routing.

Other simple applications include a model of an automated garage door (*./AutoDoor.zip*), a controller for a ticketing machine for a bus (*./BusVending.zip*), an automated coffee machine and other vending machines (*./CoffeeMachine.zip*, *./Vending.zip*), a model of a home alarm system (*./IntruderSystem.zip*), a model of an online system for a bank (*./OnlineBankingSystem.zip*), networking models (*./WirelessModemSimulator.zip*, *./WebClient.zip*), models of telephones and switches (*./telephoneswitch.zip*, *./telephone.zip*), and various clock models (*./clock.zip, ./clock_2.zip*, *./Pendulum_clock.zip*).

REFERENCES

1. Stallings, W. 1996. *Computer organization and architecture,* 4th ed. New York: Macmillan.
2. Tanenbaum, A. 1990. *Structured computer organization,* 3rd ed. Upper Saddle River, NJ: Prentice Hall.
3. Hennessy, J., and D. Patterson. 1994. *Computer architecture: A quantitative approach.* Upper Saddle River, NJ: Prentice Hall International.
4. Daicz, S., A. Troccoli, and G. Wainer. 2001. Experiences in modeling and simulation of computer architectures using DEVS. *Transactions of the Society for Modeling and Simulation International* 18:179–202.
5. Daicz, S., A. Troccoli, G. Wainer, and S. Zlotnik. 2000. Using the DEVS paradigm to implement a simulated processor. *Proceedings of 33rd IEEE/SCS Annual Simulation Symposium,* Washington, D.C.
6. Wainer, G., S. S. Daicz, L. De Simoni, and D. Wasserman. 2001. Using the ALFA-1 simulated processor for educational purposes. *ACM Journal on Educational Resources in Computing* 1:111–151.
7. Ameghino, J., and G. Wainer. 2000. Application of the cell-DEVS paradigm using N-CD++. *Proceedings of 32nd Summer Computer Simulation Conference,* Vancouver, Canada.
8. Behring, C., M. Bracho, M. Castro, and J. A. Moreno. 2000. An algorithm for robot path planning with cellular automata. *Proceedings of ACRI 2000,* Karlsruhe, Germany.
9. Tzionas, P., A. Thanailakis, and P. Tsalides. 1997. Collision-free path planning for a diamond-shaped robot using two-dimensional cellular automata. *IEEE Transactions on Robotics and Automation* 13:237–246.
10. Wainer, G. 2006. Modeling robot path planning with CD++. *Proceedings of ACRI 2006, LNCS 4173,* Perpignan, France.
11. Butler, Z., K. Kotay, D. Rus, and K. Tomita. 2002. Generic decentralized control for a class of self-reconfigurable robots. *Proceedings of 2002 IEEE International Conference on Robotics and Automation, ICRA 2002,* Washington, D.C.
12. Narendra, K. S., O. A. Driollet, M. Feiler, and K. George. 2003. Adaptive control using multiple models, switching and tuning. *International Journal of Adaptive Control and Signal Processing* 17:87–102.
13. Kofman, E. 2003. Quantized-state control. A method for discrete event control of continuous systems. *Latin American Applied Research Journal* 33:339–406.

14. Kofman, E. 2003. Discrete event control of time-varying plants. Technical report LSD0303, Universidad Nacional de Rosario, Argentina.
15. Campbell, A. 2005. Improvements to stochastic multiple model control: Hypothesis test switching and a modified model arrangement. MASc thesis, Carleton University, Ottawa, ON, Canada.
16. Campbell, A., and G. Wainer. 2006. Applying DEVS modeling for discrete event multiple model control of a time varying plant. *Proceedings of Winter Simulation Conference,* Monterey, CA.
17. Hedrick, C. 1988. Routing Information Protocol. *Network Working Group, Request for Comments* 1058.
18. Altman, E., and T. Jiménez. 2003. NS simulator for beginners. Technical report, INRIA Sophia-Antipolis. http://www-sop.inria.fr/mistral/personnel/Eitan.Altman/COURS-NS/n3.pdf
19. Bajaj, L., M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla. 1999. GloMoSim: A scalable network simulation environment. Technical report 990027, UCLA Computer Science Department.
20. Chang, X. 1999. Network Simulations with OPNET. *Proceedings of the 31st Winter Simulation Conference,* Phoenix, AZ.
21. Varga, A. 2001. The OMNeT++ discrete event simulation system. *Proceedings of the European Simulation Multiconference,* Prague, Czech Republic.
22. Tanenbaum, A. S. 2003. *Computer networks.* Upper Saddle River, NJ: Prentice Hall.
23. Gutowitz, H. 1995. Cellular automata and the sciences of complexity. Parts I–II. *Complexity* 1:16–22.
24. Wolfram, S. 2002. *A new kind of science.* Champaign, IL: Wolfram Media.
25. Ahmed, M. A. E., K. Yonis, A. Elsahfei, and G. Wainer. 2005. Design and implementation of a library of network protocols in CD++. *Proceedings of ANSS '05: 38th Annual Simulation Symposium,* Washington, D.C.
26. RFC-editor. 2003. Official Internet protocol standards. RFC 791. ftp://ftp.rfc-editor.org/in-notes/rfc791.txt
27. Malkin, G. 1998. RIP version 2. RFC 2453. Network Working Group, request for comments.
28. Farooq, U., G. A. Wainer, and B. Balya. 2007. DEVS modeling of mobile wireless ad hoc networks. *Simulation Modeling Practice and Theory* 15:285–314.
29. Perkins, C., E. Belding-Royer, and S. Das. 2003. Ad hoc on-demand distance vector (AODV) routing. *IETF Network Working Group,* RFC 3561.
30. Lee, C. Y. 1961. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers* EC-10, 2:345–365.
31. Hochberger, C., and R. Hoffmann. 1996. Solving routing problems with cellular automata. *Proceedings of the Second Conference on Cellular Automata for Research and Industry,* Milan, Italy.

14 Models of Urban Traffic

14.1 INTRODUCTION

The use of modeling and simulation technology for urban traffic control has a long history that can be traced back to the 1950s. It has become an indispensable tool for traffic managers and operators to study the potential impact of different design and control strategies. Over the years, a variety of traffic modeling and analysis tools has been developed based on different methodologies, including multiagent-based systems [1,2], queuing networks [3], cellular automata [4–9], DEVS [10], state charts [11], and Petri nets [12,13]. Likewise, different commercial tools are available (for instance, see references 14–18).

In this chapter, we present the use of DEVS and Cell-DEVS to model these applications. We will initially present a model of a bridge crossing. After that, we present a model of a toll area on a highway and a junction between a highway and a route. Then we introduce a model of a traffic light controller. We combine the traffic light controller with a model of the streets in the area, which are modeled using Cell-DEVS. We then present a multimodel of an urban city area. Finally, we describe the ATLAS traffic modeling language.

14.2 A MODEL FOR A BRIDGE CROSSING

The traffic model presented here represents a crossing on a bridge under construction. One way is blocked, allowing only one vehicle in each direction. In order to avoid conflicts, a traffic light is put at each end of the bridge, regulating traffic in one or the other way. A model like this one could be used to evaluate the most efficient way to allow traffic on the bridge and to optimize the waiting time of the drivers.

The model follows the structure of the bridge described in Figure 14.1. The vehicles arriving at the bridge line up in the north (N) and south (S) ends of the bridge. A control unit (CU) decides when each end gets the right to cross the bridge. The lane model represents the delay taken by a vehicle to cross the bridge, counting the number of vehicles on it. The DEVS model uses the structure shown in Figure 14.2 for the coupled model definition. Figure 14.3 shows the definition of the CU model in CD++, found in *./bridge.zip.*

The CU is in charge of synchronizing transit in both senses and is also in charge of scheduling the car's passage according to externally configurable parameters (maximum number of cars allowed on the bridge and maximum time in each direction). These values are used to control the traffic flow in each direction. The CU uses two output ports (to open the N/S gates) and two input ports (to receive cars coming to and leaving the bridge). When a new vehicle is detected through the *in* port, we increase the total number of cars and those in transit, and we verify whether the car can advance onto the bridge. To do so, we check whether the total number of vehicles in the current direction is below the maximum allowed. If we reach the maximum, we close the gate of that side and schedule an instantaneous internal transition (we have symmetric rules for the N/S gates). Otherwise, the car advances onto the bridge. In that case, we set the timeout (if no further vehicles arrive before it, we have to switch directions).

When a vehicle leaves the bridge, we decrease the number of cars in transit. If the bridge is empty and all the vehicles have passed, we switch directions. The output function informs which side is open or closed. Finally, the internal transition function checks the current phase. If it is *NONE,* this

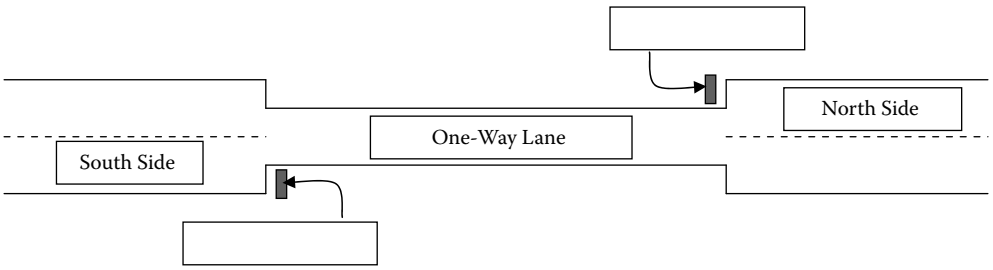


FIGURE 14.1 Schematic structure of the bridge.

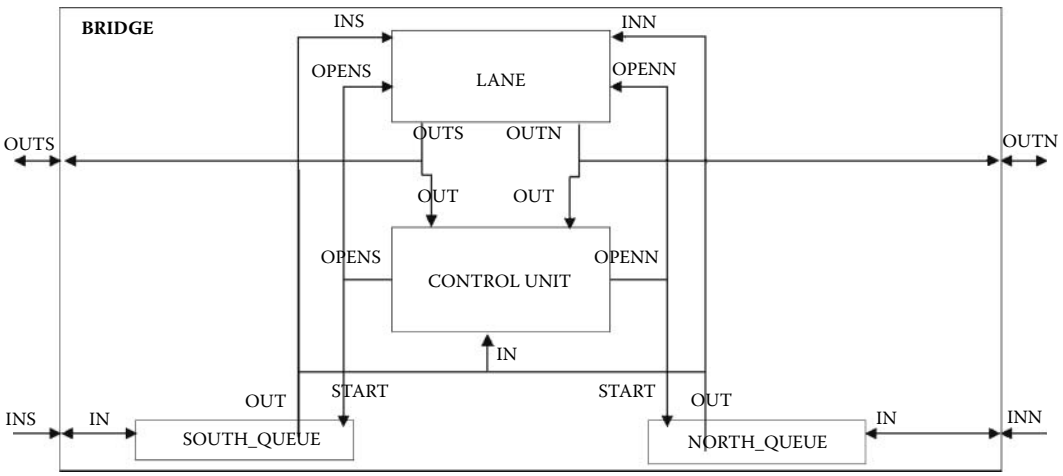


FIGURE 14.2 Bridge DEVS coupled model structure.

means that we have just received a timeout, so we have to change directions and set the timeout. Otherwise, we have just opened or closed the bridge; thus, we need to schedule the next timeout. The following table shows a simulation scenario for the model:

Inputs			Outputs		
Event Time	**Port**	**Value**	**Event Time**	**Port**	**Value**
			00:00:00:000	OPENN	1
00:00:10:000	in	1			
00:00:20:000	in	2			
00:00:30:000	in	3	00:00:30:000	OPENN	0
00:00:40:000	out	1			
00:00:50:000	out	2			
00:01:00:000	out	3	00:01:00:000	OPENS	1
00:01:10:000	in	4			
00:01:20:000	in	5			
00:01:30:000	in	6	00:01:30:000	OPENS	0
00:01:40:000	out	4			
00:01:50:000	out	5			
00:02:00:000	out	6	00:02:00:000	OPENS	1

```
Model &CarControlUnit::externalFunction (const ExternalMessage &msg) {
    if (msg.port () == in) {
        total++;
        in_transit++;
        if (way == NORTH) {
               if (total == max_north) {
                 phase = CLOSE_NORTH;
                 holdIn (active, Time::Zero);
            } else {
                 phase = NONE;
                 holdIn (active, timeout);
            }
        }
        if (way == SOUTH) {
               if (total == max_south) {
                 phase = CLOSE_SOUTH;
                 holdIn (active, Time::Zero);
            } else {
                 phase = NONE;
                 holdIn (active, timeout);
            }
        }
    }

    if (msg.port () == out) {
        in_transit--;
        if (in_transit == 0) {
            if (way == NORTH)
               if (total == max_north) {  // only change directions when all the vehicles
                       passed
                       way = SOUTH;
                       phase = OPEN_SOUTH;
               }
            else
               if (total == max_south) {   // only change directions when all the
                       vehicles passed
                       way = NORTH;
                       phase = OPEN_NORTH;
                   }
            total = 0;
            holdIn (active, Time::Zero);
       }
    }
}

Model &CarControlUnit::internalFunction (const InternalMessage &msg) {
     if (phase == NONE)   // coming from timeout; change direction
           way = (way == NORTH ? SOUTH : NORTH);
     else
        phase = NONE;  // coming from open/close: change phase and reschedule timeout
    holdIn (active, timeout);
}

Model &CarControlUnit::outputFunction (const InternalMessage &msg) {
    switch (phase) {
        case OPEN_NORTH:  sendOutput (msg.time (), openn, 1); break;
        case OPEN_SOUTH:  sendOutput (msg.time (), opens, 1); break;
        case CLOSE_NORTH: sendOutput (msg.time (), openn, 0); break;
        case CLOSE_SOUTH: sendOutput (msg.time (), opens, 0); break;
        case NONE:
               if (way == NORTH) {
                    sendOutput (msg.time (), openn, 0);
                    sendOutput (msg.time (), opens, 1);
               } else {
                    sendOutput (msg.time (), opens, 0);
                    sendOutput (msg.time (), openn, 1);
               }
            break;
    }
}
```

FIGURE 14.3 Definition of the bridge control unit model.

Initially, the N entrance is open. Then, at 10:000, the first car arrives from the N. This will trigger the external transition, which will make *total=in_transit=1.* At this point, we schedule a transition in 1 min (default timeout) and make *phase=NONE.* Ten seconds later, a new car arrives, and we repeat the procedure again at 30:000. In this case, because there are three vehicles, which is the maximum, we close the N entrance (thus, the phase is *CLOSE_NORTH,* and we schedule an instantaneous transition). The output function will then generate a value of 0 in the *OPENN* port, meaning that the N entrance is now closed. When the internal transition executes, we change our phase to *NONE* and schedule a timeout in 1 min. Then, at 40 s, a car leaves. We decrease the counter, and when the third car leaves (at 1:00:000), we change direction (*OPEN_SOUTH*) and schedule an internal transition. The output function will then open the S gate. Then three vehicles come from the S. When the three vehicles are on the bridge, the S entrance is closed. Finally, three more vehicles arrive from the N, and when they leave, the S entrance is left open.

We built two different versions of the CU model: one of them controls the number of cars on each side (like the example just presented), and the second allows cars to pass from each side during a given amount of time. Figure 14.4 shows the coupled model for the case where we use a CU counting vehicles and an experimental frame that generates vehicles and counts them. The following table shows the simulation results for this model:

Inputs			Outputs		
Event Time	**Port**	**Value**	**Event Time**	**Port**	**Value (ms)**
00:30:00:000	STOP	1	00:30:15:000	QTY2	60
			00:30:15:000	AVG2	42,900
			00:31:05:000	QTY1	60
			00:31:05:000	AVG1	100,600

```
components : ef1 ef2 bridge
in : stop
out : qty1 qty2 avg1 avg2
Link : stop stop@ef1
Link : stop stop@ef2
Link : qty@ef1 qty1
Link : avg@ef1 avg1
Link : qty@ef2 qty2
Link : avg@ef2 avg2
Link : out@ef1 ins@bridge
Link : out@ef2 inn@bridge
Link : outn@bridge in@ef1
Link : outs@bridge in@ef2

[ef1]
components : traffic1@Traffic analizer1@Analizer
in : in stop
out : out avg qty
Link : in solved@analizer1
Link : stop stop@traffic1
Link : stop stop@analizer1
Link : out@traffic1
arrived@analizer1
Link : out@traffic1 out
Link : average@analizer1 avg
Link : quantity@analizer1 qty

[traffic1]
distribution : poisson
mean : 30
...
```

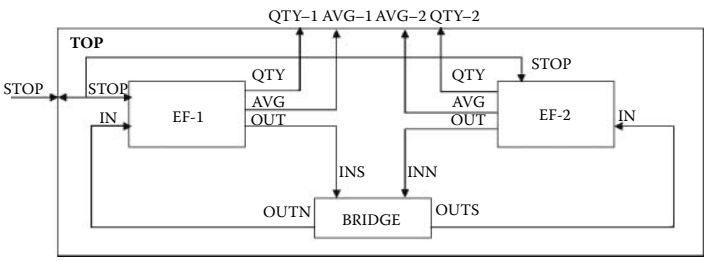


FIGURE 14.4 Definition of the bridge top model.

The simulation ran for 30 min, after which we collected the results of the vehicles passing through the bridge. We can see that, on average, every car took 42.9 s in the N–S direction and approximately 1:40:600 min in direction S–N.

The test was repeated, changing the car-based CU by the time-based CU. In order to do so, we have to include the timed CU atomic model and change the following lines in the coupled model definition:

```
components : south_queue@Queue north_queue@Queue cu@
  TimeControlUnit lane
[cu]
max_north : 00:00:30:000
max_south : 00:00:30:000
```

The following table shows the input/output events for this case:

Inputs			Outputs		
Event Time	**Port**	**Value**	**Event Time**	**Port**	**Value**
00:07:00:000	STOP	1	00:07:34:000	QTY1	14
			00:07:34:000	AVG1	55,357
			00:08:04:000	QTY2	15
			00:08:04:000	AVG2	53,000

In these 7 min of simulation, there are 14 cars in the direction S–N and 15 cars in direction N–S. When the simulation finishes, we obtain an average of 55.357 s in direction S–N and 53 s in direction N–S. When we compare the results, the average wait for the cars is better in both directions when we use the time-based CU; when we count vehicles, there might be cars waiting at a closed gate, while no cars arrive at the other end.

Exercise 14.1

Modify the CU model and include a sensor model to count vehicles arriving. If there are no vehicles on the opposite side, the timeout is extended once, allowing an extended period on the busy side.

Exercise 14.2

Run different tests, varying the maximum number of vehicles in each direction. Analyze the simulation results.

14.3 HIGHWAY TOLL STATION MANAGEMENT

In this section, we show how to apply DEVS and Cell-DEVS to model a simplified version of a junction between two highways, close to the highway toll stations, found in *./highwayDellepiane.zip* (Figure 14.5). Vehicles enter the area through areas A and B. After passing the tollbooth, vehicles converge on highway AU1 and exit through area C. The idea is to study how changes in the structure of the highway (number of lanes and their length) and the number of tollbooths can influence the throughput. In order to do so, the model must represent:

- lane length, which allows computing of the time taken by a vehicle to cross the area according to the number of lanes;
- maximum speed, which allows computing of the time taken to move from one end to the other in a lane;
- number of lanes;
- number of cars in each lane—allowing measurement of congestion in the lane; and
- number of open tollbooths.

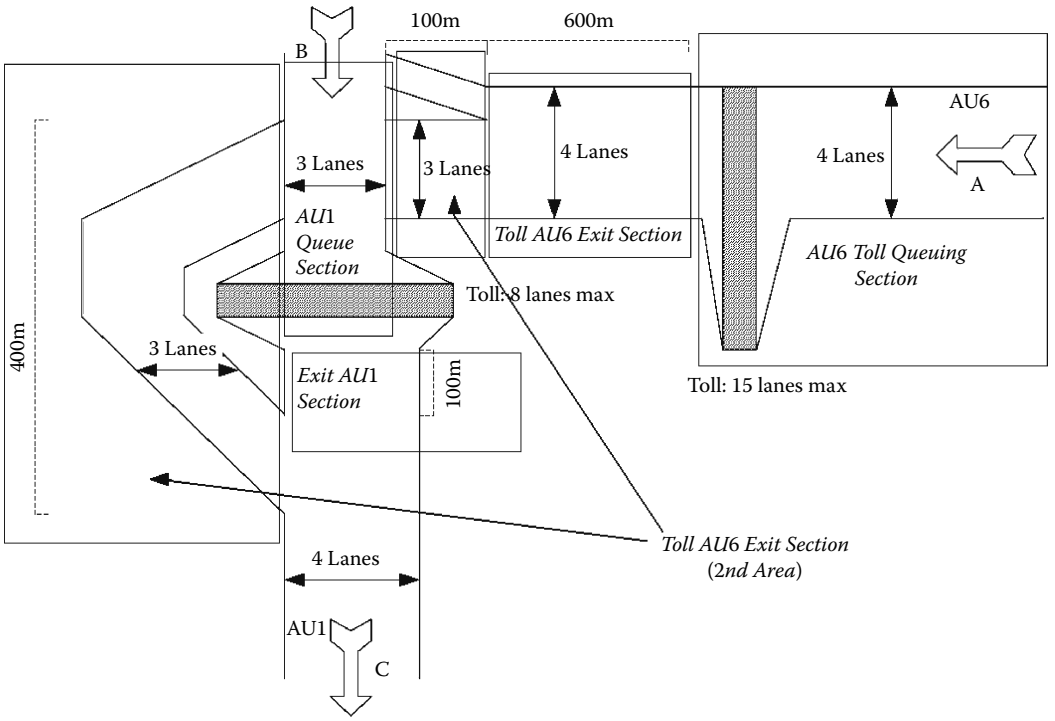


FIGURE 14.5 Highway and toll stations (different sections to analyze).

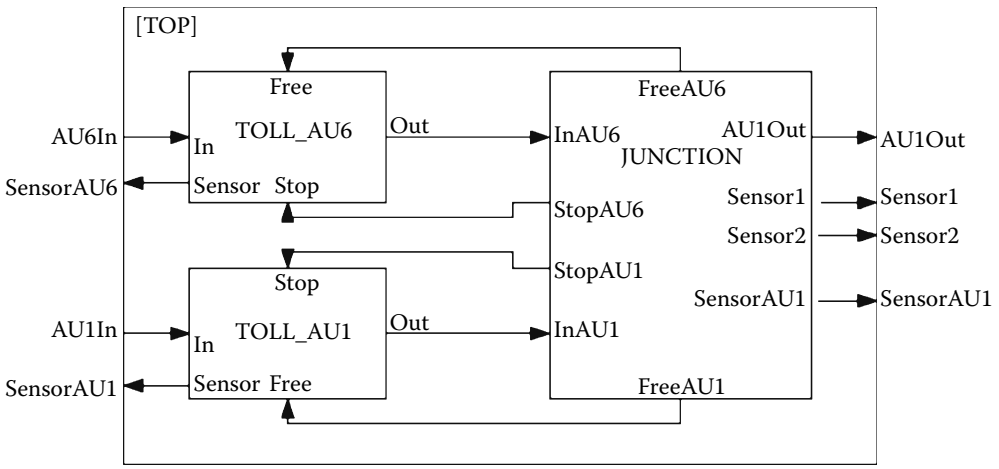


FIGURE 14.6 Junction coupled model.

We model the junction as a coupled model with the structure shown in Figure 14.6.

The coupled model has four levels (here we only show the top model, which consists of three submodels representing toll stations, sensors connected to the junction, and sensors connected to the toll stations to measure the number of passing cars). Both toll stations are modeled separately. The *AU6In* port represents the incoming vehicles to highway AU6 through entry point A. Every time a new car arrives, an external event is generated into this port, carrying the number of vehicles arriving. *AU1In* represents a similar input port for the AU1 highway and entry point B. The *sensor* output ports collect information about cars leaving the area.

Both toll models have the same structure: an input queue, receiving vehicles waiting for the tollbooth to be free, and the tollbooth, which provides service to the vehicles (both are instances of the same model, with different parameters, including queue capacity, and average service time for the tollbooth.). The *in* ports represent vehicles arriving at the toll station; the *stop* port is used to receive information by the junction (if there is congestion ahead, the booth delays vehicles). The *out* port is used to represent the cars leaving the tollbooth. Finally, the *free* port is used to indicate that the congestion situation has finished and that new cars can move to the junction.

The *junction* model represents the area between the two toll exits and the exit of highway AU1. *InAU1* and *InAU6* represent the car intake for the area. The *stop* ports are used to inform when there is congestion in the area (so that the toll stations stop the cars). The *free* ports are used to inform that a congestion situation has finished and room is available to receive more cars (so that the toll stations will allow new vehicles to get into the area). The *sensor* ports give information about the number of cars in different sections in the area.

Figure 14.7 shows the implementation of the *intersection* model (a subcomponent of *junction*) in CD++. The model has two input ports (*carA/B*) to report the arrival of new cars at the intersection. Ports *free* and *stop* are used to control the flow of vehicles. The *lanes* variable stores the number of lanes in the highway. When a new car arrives, we compute the number of cars rejected (`arrived+no_to_dispatch - lanes`); that is, the section can only accept as many vehicles as space is available (`no_to_dispatch` contains the vehicles at the intersection and not yet dispatched). We then compute the number of vehicles to dispatch (minimum between the number of lanes and the cars in the section). The variables *sl* indicate that a vehicle has been rejected. We then compute the current state:

- If the model is *passive* and we have vehicles to reject, we change to *rejecting* state. In this way, we inform the number of cars rejected to the originator (a tollbooth), which will stop sending more vehicles and add them to its queue of vehicles (to be re-sent when congestion ends). If no vehicles are rejected, the model is *active*. We use two different times for each event: *dispatchTime* to represent the time it takes the vehicles to leave the area and *restartTime* to represent the time taken to reject vehicles. In the case of rejection, we schedule an internal transition after *restartTime* and save the difference between the *dispatchTime* and the *restartTime*. (The intersection continues dispatching existing vehicles; thus, we need to record the difference between the two events.)
- If the model is *active,* we first check for rejection cases. If there are rejections and the next scheduled internal event is after the *restartTime,* we change to the *rejecting* state. Otherwise, we remain active until the next internal event. We consider the difference between the next scheduled event and the time taken to reject the overflow vehicles. (The cars are already being dispatched, and we still need to consume the dispatch time.)

When a new vehicle arrives while the model is *active,* we check for rejections and, if needed, reschedule the next internal transition. If the model is *passive* and there are rejections, we move to the *rejection* state. Otherwise, the model becomes *active*.

When the internal transition executes, we output the number of vehicles (or rejections). If we are in the *restart* state, we inform that more room is available, and the internal transition function produces a state change. If the model is active, the number of vehicles is reset. If there was a rejection, we have to inform the sender that room is available; thus, we change to *restart* and schedule an internal transition. If the model was reinitializing or rejecting, we reset all the counters.

This model shows the use of model timing information as part of the external transition function. If we compute the restart time and it is larger than a previously scheduled event, the first event takes priority and the state does not change. Otherwise, the state is changed.

```
Model &Intersection::externalFunction( const ExternalMessage &msg ) {
        int no_arrived = static_cast< int >(msg.value());

        if (msg.port() == carA){
                rejected_a += max(0, no_arrived + no_to_dispatch - lanes);
                no_to_dispatch = min(no_to_dispatch+no_arrived, lanes);
        } else if (msg.port() == carB){
                rejected_b += max(0, no_arrived + no_to_dispatch - lanes);
                no_to_dispatch = min(no_to_dispatch+no_arrived, lanes);
        }
        sla = (rejected_a > 0) || sla;
        slb = (rejected_b > 0) || slb;

        if (no_to_dispatch > 0 || rejected_a >0 || rejected_b > 0){
                if (phase == passive){
                        if ( rejected_a >0 || rejected_b > 0 ){
                                phase = rejecting;
                                TimeOfNextTransition = dispatchTime - restartTime;
                                holdIn( active, restartTime ) ;
                        } else {                        // no cars rejected: dispatch
                                phase = active;
                                holdIn( active, dispatchTime ) ;
                        }
                } else if (phase == active){
                        if ( (rejected_a >0 || rejected_b > 0) && (nextChange() >
                                restartTime) ){
                                TimeOfNextTransition = nextChange() - restartTime;
                                phase = rejecting;
                                holdIn( active, restartTime ) ;
                        } else
                                holdIn( active, nextChange() ) ;
                }
        } else
                passivate();           // no cars to dispatch or reject
}

Model &Intersection::internalFunction( const InternalMessage & )
        if (phase == active){
                no_to_dispatch = 0;
                if (sla || slb) {
                        phase = restart;
                        holdIn(active, restartTime);
                } else
                        passivate();
        } else if (phase == restart){
                rejected_a = rejected_b = sla = slb = 0;
                passivate();
        } else if (phase == rejecting){
                rejected_a = rejected_b = 0;
                phase = active;
                holdIn(active, TimeOfNextTransition);
        }
}

Model &Intersection::outputFunction( const InternalMessage &msg ) {
        if (phase == active){
                if (no_to_dispatch > 0) sendOutput( msg.time(), carOut, no_to_dispatch );
                if (rejected_a > 0) sendOutput( msg.time(), stopA, rejected_a );
                if (rejected_b > 0) sendOutput( msg.time(), stopB, rejected_b );
        } else if (phase == restart) {
                if (sla) sendOutput( msg.time(), freeA, 0 );
                if (slb) sendOutput( msg.time(), freeB, 0 );
        } else if (phase == rejecting) {
                if (rejected_a > 0) sendOutput( msg.time(), stopA, rejected_a );
                if (rejected_b > 0) sendOutput( msg.time(), stopB, rejected_b );
        }
}
```

FIGURE 14.7 Intersection atomic model.

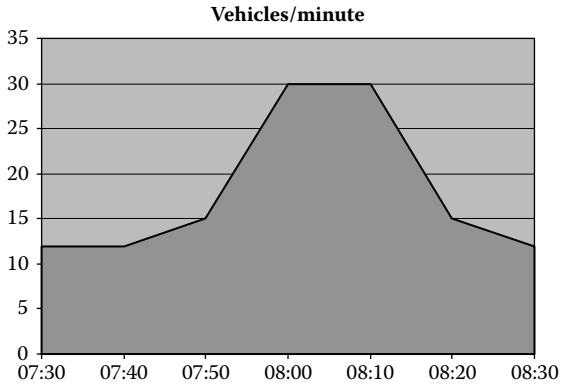


FIGURE 14.8 Input vehicle distribution.

We run this model using different scenarios. The case we show here represents the case of peak time (7:45 to 8:30 a.m.), and it uses the distribution presented in Figure 14.8 to generate the inputs to the area (a similar distribution was used for both highways).

In order to study the simulation results in detail, we registered the information provided by the sensor ports in the model, and we computed the number of vehicles through those sensors in time. The sensors periodically report the number of vehicles on each of the sections depicted in Figure 14.5. *SensorAU1* collects information about the exit of the highway AU1; *Sensor1* collects information on tollbooth AU6 and *Sensor2* on the second area of AU6.

The diagrams in Figure 14.9 show the traffic status in each sector. The timescale in every figure starts at 7:45 a.m. and ends at 9:00 a.m. (the last vehicle enters the area at 8:32 a.m.). On AU6, although the number of vehicles reduces with time, there is congestion at the exit of the toll area. The opposite occurs on AU1: traffic is congested quickly, but transit in the direction of AU1 is fluid (eight cars constantly). When we execute a similar test in which AU1 receives an average flow of six cars/min, we can observe that, although traffic is heavy on AU6, it is fluid and there is no congestion. A different test uses a similar distribution for AU1, while AU6 receives six cars/min. The result is congestion in the entrance of AU1 (due to the small capacity) and fluid traffic in the rest of the system.

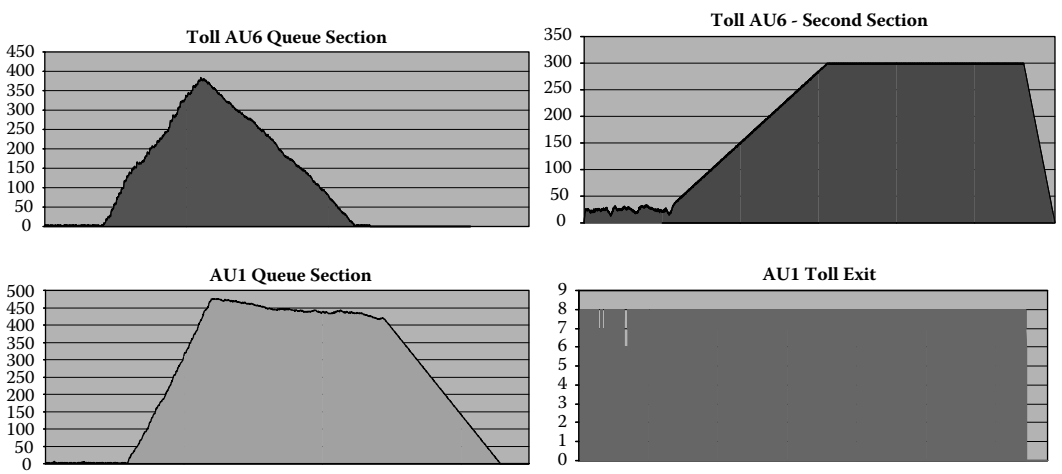


FIGURE 14.9 Traffic status in different sections.

Exercise 14.3

Modify the number of lanes in the Toll AU6 second section and study the results of the simulation.

Exercise 14.4

Change the car distribution input and repeat the study.

Exercise 14.5

Make AU1 have five lanes in total and analyze the results.

Exercise 14.6

Create a test in which the delay taken for the tollbooths is reduced or extended and analyze the simulation results for the area.

14.4 HIGHWAY JUNCTION

In this section, we present a model of the intersection of two routes converging onto a highway. Both routes have three lanes, and the highway is five lanes wide, as seen in Figure 14.10.

A model of this section would allow the modeler to analyze the behavior emerging from reducing from six to five lanes in total and to study congestion problems. The model was built as three Cell-DEVS models coupled to each other; two represented the routes and the third represented the highway. We used DEVS models to generate and consume traffic in the area.

The cellular models consider the driving behavior, including three basic movements: forward movement, passing slower cars using the left lane, and, if the left lane is occupied, passing slower cars using the right lane. Figure 14.11 shows the general structure of route R9.

This model can be defined as

$$\text{R9} = < X, Y, X_{list}, Y_{list}, \mu, N, \{m,n\}, C, B, Z, select> \tag{14.1}$$

where

X_{list} = {(0,0);(1,0);(2,0);(0,9);(1,9);(2,9)}
Y_{list} = {(0,9);(1,9);(2,9)}

FIGURE 14.10 Scheme of the route junction.

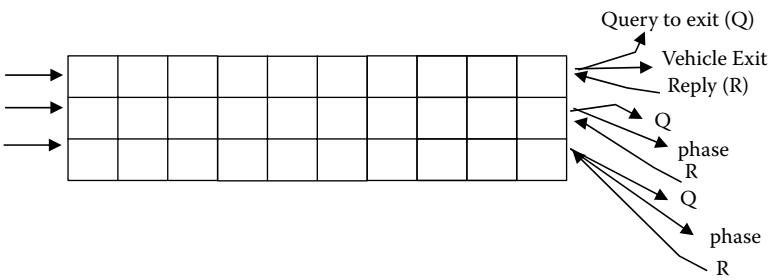


FIGURE 14.11 Scheme of the route junction.

I = <P^x, P^y>, where
P^x = { <X(0,0),binary>,<X(0,1),binary>,<X(0,2),binary>,<X(0,9),binary>,<X(1,9),binary>, <X(2,9),binary> };
P^y = { <$Y(0,9)^1$,binary>, <$Y(0,9)^2$,binary>,<$Y(1,9)^1$,binary>, $Y(1,9)^2$,binary>,<$Y(2,9)^1$,binary >, <$Y(2,9)^2$,binary>}; where
1 and 2 represent each of the output ports for the cells (which connect with another route or highway). One of the ports is used to query whether there is room in the next route and the other one to obtain a response from it
μ = 11; N = { (–2,–1), (–2,0), (–1,–1), (–1,0), (–1,1), (0,–1), (0,0), (0,1), (1,–1), (1,0), (1,1) }
$X = Y$ = {0,1}; **m** = 3; **n** = 10; **B** = nowrapped
Z is defined as in Cell-DEVS specification
select = {(–1,–1), (–1,0), (–1,1), (0,–1), (0,0), (0,1), (1,–1), (1,0), (1,1), (–2,–1), (–2,0) }

Each cell in the cell space is defined by

$$\mathbf{Cij} = <I, X, S, Y, N, \delta_{int}, \delta_{ext}, d, \tau, \lambda, D> \tag{14.2}$$

where
I = < η, P^x, P^y >, where μ = 11
P^x = { <X_1,binary>,..,<X_{11},binary> }, P^y = { <Y_1,binary>,..,<Y_{11},binary> }
$X = Y$ = {0,1}
S = $\begin{cases} 1, & \text{if there is a car in the cell} \\ 0, & \text{otherwise} \end{cases}$
N = {(–2,–1), (–2,0), (–1,–1), (–1,0), (–1,1), (0,–1), (0,0), (0,1), (1,–1), (1,0), (1,1)}
d = speedA (a function of the speed of the vehicles)

We use an extended Moore's neighborhood. We need to include two extra cells to the N in order to permit the vehicles to pass on the right without colliding. If a car decides to pass a vehicle using the right lane, that is because it tried to pass on the left first and could not make it, as seen in Figure 14.12.

In order to define the behavior of the cell to which the arrow is pointing, we need to check the value of the cell to the N, the second cell to the N (in order to see if it is blocked, as in the figure), and the second cell to the NW (cell *(–2,–1)*). If there is a vehicle moving forward in that cell, the vehicle cannot pass on the left, either; this is used to model a shoulder check action.

The rules defined in the model include different kinds of behaviors. The following rules are used to model forward movement (Figure 14.13(a)):

```
0 d {(0,0)=1 and (0,1)=0}
1 d {(0,0)=0 and (0,-1)=1}
```

The following rules describe the behavior of Figure 14.13(b) (i.e., a vehicle passing on the left):

```
0 d {(0,0)=1 and (0,1)=1 and (-1,1)=0 and (-1,0)=0}
1 d {(0,0)=0 and (1,0)=1 and (1,-1)=1 and (0,-1)=0}
```

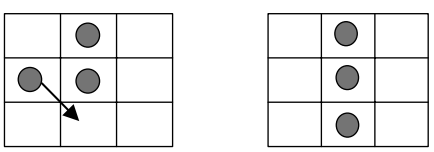


FIGURE 14.12 Passing on the right.

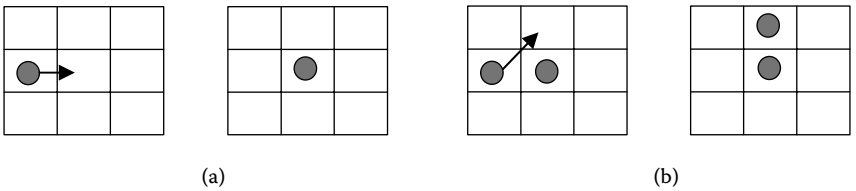


FIGURE 14.13 (a) Forward movement; (b) passing on the left.

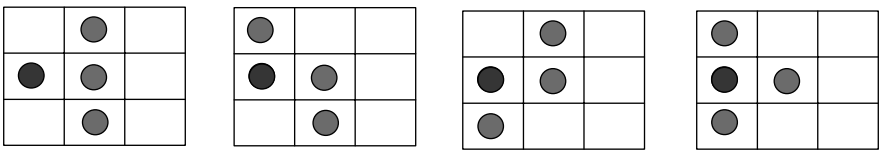


FIGURE 14.14 Stopped vehicles.

We check whether the cell (–1,0) is empty, because we will first move to this cell, and then we move forward. To pass on the right, we do as in Figure 14.12. In this case, we need the cell (1,0) to be empty because we will first move to this cell and then we move forward. In any of the cases shown in Figure 14.14, the vehicle cannot move forward.

Figure 14.15 shows a definition of the model, found in *./highwayint.zip,* using CD++. Forward movement rules are like those previously defined in Figure 14.13 (we use state values 40 and 41 to distinguish vehicles on the highway from those in R8 and R9). The model for route 8 is identical to the one used for route 9. The HW1 model is similar to the route 9 model, changing the size of the cell space (m = 5; n = 10). We also use a DEVS generator, which generates traffic using a normal distribution with mean value of 100 and standard deviation of 0. We have specialized behavior for the cells joining R9 with HW1: the first rules represent a vehicle leaving R9 to the highway (whose cells are marked with the value 40). Vehicles try to pass on the left; if blocked, they try on the right (Figure 14.16).

The last two cells of R8 are connected to the same cell in HW1 (4,0). The idea is to model the fact that these two cells are joined into one before entering HW1. We need different behavior in each of the last two cells in R8 in order to deal with conflicts if both cells are occupied at the same time (in this case, lane 2 has higher priority). The other cells in R8 and R9 are directly connected to HW1 in the corresponding lanes (Figure 14.17).

14.5 TRAFFIC LIGHT CONTROLLER

In this section, we present a simple local traffic control model (depicted in Figure 14.18), which reproduces the behavior of the traffic lights in a crossing and can be used to identify the factors that can affect traffic throughput. We model only one intersection composed of four traffic lights, each of them controlling traffic in one direction. The following assumptions serve as the boundary conditions of the system (which conform to the experimental frame of the model):

- Only traffic in four directions is considered; right and left turn traffic is not studied.
- The traffic in each direction has at least one lane.
- The length of each lane is unlimited (i.e., each lane is capable of accommodating an unlimited number of vehicles).
- The capacity of the intersection is limited (the maximum number of vehicles it accommodates is constant).

```
components : traffic genLane1-R9@generator genLane2-R9@generator genLane3-R9@generator
components : genLane1-R8@generator genLane2-R8@generator genLane3-R8@generator

link : out@genLane1-R9 inLane1-R9@traffic
link : out@genLane2-R9 inLane2-R9@traffic
link : out@genLane3-R9 inLane3-R9@traffic

link : out@genLane1-R8 inLane1-R8@traffic
link : out@genLane2-R8 inLane2-R8@traffic
link : out@genLane3-R8 inLane3-R8@traffic

[traffic]
type : cell            dim : (8,20)          delay : transport     border : nowrapped
neighbors :    (-2,-1) (-2,0) (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1) (1,0) (1,1)
localtransition : rules
in : inLane1-R9 inLane2-R9 inLane3-R9
in : inLane1-R8 inLane2-R8 inLane3-R8
link : inLane1-R9 inLane1-R9@traffic(1,0)
link : inLane2-R9 inLane2-R9@traffic(2,0)
link : inLane3-R9 inLane3-R9@traffic(3,0)

[rules]
...

%---------- Forward movement, R9 ------------------------
rule : 0 100 {(0,0)=1 and (0,1)=0}
rule : 0 100 {(0,0)=1 and (0,1)=1 and (-1,1)=0 and (-1,0)=0}
rule : 0 100 {(0,0)=1 and (0,1)=1 and (1,0)=0 and (1,1)=0 and ((-1,1)=1 or (-1,0)=1) }

...
%---------- Forward movement, HW1 ------------------------
% passing through the right when we are on the first lane
rule : 40 100 {(0,0)=41 and (0,1)=41 and (1,1)=40 and (1,0)=40}
rule : 41 100 {(0,0)=40 and (0,-1)=40 and (-1,-1)=41 and (-1,0)=41}

%---------- Joining R9 and HW1 ------------------------
rule : 0 100 {(0,0)=1 and (0,1)=40}                  ; moving forward
rule : 0 100 {(0,0)=1 and (0,1)=41 and (-1,1)=40 and (-1,0)=0}  ; passing on the left
rule : 0 100 {(0,0)=1 and (0,1)=41 and (1,0)=0 and (1,1)=40 and ((-1,1)=41 or (-1,0)=1)} ; right

rule : 41 100 {(0,0)=40 and (0,-1)=1}        ; moving forward
rule : 41 100 {(0,0)=40 and (1,0)=41 and (1,-1)=1 and (0,-1)=0}    ; passing on the left
rule : 41 100 {(0,0)=40 and (-1,0)=41 and (-1,-1)=1 and (0,-1)=0 and ((-2,0)=41 or (-2,-1)=1) )}
...

[newVehicle-rule]
%a new vehicle arrives. (0,0)=0 in R9, and 20 in R8. 1 and 21: the cells are occupied
rule : {(0,0)+1} 100 {portvalue(ThisPort)> 0 and ( (0,0)=0 or (0,0)=20 )}
```

FIGURE 14.15 Cell-DEVS definition of the model.

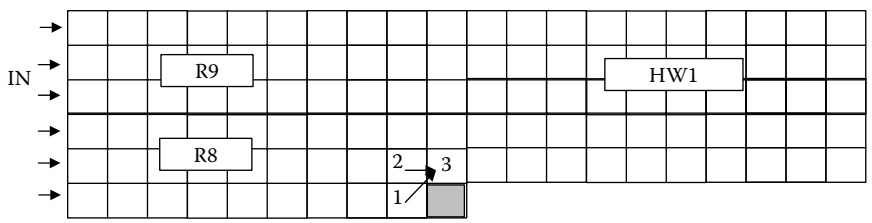


FIGURE 14.16 Coupling scheme of the cell spaces.

- We consider red light overlapping for any two intersecting directions; when the light in one direction becomes red, the lights in the other intersecting directions must be red before turning into green.
- Vehicles do not change lanes.
- Vehicles moving in a given direction never enter the intersection if the traffic light for that direction is red.

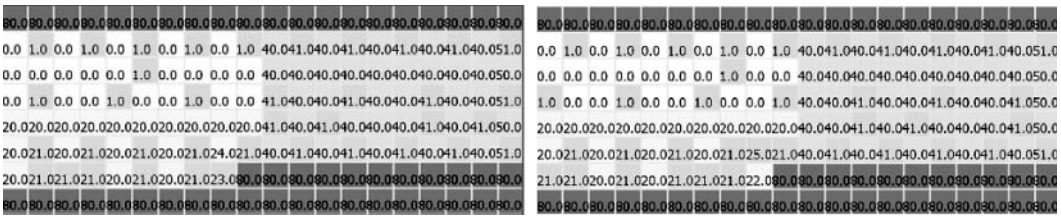


FIGURE 14.17 Simulation results.

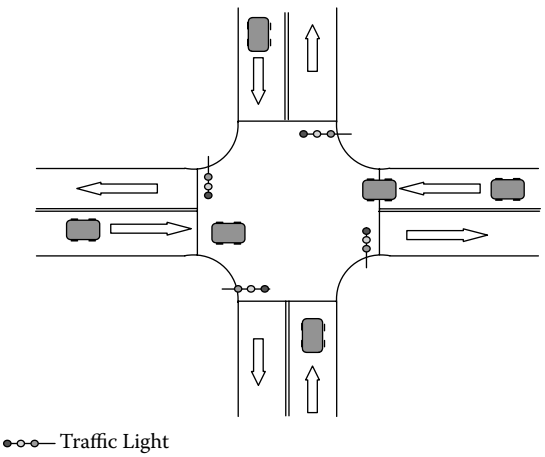


FIGURE 14.18 Simple local traffic control system (one intersection).

- A vehicle in a given direction enters the intersection only if the traffic light is not red or yellow.
- No vehicle stops in the intersection area at any given time.
- Each vehicle in traffic uses the same amount of time to pass through the intersection.
- No pedestrian control devices are considered.

Figure 14.19 shows the general structure of the intersection model we created, which can be found in *./trafficlight.zip*.

The following components can be identified:

- *light_group* (two instances: ab–cd) is a controller responsible for managing a group of lights (one per direction). The controller communicates with other group controllers to synchronize the crossing lights, and it outputs the same signals to the two lights belonging to the group.
- *traffic_light* (four instances: a–d) components represent each light for unidirectional traffic.
- *light_controller* receives signals from a group controller, and it turns on only one of its three lights according to the signal. It then outputs an enable or a disable signal to all *lane* models controlled by this light (to permit or avoid vehicles in the intersection).
- *lane* represents one lane and its connection with the area of the intersection.
- *lane_queue* queues vehicles in the specified lane waiting to pass through the intersection.

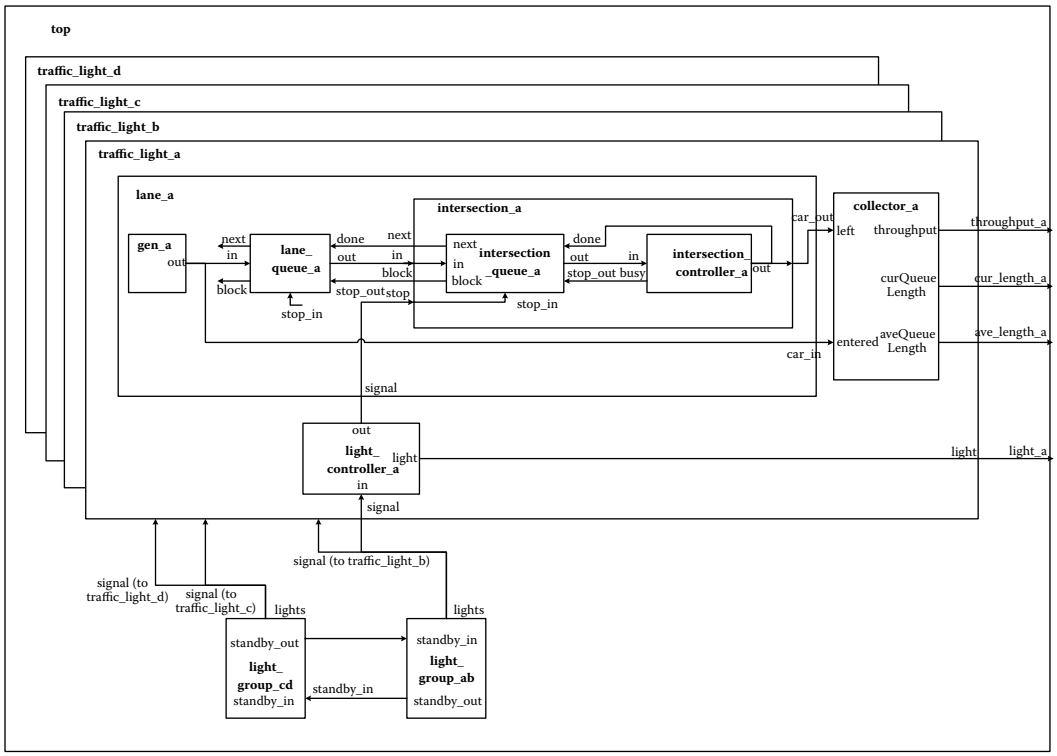


FIGURE 14.19 Traffic light system structure.

- *intersection* is a coupled component that represents the intersection area.
- *intersection_queue* queues the cars passing through the intersection. It has a limited capacity (one to two vehicles, depending on the real length of the intersection).
- *intersection_controller* is responsible for controlling each car passing through the intersection.

A detailed formal specification of each of the models can be found in *./trafficlightsys.zip*. In the following, we show the specification for the *light_group* atomic model:

$$\text{lights\_group} = <S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta> \tag{14.3}$$

where

$X = \{standby\_in\}$, which is used to represent red overlapping
$Y = \{standby\_out, lights \in \{$ GREEN, YELLOW, RED $\}\}$
$S = \{light_state \in \{$GREEN, YELLOW, STANDBY, RED$\}\}$

$\delta_{int}(light\_state)$ {
  switch(*light_state*)
    case GREEN: set light_state as YELLOW;
    case YELLOW: set light_state as RED;
    case STANDBY: set light_state as GREEN;
    case RED: cannot happen; throw an exception;
}

```
δext(light_state,e, x) {
  if(light_state is not RED) // states of two sets of lights are not synchronized correctly.
       Throw an exception;
  light_state = STANDBY;
}

λ(light_state) {
       switch(light_state)
              case GREEN:    output YELLOW at port lights;
              case YELLOW:   output RED at port lights;
                             output RED at port standby_out;
              case STANDBY:  output GREEN at port lights;
              case RED:      throw an exception;
}

ta(light_state) {
  switch(light_state)
       case GREEN:    ta(s)= green_time;
       case YELLOW:   ta(s)= yellow_time;
       case STANDBY:  ta(s)= standby_time;
       case RED:      ta(s)=infinity;
}
```

The internal transition function is in charge of switching the states of the traffic lights in the group. According to the current color, if the light was green, the output function transmits the *YELLOW* color on the *lights* port. When the light is yellow, a *RED* value is sent to the *lights* and the *standby_out* ports (to create overlapping red signals; a green signal can only happen after a standby). We guarantee the occurrence of a standby state (in which the two traffic lights are red) before changing to green. After transmitting the current state, the internal transition function changes the state of the traffic light. We only change to green after a standby period, which is controlled by the external transition function (and will be activated only when a standby signal is received from the opposite light). A model with a red light becomes passive waiting for this signal.

Figure 14.20 shows the implementation of the intersection controller model using CD++. The controller controls each car in the intersection. The intersection controller can be *passive* or *busy,* or *outputting* a signal. If the external transition activates during the *passive* state, the model changes to *busy* during the time taken to process this input. If the input arrives during a *nonpassive* state, the previous internal transition time is rescheduled at the original time (i.e., we have to finish the previous request by scheduling a transition at the original time, computed as the difference between the originally scheduled time and the current time). If the model is *passive,* we schedule an internal transition according to the response time of the controller. When the time is consumed, the current state is transmitted. Then we activate the internal transition function, which changes the current state to the next phase.

Figures 14.21 and 14.22 show the simulation of the whole system case for a given period. In our first example, a regular traffic control scheme was used and the duration of each color was

- in A–B direction, G: 30 s, Y: 3 s, R_overlap(standby): 2 s; and
- in C–D direction: G: 45 s, Y: 3 s, R_overlap(standby): 2 s.

Figure 14.21 shows the simulation results by checking the green, yellow, and standby time for AB and CD, respectively, and the throughput at red light time and green light time. At time

```
Model &IntersectionController::externalFunction( const ExternalMessage &msg )  {
        if(state()==passive)          {
                action |= BUSY;
                holdIn(active, responseTime);
                serviceTime = msg.value()>0 ? Time(0,0,0,int(msg.value() * 1000)) : responseTime;
        }
        else    {
                printf("%s@%s: discarded input[%f] when busy\n", description().data(),
                                        className().data(), msg.value());
                holdIn(active, nextChange-msg.time());
        }
}

Model &IntersectionController::internalFunction( const InternalMessage & )  {
        if(action & BUSY)      {
                action &= ~BUSY;
                action |= OUTPUT;
                holdIn(active, serviceTime - responseTime);
        }
        else if(action & OUTPUT)      {
                action = 0;
                passivate();
        }
        else    {
                printf("%s@%s: error state in output function %d\n",
                        description().data(), className().data(), action);
                action = 0;
                passivate();
        }
}

Model& IntersectionController::outputFunction( const InternalMessage &msg )  {
        if(action & BUSY)      {
                sendOutput(msg.time(), busy, 1);
        }
        else if(action & OUTPUT)      {
                sendOutput(msg.time(), out, 1);
        }
        else    {
                printf("%s@%s: error state in output function %d\n",
                        description().data(), className().data(), action);
        }
}
```

FIGURE 14.20 Definition of the intersection.

00:00:02:010, both *light_a* and *light_b* are green (value = 2). At 00:00:32:010, the lights turn yellow (value = 3) because G_{AB} = 30 s; at 00:00:35:010, *light_a* and *light_b* turn into red (value = 1) because Y_{AB} = 3 s. At the same time, *light_c* and *light_d* start the standby period. At 00:00:37:010, *light_c* and *light_d* turn green because $STANDBY_{CD}$ = 2 s. We can also notice that when lights C and D are red (from 00:00:02:010 to 00:00:37:010), the throughput in the CD direction is 0; after lights C and D become green, the throughput in C and D becomes 20. We can also see the throughput of the system and the size of the queues.

Our next example shows how the traffic light time schedule for one direction is affecting its average queue length. If we consider G_{AB} = 30, Y_{AB} = 3, and $STANDBY_{AB}$ = 2 and G_{CD} = 45, Y_{CD} = 3, and $STANDBY_{CD}$ = 2, then we can guess that in the direction CD, the average length is likely shorter

```
00:00:02:010 light_a 2
00:00:02:010 light_b 2
00:00:30:000 throughput_a 16
00:00:30:000 curlen_a 1
00:00:30:000 avelen_a 1
00:00:30:000 throughput_b 20
00:00:30:000 curlen_b 1
00:00:30:000 avelen_b 1
00:00:30:000 throughput_c 0
00:00:30:000 curlen_c 10
00:00:30:000 avelen_c 10
00:00:30:000 throughput_d 0
00:00:30:000 curlen_d 9
00:00:30:000 avelen_d 9
00:00:32:010 light_a 3
00:00:32:010 light_b 3
00:00:35:010 light_a 1
00:00:35:010 light_b 1
00:00:37:010 light_c 2
00:00:37:010 light_d 2
00:01:00:000 throughput_a 6
00:01:00:000 curlen_a 7
00:01:00:000 avelen_a 4
```

FIGURE 14.21 Intersection simulation results.

```
00:14:30:000 avelen_a 62.6552
00:14:30:000 avelen_b 56.7931
00:14:30:000 avelen_c 31.1724
00:14:30:000 avelen_d 27
...
00:48:30:000 avelen_a 202.454
00:48:30:000 avelen_b 183.773
00:48:30:000 avelen_c 91.3608
00:48:30:000 avelen_d 75.3814
...
```

FIGURE 14.22 Average size of the queues.

than in the direction AB over a certain period. We can verify this by analyzing the output data in Figure 14.22. We observe that the average length of the traffic queue in the AB direction is always longer than in the CD direction. More interestingly, the lengths of all of the queues are increasing with time. This means that there are too many vehicles for the observed intersection or that the traffic capacity of the intersection needs improvement.

Exercise 14.7

Modify the *Intersection_Controller* model in order to allow left and right turns. To do so, the component must check the status of the other controller components before allowing a car to cross the intersection.

Exercise 14.8

Extend the previous model to support pedestrians crossing the intersection. Construct a pedestrian crossing controller and connect it to the coupled model previously defined.

We extended this model and put two traffic controllers together, using a Cell-DEVS connecting them and reproducing the traffic behavior between two crossings (as shown in Figure 14.23). We used the following assumptions for the model:

- There is only one lane per direction.
- Vehicles use four different speeds: stopped, low, middle, and high, which are used to decide how long it takes to move from one cell to the next.
- Lanes are composed of a limited number of cells, and each cell can be occupied by only one vehicle at a time. When all cells are occupied, no more vehicles are allowed to move into the lane. If a lane is full, no vehicles are allowed in the previous intersection, either.
- A vehicle can only travel from one cell to its adjacent cell ahead, with speed variation not exceeding one level.
- Each vehicle is responsible for keeping a safe space behind the vehicle in front of it. The safety space size is in proportion to its speed: three spaces ahead for a high-speed car, two for middle speed, and one for low speed.
- Within the intersection area, a vehicle can change its speed to a lower or a higher level, and the total intersection passing time is determined by the vehicle speed.
- When the traffic light is yellow, if a vehicle can stop safely, it must stop; otherwise, it should pass through the intersection.

As shown in Figures 14.23 and 14.24, the models we used are similar to the one presented earlier, combined with a coupled Cell-DEVS that defines the behavior of the vehicles between the intersections. We have three new basic models—namely, the *light controller,* the *segment,* and the *crossing.*

Each coupled model defines the behavior of each lane, intersection, and traffic light. The *light controller* model has the functionality of the group controller of Figure 14.19. The *segment* model represents a lane queue, and the intersection contains a queue for crossing vehicles going to the *crossing* model. We considered only the case of the interconnection between two intersections and traffic in horizontal direction. The segment model is defined as in Figure 14.25.

The segment is a 4×20 Cell-DEVS with inertial delays for each lane in each direction (as shown in Figure 14.25). A segment begins from the point where a car enters and ends at the point where a car leaves and enters a crossing. The first row is used to keep the position of a car in the segment and to record the current state of the car. The second row is employed for timing the state change; the

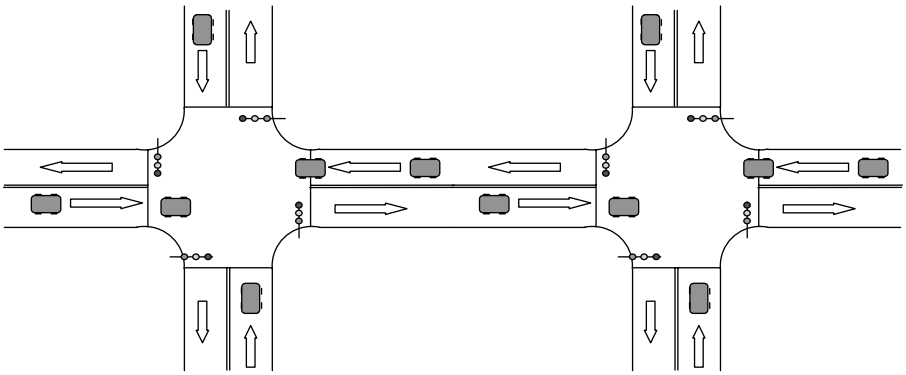


FIGURE 14.23 Two intersections.

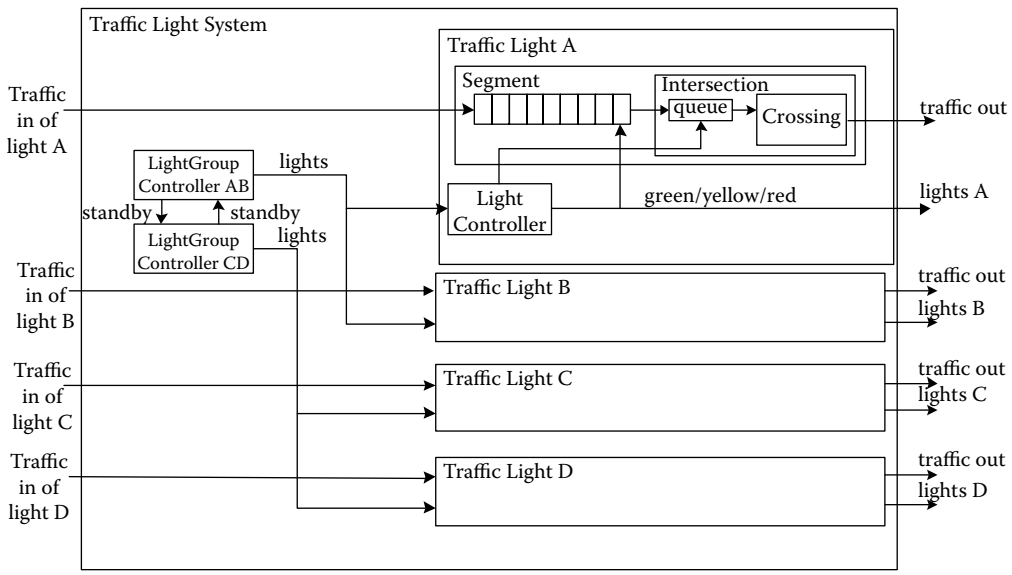


FIGURE 14.24 Two-intersection coupled model definition.

third row conducts generic control (including the decision of increasing or decreasing the current speed), and the fourth gets the light signal (input) for cells that are close to the intersection.

The car state is encoded with the format *s.nodd,* with *s* the current speed, *n* the next speed, *o* the output value (only used for the cell with *car_out* port; no output is generated when *output* is 0), and *dd* is the timing delay. For example, *4.0306* represents a high speed (4), the next is null (0), the *output* is middle speed (3), and the time that the current car will be leaving the current cell to the next (delay) is $06 \times 100 = 600$ ms.

Figure 14.26 shows the CD++ definitions for the model. The model defines a cell space with the structure presented in Figure 14.25. We use a special rule for the border cells because these should receive and transmit vehicles. The crossing model is also defined as a Cell-DEVS using the same rules for segments but reduced size for the cell space. Likewise, we use a subset of the transition functions (i.e., local transition, in-port transition, and zone transition functions) of the *segment.*

The *light controller* was also defined as a Cell-DEVS model as follows:

$$\text{Light} = \langle I, X, Y, Xlist, Ylist, \eta, N, \{m,n\}, C, B, select \rangle \tag{14.4}$$

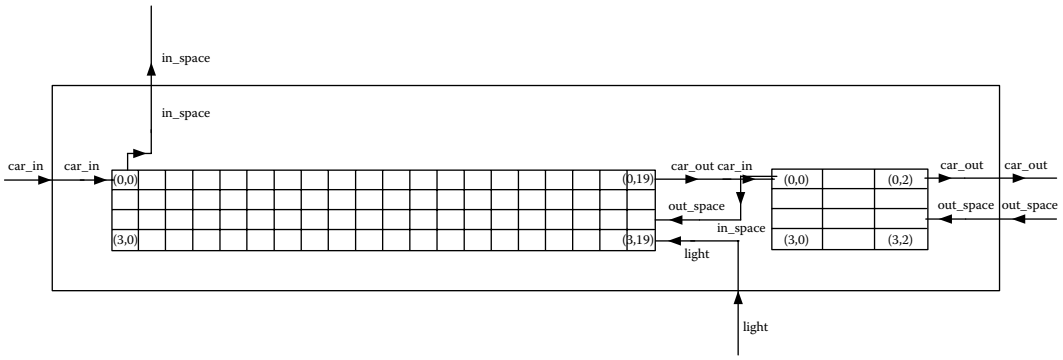


FIGURE 14.25 Segment model.

```
[segment]
type : cell                    height : 4        width : 20        delay : inertial
border : nowrapped
neighbors : (-3,0)  (-2,0)  (-1,0)  (0,-1) (0,0) (0,1) (0,2) (0,3)  (1,0)  (2,0)  (3,0)
localTransition : traffic_rule
zone : time_rule { (1,0)..(1,19) }
zone : general_update_rule { (2,0)..(2,19) (3,0)..(3,19) }
zone : lane_beginning_rule { (0,0) }
zone : lane_end_rule1 { (0,19) }
zone : lane_end_rule2 { (0,18) }
zone : lane_end_rule3 { (0,17) }
in : light car_in out_space
out : in_space car_out
link : light light@segment1a(3,19)
link : in_space@segment1a(0,0) in_space
link : car_out@segment1a(0,19) car_out
link : car_in car_in@segment1a(0,0)
link : out_space out_space@segment1a(2,19)

[traffic_rule]
rule : {0.10+#macro(stop_delay)} 10 { (0,-1)=1 and (0,0)=0 and trunc(#macro(rcell_1))!=0 } %101
rule : {0.20+#macro(stop_delay)} 10 { (0,-1)=1 and (0,0)=0 and trunc(#macro(rcell_1))=0 } % 100
rule : {0.10+#macro(low_delay)} 10 { (0,-1)=2 and (0,0)=0 and trunc(#macro(rcell_1))!=0} % 201
rule : {0.20+#macro(low_delay)} 10 { (0,-1)=2 and (0,0)=0 and trunc(#macro(rcell_2))!=0 } % 2001
rule : {0.30+#macro(low_delay)} 10 { (0,-1)=2 and (0,0)=0 and trunc(#macro(rcell_2))=0 } % 2000
...
rule : {0.30+#macro(high_delay)} 10 { (0,-1)=4 and (0,0)=0 and trunc(#macro(rcell_3))!=0 } %40001
rule : {0.40+#macro(high_delay)} 10 { (0,-1)=4 and (0,0)=0 and trunc(#macro(rcell_3))=0 } %40000

[time_rule]
rule : 9 { #macro(delay) - 10} { fractional((-1,0))!=0 }

[general_update_rule]
rule : { (0,0) } 0 { isUndefined((0,1)) }
rule : { (0,1) } 0 { t }
...
```

FIGURE 14.26 Segment model definition in CD++.

where

Xlist = {φ}; *Ylist* = {(0,0);(0,1)}; η = 2;
$I = <P^x, P^y>$, with P^x = {<X(0,0),light>, <X(0,1),light>}; P^y = {φ};
N = {(0,0), (0,1)} ; *X* ={φ} ; *Y* = {5,6,7,8} for lights;
m =1; *n* = 2; *B* = {φ} ; *C* = $\{C_{ij}/i \in [0], j \in [0, 1]\}$;
Z is defined as Cell-DEVS formal specification; and
select = { (0,0), (0,1)}

The state of each cell is S = {5,6,7,8}, where 5 = green, 6 = yellow, 7 = red, and 8 = standby, and the location transition function is defined as

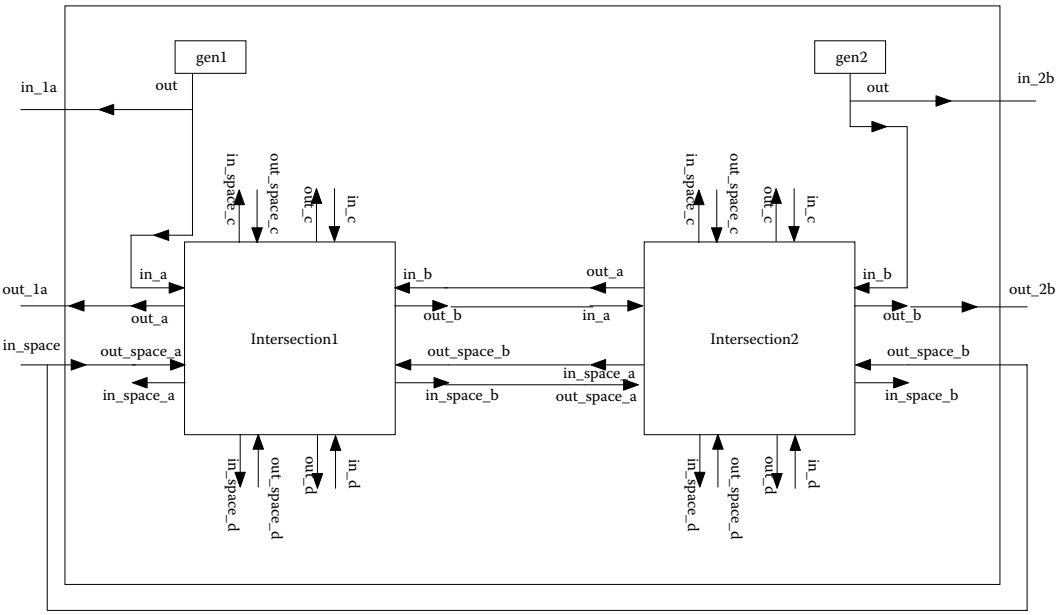


FIGURE 14.27 Two crossings: coupled model definition.

```
rule : {5}   1000  { (0,0)=7 and (0,1)=8} ; change to green
rule : {6}   45000 { (0,0)=5 and (0,1)=7 } ; green cycle
rule : {8}   3000  { (0,0)=6 and (0,1)=7 } ; yellow cycle
rule : {7}   1000  { (0,0)=8 and time > 0 } ; standby
```

The cell changes to green when it is red and the next one is on standby (it takes 1 s to change). Then the green cycle takes 45 s, after which the lights change to yellow, and, 3 s after that, to standby (which will ensure there is no overlapping). The intersection coupled model uses two segments and a traffic light controller, following the description in Figure 14.24.

Figure 14.27 shows how the model (found in *./trafficCrossing.zip*) can be defined in CD++. We use two generators to feed the model with traffic in directions A and B (in the directions of C and D only a small amount of traffic is circulated).

Figure 14.28 shows the simulation results for the segment model. At the beginning of the simulation, there are different vehicles at different speeds and the traffic light is in standby (7 on the fourth row). One second after, the light turns green and the traffic starts moving. At this point, the only vehicle moving is the one with maximum speed. Then, the same car (at cell no. 10) changes its speed level to mid (2), and it is supposed to move to the next cell on the next step. Because only one cell is between this car and the car in front of it (and this is not safe because we need at least two empty cells between them), the car slows down (from 2 to 1). Then the car moves to cell no. 11. In the next figure, we can see a car with a speed of 2 in cell no. 17; from the third row, we know that there is one free space in the intersection or the next lane, plus two more spaces in front of this car. Based on our assumption, we know it is time for this car to speed up.

Figure 14.29 shows the results of the light control model. The fourth row receives the control signals from the light controller, so we can also observe this information on the segment. We can see how vehicles advance with a yellow light, but they stop before the crossing when it becomes red.

14.6 A MODEL OF A CITY SECTION

In this section, we present the definition of a model introduced in Wainer [19] and described in Chapter 3 which is used to study traffic in a section of urban population. This shows an example of

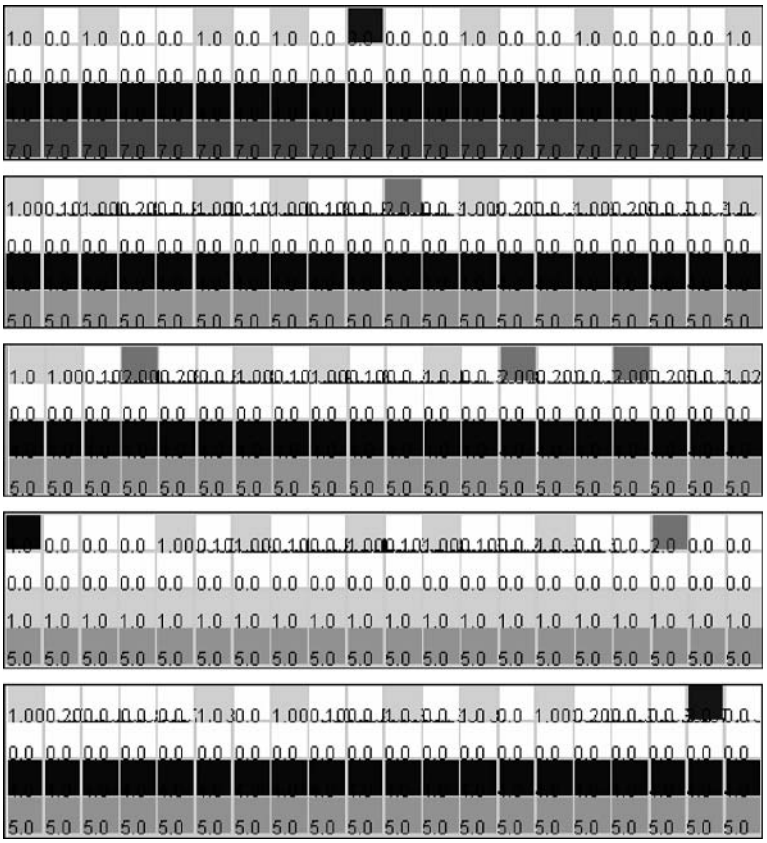


FIGURE 14.28 Two crossings execution.

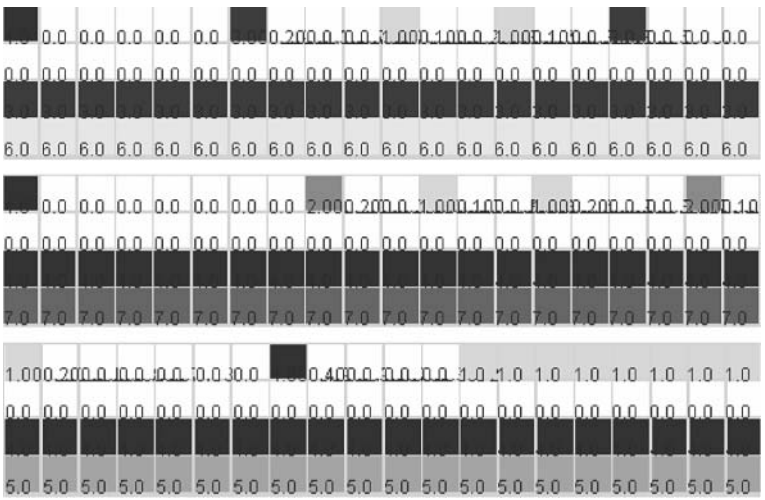


FIGURE 14.29 Two crossings: light control.

a multimodel, with different behavior on each of the components, using different methods on each submodel. The model, depicted in Figure 3.14 in Chapter 3, can be found in *./commercial.zip*.

The full specification of the models can be found in Wainer [19]; here we show the definition of the residential neighborhood, which represents smog diffusion (Figure 14.30). An inertial delay has

```
components : residential
in : in_factory in_highway
link : in_factory in_factory@residential
link : in_highway in_highway@residential

[residential]
type : cell     dim : (9, 10)    delay : inertial     border : nowrapped
neighbors : (-1,1) (-1,0) (0,1) (-1,-1) (0,0) (1,1) (0,-1) (1,0) (1,-1)
in : in_factory in_highway
link : in_factory in_factory@residential(0,9)
link : in_highway in_highway@residential(1,9)
portInTransition : in_factory@residential(0,9) got-exhaust-air
portInTransition : in_highway@residential(1,9) got-exhaust-air
localtransition : diffussing

[got-exhaust-air]
rule : { portValue(thisPort) } 500 {t}
% following rules are same as [diffussing]
...
[diffussing]
rule : 1 500 { ((0,0)=0 and (0,1) = 1) or   ((0,0)=0 and (0,1)=0 and (-1,1)=1 and (-1,0)=1) }
rule : 0 500 { (truecount = 1) or  ((0,0) = 1 and (0,-1) = 0) or
                      ((0,0) = 1 and (0,-1) = 1 and (1,-1) = 0 and (1,0) = 0)      }
```

FIGURE 14.30 Smog in the residential neighborhood.

been used to model the pollution diffusion so that if the wind removes the smog before the delay, pollution does not spread to the neighbors. The model receives inputs on cells (0,9) and (1,9). When a particle arrives through those cells, the *got-exhaust-air* rule is executed. This rule takes the input value arriving through the corresponding input port and makes it the value of the cell. For every other cell, we check whether a smog particle is in a neighboring cell. If the particle remains there during the delay, the particle then moves to the present cell. Otherwise, the cell remains unchanged.

Figure 14.31 shows some of the rules used to model traffic in model B. The first set of valid movements shows the case in which no vehicle is in the cell and a car is coming from the S. The second one shows the case where the cell is empty and a vehicle is coming from the west (W). The third case is when a vehicle is in the cell and it is blocked from moving in direction N or E. In any of these cases, the next state for the cell is that there will be a vehicle in it.

The second set of rules considers preconditions for a vehicle to become empty (i.e., a vehicle abandons the cell or a cell remains empty). The first case is when a vehicle is in the cell and the N cell is empty. The second case represents a vehicle that cannot move to the N and the E cell is empty.

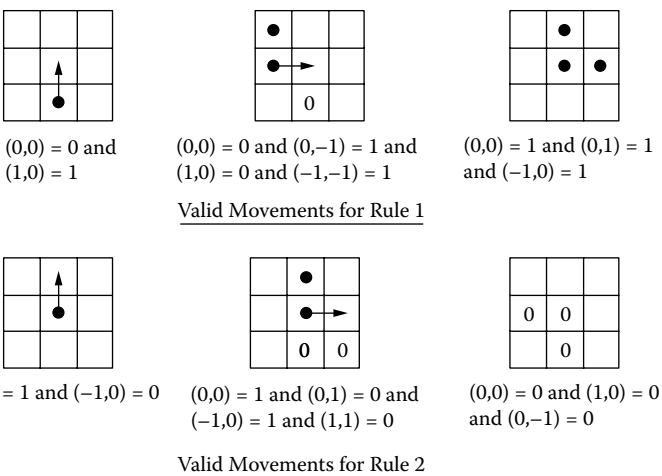


FIGURE 14.31 Valid rules for the commercial neighborhood.

```
Model &Ferry::initFunction() {
        ferryQueue.erase( ferryQueue.begin(), ferryQueue.end() ) ;
}

Model &Ferry::externalFunction( const ExternalMessage &msg ) {
        if( msg.port() == in )        {
                ferryQueue.push_back( 1 ) ;
                if( ferryQueue.size() == 1 )
                        Load_time = Time::Time(0, ignpoi(15), 0, 0);
        }
        if( msg.port() == done )      {
                ferryQueue.pop_front() ;
                if( !ferryQueue.empty() )
                        Load_time = Time::Time(0, ignpoi(15), 0, 0);
        }
        holdIn( active, Load_time );
}

Model &Ferry::internalFunction( const InternalMessage & )  {
        passivate();
}

Model &Ferry::outputFunction( const InternalMessage &msg ) {
        sendOutput( msg.time(), out, ferryQueue.front() ) ;
}
```

FIGURE 14.32 Definition of the ferryboat model.

Finally, we show an empty cell without vehicles to the W or S (thus, no new vehicles will arrive in the cell).

The highway model is similar to the model in the previous section, and the ferry and factory act as queuing servers (as with the many different models discussed earlier), as shown in Figure 14.32. The model uses two input ports and one output port to receive and transmit vehicles, which are queued waiting for service from the ferry. Initially, the queue is empty. When we receive a new request (*in* port), we then queue it. If it is the only element in the queue, we start processing it and schedule a transition. When this time is consumed, the first element in the queue is transmitted. If we receive an event through the *done* port, it means that we can send one more vehicle, and we do it by taking it from the queue.

14.7 THE ATLAS LANGUAGE

Based on the Cell-DEVS formalisms, we defined a traffic specification language known as ATLAS (Advanced Traffic Language Specifications). ATLAS enables users to specify the topology and detailed constructions of a city section in high-level descriptions and to carry out microscopic traffic simulation using automatically generated executable models [20–22].

A city section is composed of a set of different constructions representing all kinds of standard elements that can be found in a city landscape. The built-in constructions defined in ATLAS include street segments, parking lanes, crossings (or intersections), traffic lights, traffic signs, railways, and road worksites. The syntax and implementation of these constructions will not be elaborated here; interested readers can refer to [20–22] for an in-depth discussion of the language and compiler.

There are several inherent advantages associated with this technique. Users can concentrate on the traffic problem to be solved, rather than focus on the details of low-level programming. By decoupling the ATLAS language from the underlying simulation environment, users can reduce the learning curve. The language description has been extensively reported in references 20–24.

The general architecture of the language and components is described in Figure 14.33. ATLAS specification language (1) focuses on the detailed specification of traffic behavior from the user's

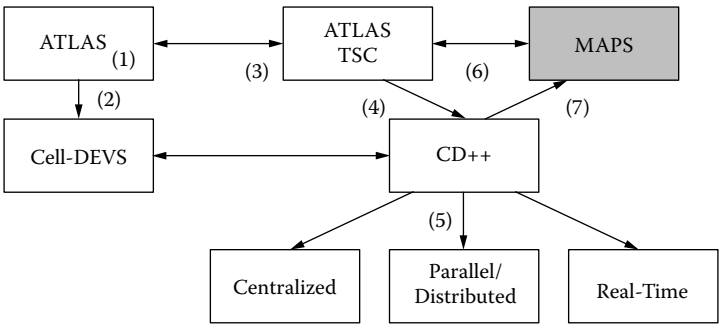


FIGURE 14.33 ATLAS software architecture.

point of view according to the shape of a city section and its transit attributes [20]. A static view of the city section can be easily described, including definitions for traffic signs, traffic lights, etc. The language constructions are formally described using DEVS and Cell-DEVS (2). Based on these specifications, we built a compiler [24], called the ATLAS traffic simulator compiler (TSC). TSC (3) generates code by using a set of templates that can be redefined by the user, easily adapting the generation of behavior to different modeling and simulation techniques. TSC runs on CD++, allowing the users to execute in stand-alone, real-time, or parallel mode (5). TSC is a text-based tool and the system outputs generate text-based log files. The front-end application MAPS converts the constructions defined using the graphical notation into TSC text (6). This allows the user to draw a city section with roads, crossings, and decorations and then parse the drawing to create a valid TSC file (7). Likewise, the output is viewed with three-dimensional graphics [25].

In ATLAS, the structure of a city section is represented by a set of streets connected by crossings [20]. Some of the language components include:

- Segments: They represent sections between two intersections. Every lane in a given segment has the same direction (one-way segments) and a maximum speed. They are specified as: segments = { $(p_1, p_2, n, a, dir, max) / p_1, p_2 \in City \wedge n, max \in N \wedge a, dir \in \{0,1\}$ }, where p_1 and p_2 represent the boundaries of each segment, n is the number of lanes, and dir represents the vehicle direction. The parameter a defines the shape of the segment, and max is the maximum speed allowed.
- Crossings: They represent the places where the streets (represented as sets of segments) are gathered. Each crossing can connect any number of segments. They can be defined as crossings = { $c / \exists\, t, t' \in segments \wedge t = (p_1, p_2, n, a, dir, max) \wedge t' = (p_1', p_2', n', a', dir', max') \wedge t \neq t' \wedge (p_1 = c \vee p_2 = c) \wedge (p_1' = c \vee p_2' = c)$ }.
- Traffic lights: Crossings with traffic lights are defined as TLCrossings = { $c / c \in$ crossings }. Every $c \in$ *TLCrossings* is a set of models representing the traffic lights in an intersection and the corresponding controller. Each of these models is associated with a crossing input. It sends a color value related with the traffic light to the corresponding segment in the intersection.
- Railways: They are built as a sequence of level crossings overlapped with the city segments. The railway network is defined by RailNet = { (station, rail) / station is a model, rail ∈ RailTrack }, where RailTrack = { $(s, \delta, seq) / s \in segments \wedge \delta \in N \wedge seq \in N$ }. *RailNet* represents a set of stations connected to railways, thus defining a part of the railway network. *RailTrack* associates a level crossing with other existing constructions in the city section. Each element identifies the segment that is crossed (s) and the distance to the railway from the beginning of the section (δ). Finally, a sequence number (*seq*) is assigned to each level crossing, defining its position in the *RailTrack*.

- Men at work: They are specified as jobsite = { (s, ni, δ, #n) / s $\in$ segments $\wedge$ s = (c_1, c_2, n, a, dir, max) $\wedge$ ni $\in$ [0, n – 1] $\wedge$ $\delta \in$ N $\wedge$ #n $\in$ [1, n + 1 – ni] $\wedge$ #n $\equiv$ 1 mod 2 }. Here, each (*s*, *ni*, δ, #*n*) $\in$ *jobsite* is related to a segment where the construction work is being done. It includes the first lane affected (*ni*), the distance between the center of the jobsite and the beginning of the segment (δ), and the number of lanes occupied by the work (#*n*). These values are used to define a rhombus where the cars cannot advance.
- Traffic signs: They are defined by control = { (s, t, δ) / s $\in$ segments $\wedge$ $\delta \in$ N $\wedge$ t $\in$ {bump, depression, school, pedestrian crossing, stop, others} }. Each tuple here identifies the segment where the traffic sign is used, the kind of signal, and the distance up to it from the beginning of the segment.

Using these constructions, ATLAS allows the definition of a city section with detail. The *segments* are connected by *crossings* and they define a static view of the model (representing a city map, as seen in Figure 14.34), with implicit dynamic behavior associated. Different decorations can be added, including railways, traffic signs, parking sections, traffic lights, etc.

Once the user creates a model, it can be exported it to TSC format, which can be seen in Figure 14.35. In this case, we have three segments (*t1, t2, t6*) connected through crossing *c1*.

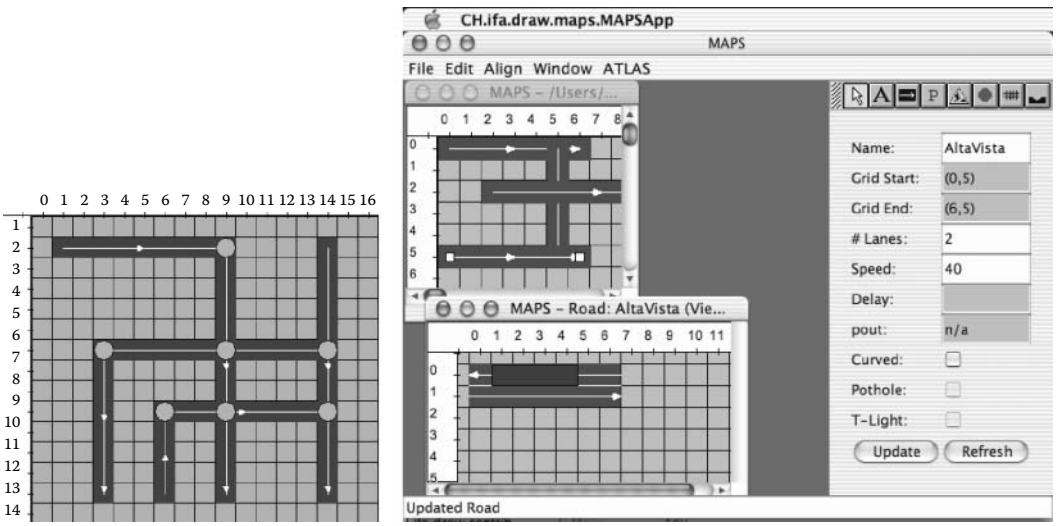


FIGURE 14.34 Defining segments/crossings and decorations in ATLAS.

```
begin segments
  t1 = (1,5),(1,1),2,straight,go,21,1100,parkNone
  t2 = (1,1),(5,1),2,straight,go,22,1200,parkRight
  t6 = (10,8),(10,1),2,straight,back,26,1600,parkLeft
end segments

begin crossings
  c1 = (1,1),11, withoutTL, withHole, 221, 111
end crossings

begin railnets
  rn1 = (t1,1),(t2,1),(t6,2),331
end railnets
```

FIGURE 14.35 Segments/crossings of Figure 14.34 in ATLAS.

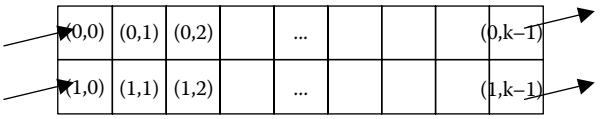


FIGURE 14.36 A two-lane segment.

Segments *t1* and *t2* are one-way, and each has two lanes. One cannot park on *t1,* and *t2* has parking on the right. The two segments are crossed by a railway (*rn1*).

Based on these specifications, we construct DEVS and Cell-DEVS models that represent the model's streets and the vehicle behavior. For instance, for segments with two lanes, we translate the segment $s = (p_1, p_2, 2, a, dir, max)$ into a two-dimensional Cell-DEVS with the structure shown in Figure 14.36. Each row of this space acts as a border of the model. Vehicles in the first row can change to the right, and those in the second row can move to the left. Therefore, each row must be specified separately. The atomic cells in the first row will be defined using a one-lane model like the one presented in Section 14.5. The τ function for these cells also includes the following rules to model lane changes:

τ(N)	N
1	(0,0) = 0 and (0,–1) = 0 and (–1,–1) = 1 and (–1,0) = 1
0	(0,0) = 1 and (–1,1) = 0 and (–1,0) = 0

These rules of lane change consider that a vehicle tries first to move straight and that it has priority to use the position in front of it. The first rule here represents a vehicle arriving in diagonal. To define the priority access, the diagonal movement checks whether a car is waiting to arrive from the cell in diagonal. If that is not the case, it can advance. The function τ for cells in lane 1 is symmetric to this one.

The coupled model corresponding to the segment is defined by

$$TC2(k, max) = < X_{list}, Y_{list}, I, X, Y, n, \{t_1,\ldots,t_n\}, \eta, N, C, B, Z > \quad (14.5)$$

$\boldsymbol{Y_{list}} = \boldsymbol{X_{list}}$ = { (0,0), (1,0), (0,k – 1), (1,k – 1) };

$\boldsymbol{I}$ = <P^x, P^y>, where P^x = {<$X_{\eta+1}$(0,0), binary>, <$X_{\eta+1}$(1,0), binary>, <$X_{\eta+1}$(0,k – 1), binary>, <$X_{\eta+1}$(1,k – 1), binary>};

P^y = {<$Y_{\eta+1}$(0,0), binary>, <$Y_{\eta+1}$(1,0), binary>, <$Y_{\eta+1}$(0,k – 1), binary>, <$Y_{\eta+1}$(1,k – 1), binary>};

$\boldsymbol{X}$ = $\boldsymbol{Y}$ = { 0, 1 }; $\boldsymbol{n}$ = 2; $\boldsymbol{t_1}$ = 2; $\boldsymbol{t_2}$ = k; η = 6; $\boldsymbol{N}$ = { (0,0), (0,1), (1,0), (1,1), (0,–1), (1,–1) };

$\boldsymbol{B}$ = { (0,k – 1), (1,k – 1), (0,0), (1,0) }; and

$\boldsymbol{Z}$ is built using the definition given in the Cell-DEVS formalism.

The interface of this model is composed of the cells of the first and last columns, used to interchange information with each of the crossings. The external ports and the rules for the crossings are extensions of those defined in Figure 14.25.

Based on this specification, we generate CD++ models as in the examples presented in Sections 14.3, 14.4, 14.5, and 14.6. The CD++ specification generated for our example, found in *./citySection.zip,* is as shown in Figure 14.37. When this model executes, we obtain the results found in Figure 14.38. Initially (13), the `t1gen` model generates a vehicle through port `y_t_car0`. The next vehicle will be generated 3 s after that (14). The output message X is translated into an input to `t1` (14), which goes into cell (0,0) in the space (15). After that point, we schedule a car movement in 200 ms. We also add one vehicle to the `bigcounter` model.

```
components : rn11@RailNet t2Cons@Consumer t2 rn10@RailNet t1Gen@Generator t1 rn12@RailNet
components : t6Gen@Generator t6Cons@Consumer t6 c1 rn1@SynchroRailNet
link : y-t-train0bt@rn11 x-vt-train01@t2
link : y-t-train1bt@rn11 x-vt-train11@t2
link : y-t-train0at@rn11 x-vt-train02@t2
...

[t2]
type : cell  width : 4   height : 2  delay : transport   border : nowrapped
neighbors :  (1,-1)   (1,0)   (1,1)  (0,-1)   (0,0)   (0,1)  (-1,-1)  (-1,0)  (-1,1)
in : x-vt-train01  x-vt-train11  x-vt-train02  x-vt-train12  x-c-hayauto00  x-c-hayauto10
out: y-c-room00   y-c-room10   y-co-hayauto03 y-co-hayauto13
link : x-vt-train01 x-vt-train@t2(0,1)
localtransition : t2-segment2-lane0-rule
...
zone : t2-segment2-lane1-rule  {(1,1)..(1,3-1)}

[t2-segment2-lane1-rule]
rule : 1 22 { (0,0)=0 and (0,-1)=1 } ; Coming from the back
rule : 1 22 { (0,0)=0 and (1,-1)=1 and (1,0)=1 and (0,-1)=0 } ; Coming from left with priority
rule : 0 22 { (0,0)=1 and (0,1)=0 } ; Moving forward
rule : 0 22 { (0,0)=1 and (1,0)=0 and (1,1)=0 }; Forward to left lane
```

FIGURE 14.37 Translating Figure 14.35 to CD++.

```
Message * / 00:00:00:000 / top(01) to t1gen(13)
Message Y / 00:00:00:000 / t1gen(13) / y_t_car0 / 1 to top(01)
Message D / 00:00:00:000 / t1gen(13) / 00:00:03:000 to top(01)
Message X / 00:00:00:000 / top(01) / x_ge_car00 / 1 to t1(14)
Message X / 00:00:00:000 / t1(14) / x_ge_car / 1 to t1(0,0)(15)
Message Y / 00:00:00:000 / t1(0,0)(15) / y_t_car_arriving / 0 to t1(14)
Message D / 00:00:00:000 / t1(0,0)(15) / 00:00:00:200 to t1(14)
Message Y / 00:00:00:000 / t1(14) / y_t_car_arriving00 / 0 to top(01)
Message D / 00:00:00:000 / t1(14) / 00:00:00:200 to top(01)
Message X / 00:00:00:000 / top(01) / arrived / 0 to bigcounter(02)
...
Message X / 00:00:00:800 / top(01) / x_t_can_cross2 / 0 to c1(23)
Message X / 00:00:00:800 / c1(23) / x_t_can_cross / 0 to c1(0,2)(26)
Message Y / 00:00:00:800 / c1(0,2)(26) / y_t_room_available / 0 to c1(23)
Message D / 00:00:00:800 / c1(0,2)(26) / ... to c1(23)
Message Y / 00:00:00:800 / c1(23) / y_t_room_available2 / 0 to top(01)
Message D / 00:00:00:800 / c1(23) / ... to top(01)
...
Message Y / 00:00:01:200 / c1(23) / y_t_can_leave0 / 0 to top(01)
Message D / 00:00:01:200 / c1(23) / 00:00:00:000 to top(01)
Message X / 00:00:01:200 / top(01) / x_c_can_cross00 / 0 to t2(04)
Message X / 00:00:01:200 / t2(04) / x_c_can_cross / 0 to t2(0,0)(05)
Message Y / 00:00:01:200 / t2(0,0)(05) / y_c_room_available / 0 to t2(04)
Message D / 00:00:01:200 / t2(0,0)(05) / ... to t2(04)
Message Y / 00:00:01:200 / t2(04) / y_c_room_available00 / 0 to top(01)
Message D / 00:00:01:200 / t2(04) / ... to top(01)
...
Message X / 00:00:04:400 / t2(04) / neighborchange / 0 to t2(0,0)(05)
Message X / 00:00:04:400 / t2(04) / neighborchange / 0 to t2(0,1)(06)
Message X / 00:00:04:400 / t2(04) / neighborchange / 1 to t2(0,2)(07)
Message D / 00:00:04:400 / t2(0,0)(05) / ... to t2(04)
Message D / 00:00:04:400 / t2(0,1)(06) / 00:00:00:200 to t2(04)
Message D / 00:00:04:400 / t2(0,2)(07) / 00:00:00:200 to t2(04)
Message D / 00:00:04:400 / t2(04) / 00:00:00:200 to top(01)
```

FIGURE 14.38 Model execution.

We then show a vehicle arriving at the crossing `c1` (23) through port `x_t_can_cross`. Because there is no room available (`y_t_room_available / 0`), the crossing passivates. Eventually, after there is room available, the car leaves the crossing at 1:200 and leaves into `t2`

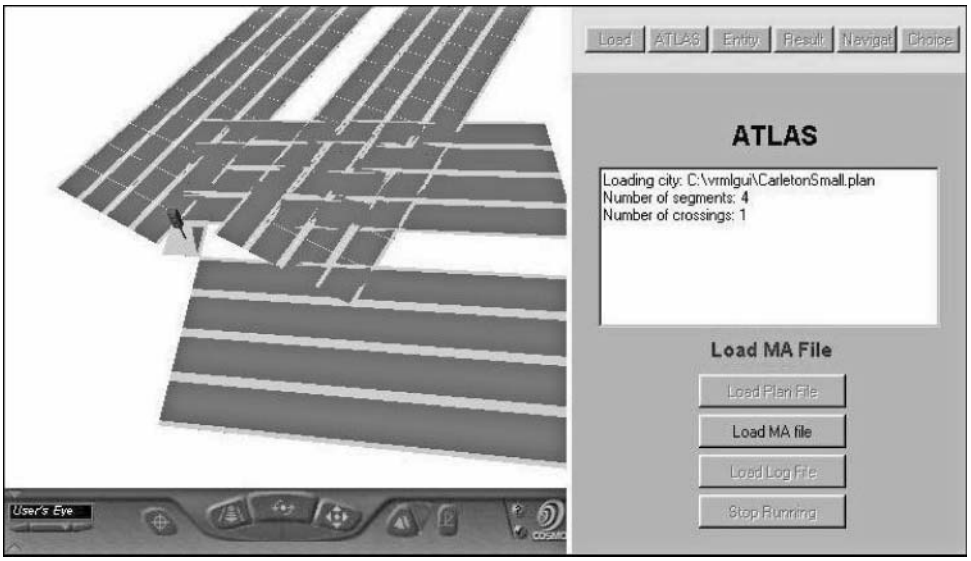


FIGURE 14.39 Three-dimensional visualization: 2 two-way/four-lane segments and crossing with traffic light.

through the port `x_c_can_cross`. The corresponding cell (0,2) then passivates waiting for the following vehicle. 200 ms after that, the vehicle abandons the crossing toward segment `t2`, where the vehicle advances (X messages at 4:400), scheduling a delay related to the vehicle speed.

Figure 14.39 shows the simulation results of this model using an ATLAS three-dimensional visualization applet, which can be executed at http://www.sce.carleton.ca/faculty/wainer/atlas. The model represents traffic at Carleton University, and the corresponding model can be found in *./ATLASCarl.zip*.

14.8 SUMMARY

An urban traffic system consists of a network of roads and intersections on which various types of vehicles go through the system following the rules that reflect specific traffic policies. Although microscopic models require significant input data and computation time to perform the simulation, they can generate very detailed and realistic results and constitute a powerful and versatile tool for traffic analysis.

In this chapter, we introduced the use of DEVS and Cell-DEVS to model problems in traffic, one of the most popular applications of modeling and simulation. We presented a simple model of a traffic controller for a bridge under repair, a toll station for a highway, the intersection of two routes into a highway, and various traffic light controllers. Finally, we introduced the ATLAS modeling language, a high-level language mapped to DEVS and Cell-DEVS that permits defining traffic with generic constructions focusing on the topology of a city section.

The model repository contains numerous other models in this area, including different traffic light models (*./trafficlight.zip*) and toll station models (*./tollStation.zip*). The *./ferry.zip* model introduces a simple model to analyze the congestion of traffic on a ferryboat. The repository also includes models on routing vehicles using origin/destination (O/D) matrixes (*./routingOD.zip*). The*./CityRouting.zip* model defines routing on a city section using an O/D matrix, and *./congest.zip* defines a model that reroutes traffic (using O/D matrix information) in the case of congestion.

REFERENCES

1. France, J., and A. A. Ghorbani. 2003. A multiagent system for optimizing urban traffic. *Proceedings of IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology,* Halifax, Canada.
2. Dresner, K., and P. Stone. 2005. Multiagent traffic management: An improved intersection control mechanism. *Proceedings of AAMAS '05: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems,* Utrecth University, the Netherlands, 471–477.
3. Schmidt, M., R. Schäferr, and K. Nökel. 1998. SIMTRAP: Simulation of traffic-induced air pollution. *Transactions of the Society for Computer Simulation International* 15:122–132.
4. Treiber, M., A. Hennecke, and D. Helbing. 2000. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E* 62:1805.
5. Wagner, P., K. Nagel, and D. Wolf. 1997. Realistic multi-line traffic rules for cellular automaton. *Physica A* 234:687.
6. Maniezzo, V. 2004. CA and roundabout traffic simulation. *Proceedings of Sixth International Conference on Cellular Automata for Research and Industry,* Amsterdam, the Netherlands, LNCS, vol. 3305.
7. Esser, J., and M. Schreckenberg. 1997. Microscopic simulation of urban traffic based on cellular automata. *International Journal of Modern Physics C* 8:1025.
8. Marinosson, S. 2002. Simulation of the Autobahn traffic in North Rhine-Westphalia. *Proceedings of 5th International Conference on Cellular Automata for Research and Industry,* Geneva, Switzerland, LNCS, vol. 2493.
9. Rickert, M., K. Nagel, M. Schreckenberg, and A. Latour. 1996. Two lane traffic simulations using cellular automata. *Physica A* 231:44, 534–550.
10. Chi, S., J. Lee, and Y. Kim. 1997. Using the SES/MB framework to analyze traffic flow. *Transactions of the SCS* 14 (4): 211–221.
11. Chou, H., W. Huang, and J. A. Reggia. 2002. The trend cellular automata programming environment. *Simulation* 78:59–75.
12. Tolba, C., D. Lefebvre, P. Thomas, and A. El Moudni. 2005. Continuous and timed Petri nets for the macroscopic and microscopic traffic flow modeling. *Simulation Modeling Practice and Theory* 13:407–436.
13. Basile, F., C. Carbone, P. Chiacchio, R. K. Boel, and C. C. Avram. 2004. A hybrid model for urban traffic control. *Proceedings of 2004 IEEE International Conference on Systems, Man and Cybernetics,* 1795–1800.
14. Kosonen, I., and M. Pursula. 2007. *HUTSIM.* URL:http://www.tkk.fi/Units/Transportation/HUTSIM/ Accessed: 5/3/2007.
15. Owen, L. E., Y. Zhang, L. Rao, and G. McHale. 2000. Street and traffic simulation: Traffic flow simulation using CORSIM. *Proceedings of WSC '00, 32nd Winter Simulation Conference,* Orlando, FL, 1143–1147.
16. Chopard, B., P. A. Queloz, and P. Luthi. 1996. Cellular automata model of car traffic in two-dimensional street networks. *Journal of Physics A* 29:2325–2336.
17. Barceló, J., E. Codina, J. Casas, J. L. Ferrer, and D. García. 2005. Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems. *Journal of Intelligent and Robotic Systems* 41:173–203, 01/01.
18. Cameron. 1996. PARAMICS—Parallel microscopic simulation of road traffic. *Journal of Supercomputing* 10:25.
19. Wainer, G. 1998. Discrete-event cellular models with explicit delays. PhD thesis, Université d'Aix-Marseille III, France.
20. Wainer, G. 2006. ATLAS: A language to specify traffic models using cell-DEVS. *Simulation Modeling Practice and Theory* 14:313–337.
21. Wainer, G. 2007. Defining a traffic modeling language using cellular discrete-event abstractions. *Journal of Cellular Automata* 2:291–343.
22. Wainer, G. 2007. Developing a software toolkit for urban traffic modeling. *Software Practice and Experiment* 37:1377–1404.
23. Diaz, A., V. Vazquez, and G. Wainer. 2001. Application of the ATLAS language in models of urban traffic. *Proceedings of the 34th Annual Simulation Symposium,* Seattle, WA.
24. Tartaro, M., C. Torres, and G. Wainer. 2001. Defining models of urban traffic using the TSC tool. *Proceedings of Winter Simulation Conference,* Washington, D.C.
25. Wainer, G., S. Borho, and J. Pittner. 2001. Defining and visualizing models of urban traffic. *Proceedings of 1st Mediterranean Multiconference on Modeling and Simulation,* Genoa, Italy.

Section 4

Simulation and Visualization

15 Building DEVS Simulators

15.1 INTRODUCTION

In previous chapters, we have focused on how discrete-event models are specified using DEVS and Cell-DEVS and introduced their implementation using the CD++ toolkit. Until now, we have not discussed details about the simulation engines that drive the execution of these models. We were able to do so, thanks to the separation of concerns in DEVS: the modeler needs to focus only on the models being created, avoiding the details about the simulation engine that drives them.

In this chapter, we present a detailed explanation about the simulation algorithms for DEVS and their implementation in the CD++ simulation engine. The goal is to permit developers interested in the simulation engine to create advanced algorithms (using the open source version of CD++ available at http://sourceforge.net/projects/cdpptoolkit/). The discussion also permits a better understanding of the detailed behavior of the simulation, which can be useful when validating the models.

DEVS simulators are based on the abstract simulation techniques presented in Zeigler, Praehofer, and Kim [1]. These simulation algorithms are guaranteed to execute the hierarchical DEVS specifications correctly. It has been proven that these algorithms are correct to simulate DEVS models. This includes cases of hierarchical composition, individual atomic model execution, and detection of termination conditions (when all the models in the simulation are passive, the simulation can end). In this chapter, we show how different simulation engines can be created based on these algorithms, including a stand-alone version (which is available as an open source project), a parallel simulation algorithm, a distributed simulator, and a real-time engine. Finally, we show how to use wrappers on DEVS simulators to permit operability between different existing DEVS tools.

15.2 THE STAND-ALONE SIMULATOR

The main idea of DEVS abstract simulation algorithms is to create a hierarchy of execution engines based on the modeling hierarchy created by the user (i.e., the hierarchical models presented in previous chapters). We call these entities *Processors.* Atomic/Coupled *Models* define the structure and behavior of the system of interest, while their corresponding *Processors* implement the simulation dynamics (using an abstract mechanism hidden from the models), as sketched in Figure 15.1. This figure shows the different kinds of *Processors: Simulators* are associated with atomic models and *Coordinators* with coupled models. The *Root Coordinator* drives the global aspects of the simulation; it maintains the global time, starts/finishes the simulation (when a termination condition is detected), and is related to the Coordinator of the top-level coupled model (collecting the outputs from it and feeding it with external input events).

Simulation is driven by passing messages among the *Processors;* each represents an event to process. The messages include information about the event origin/destination, the time of the event the message represents, and its content. Four kinds of messages are used:

- * messages signal the occurrence of internal events.
- *X* messages carry information about external input events.
- *Y* messages transmit the model's output events.
- *done* messages carry scheduling information for future events, indicating that a model has finished with its current task.

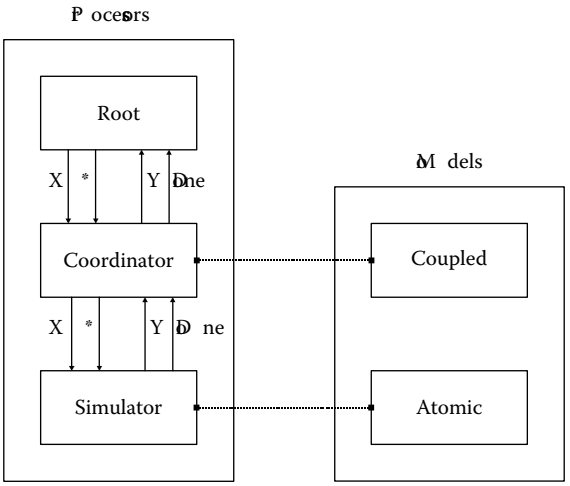


FIGURE 15.1 Relationship between models and processors.

The simulation algorithm we will present uses two variables with scheduling purposes: *TimeLast* (which records the time of the last event) and *TimeNext* (time of the next scheduled event). Coordinators are responsible for routing messages among their children and their parent Coordinators. In addition, they evaluate the minimum *TimeNext* for their children, and they report these values to their parent Coordinators.

The simulator with the smallest *TimeNext* value in the hierarchy is called **imminent** (if there is more than one, the *select* function in the coupled models are used to choose one). The * message must be sent to the imminent Simulator, starting at root and passing through the middle-level Coordinators. Therefore, Coordinators maintain a list including the imminent times for each of their children processors. Each simulation cycle starts when the *Root Coordinator* analyzes the list of external events (i.e., those that must be sent to the *top* coupled model) and the time for the imminent simulator (i.e., the time for the next scheduled internal transition, also called an *imminent time*). The one with the smallest time is chosen; accordingly, a message X or * is sent to the top-level Coordinator.

If a * message is generated, the top-level Coordinator chooses its imminent child and forwards the message. This procedure is repeated by all the intermediate Coordinators until the imminent Simulator is reached. The Simulator first executes the corresponding model's output function λ, which can generate an output event $y \in Y$ (represented as a Y message). Each output message is sent to the parent Coordinator, which queries the *Zij* translation function of its corresponding coupled model to find the model's coupling and (if needed) translates the outputs (Y messages) into inputs (X messages) to the corresponding models. After this, the Simulator activates the internal transition function δ_{int}, producing a state change. Finally, $ta(s)$ is activated, which schedules the next internal transition. This information is carried in a *done* message. The Coordinator receives *done* messages from all its imminent children, and it picks the one with the earliest future time. A *done* message is then created and transmitted to the upper-level Coordinator, carrying this information, which is used to schedule the next internal event for the corresponding coupled model. When this message arrives at the Root Coordinator, the time for the next event is updated, and the cycle starts again.

When an external event $x \in X$ is generated by the Root Coordinator, it is rerouted by the Coordinators using the corresponding coupled model definitions until it reaches the corresponding Simulator. When the message arrives at the Simulator, it activates the corresponding atomic model, triggering the external transition function δ_{ext}. After executing $ta(s)$, the Simulator generates a *done* message (carrying the value of $ta(s)$) for the Coordinator using the scheduling mechanism explained before.

The algorithmic description in Figure 15.2 defines detailed the behavior of the Simulator and Coordinator upon reception of each of the four messages. The Simulator first checks the validity of

```
Simulator(Model:Atomic, m: message <type, source, destination, time, value>)) {
if (m.time < timeLast OR m.time > TimeNext) raise an error;
    e = m.time-timeLast;
    if (m.type = Y or m.type = done or m.type does not exist) raise an error;
    if (m.type = X)           // Execute external transition
        Model.s = Model. ext(Model.s, e, m.value);

    if (m.type = *) {
            if (m.time != timeNext) raise an error;
            send(Y, Model. (s)) to the parent Coordinator; // Execute Output function
            Model.s = Model. int(Model.s);                    // Execute Internal
            transition
         }
      send(done, Model.ta(Model.s)) to the parent Coordinator;
      timeLast = m.time; timeNext=m.time+Model.ta(Model.s);
}

Coordinator(Model:Coupled, m: message <type, source, destination, time, value>)) {
if (m.time < timeLast OR m.time > TimeNext) raise an error;
    if (m.type = X) {         // route X message using the coupling scheme
           m.destination   Z[m.source,m.destination] {
                 m.destination = Query(Model.Zij(m.source));
                 send(X, m.source, m.destination, m.time, m.value) to
                 Model[m.destination];
          }

    if (m.type = *) {
            if (m.time != timeNext) raise an error;
              m.destination   Z[m.source,m.destination] {  // find imminent child
                 find m.destination such that Model.Imminent[m.destination]= m.time;
                 if there is more than one, pick one using the select function;
                 send(*, m.source, m.destination, m.time, m.value) to
                 Model[m.destination];
           }
      }

   if (m.type = Y) {  // Y message going upwards: translate using Zij
            if (m.time != timeNext) raise an error;
              m.destination   Z[m.source,m.destination] {    // Query the Coupled Model
                   m.destination = Query(Model.Zij(m.source));
                   if (m.destination != self)     // internal coupling
                      send(X, m.source, m.destination, m.time, m.value) to
                      Model[m.destination];;
                   else
                     send(Y, m.source, m.destination, m.time, m.value) to
                      Model[m.destination];
           }
  }

 if (m.type = done) {  // done message going upwards: translate using Zij
   Model.Imminent[m.destination] = m.time;
   if all children Processors transmitted done messages {
      m.time=minimum(Model.Imminent[m.destination])   m.destination
      Z[m.source,m.destination]
      send(done, m.time) to the parent Coordinator;
    }
    timeLast = m.time;
    timeNext = minimum(Model.Imminent[m.destination])   m.destination
Z[m.source,m.destination]
}

RootCoordinator(Model:TopCoupled, m: message <type, source, destination, m.time,
m.value>)) {
    if (timeNext = infinity AND no more external events) END;
    if (m.type = Y) save message in output file;
    if (m.type = done)
               timeNext = timeLast+m.time;
               timeLast = m.time;
               send (*, Root, Top, timeNext, m.value);
```

FIGURE 15.2 Coordinator/simulator simulation algorithms.

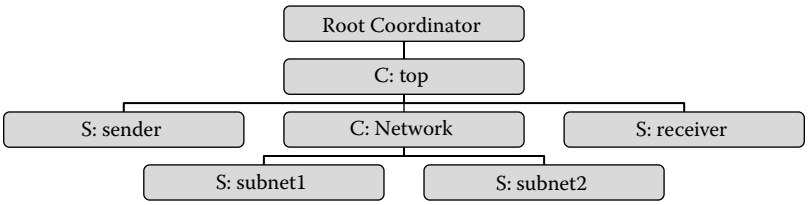


FIGURE 15.3 Alternating bit protocol simulation hierarchy.

the timing information of the message received, and it computes the elapsed time. If an X message is received, the external transition function is triggered. If a * message is received, we first verify that the model is imminent (i.e., the time of the next scheduled event is the current message time). In this case, we execute the output and internal transition functions, as explained earlier. Finally, we send a *done* message and update the time for the last and next events.

The Coordinator first checks that the timing of the message is within the expected scope. If the message is an input (X), we query the coupled model definition to find the model's influencees and reroute the message. If a * message is received, we query the coupled model and pick the imminent model (using the *select* function if needed) and reroute the message. If we receive an upwards Y message, we query the coupling scheme and, if needed, convert the message into an X message for the destination. If the message should be transmitted to the parent model, we just reroute it upward. We then wait for *done* messages from all the newly activated models and save their time in an imminent time list. Finally, when *done* messages are received, we save them in the imminent list. If we received messages from all of them, we pick the one with the smallest time and transmit it upward. We then update *TimeNext* and *TimeLast*. The root Coordinator checks for the termination condition and external messages and starts a new simulation cycle.

We will give an example of execution of the simulation algorithm using the APB model presented in Chapter 13 (Figure 15.3). As discussed in Chapter 13, the model consists of three components: a *sender,* the *network,* and a *receiver* (see Figure 13.41 in Chapter 13). The sender transmits packets with random delays, which are transmitted to the network together with an alternating bit. The network is decomposed further into two subnets corresponding to sending and receiving channels, respectively. The sending and receiving subnets model the transmission latency in each direction. Finally, the receiver consumes the packets and transmits acknowledgment messages. Figure 15.4 shows an excerpt of the simulation log for this model when we use the external event `00:00:10:00 controlIn 20`, which represents a request to transmit 20 messages.

After executing their initialization functions at `00:00:00:000`, all the models become *passive* ("…" represents infinity). Consequently, *done* messages (1–6) are sent to the *top*-level Coordinator (which creates the imminent list with infinite time for each of the children *Processors*, meaning that all the components are passive). At this point, because every single model is passive, the simulation could be terminated. Nevertheless, the *Root* Coordinator checks the external event list and finds the external event mentioned earlier. Therefore, an X message is transmitted to the *top*-level Coordinator (7). This model checks the coupling scheme in the top model definition, and it finds out that inputs in *controlin* at the *top* model should be redirected to the *controlin* port in the *sender* model. Thus, it reroutes the X message (8).

The *sender* simulator now activates the model's external transition function, which programs the transmission of 20 packets. It then schedules an internal transition in 10 s; thus, the simulator creates a *done* message (D) with this value, which is transmitted to the parent (9). At this point the imminent list for the top-level Coordinator contains no other active models in the simulation (all the remaining models are passive), and the next imminent event will occur in 10 s (in the *sender* model). A *done* message with this value is created by the top-level Coordinator and transmitted to *Root* (10).

```
 1.  Message D / 00:00:00:000 / sender(02) / ... to top(01)
 2.  Message D / 00:00:00:000 / receiver(06) / ... to top(01)
 3.  Message D / 00:00:00:000 / subnet1(04) / ... to network(03)
 4.  Message D / 00:00:00:000 / subnet2(05) / ... to network(03)
 5.  Message D / 00:00:00:000 / network(03) / ... to top(01)
 6.  Message D / 00:00:00:000 / top(01) / ... to Root(00)
 7.  Message X / 00:00:10:000 / Root(00) / controlin /     20.00000 to top(01)
 8.  Message X / 00:00:10:000 / top(01) / controlin /     20.00000 to sender(02)
 9.  Message D / 00:00:10:000 / sender(02) / 00:00:10:000 to top(01)
10. Message D / 00:00:10:000 / top(01) / 00:00:10:000 to Root(00)
11. Message * / 00:00:20:000 / Root(00) to top(01)
12. Message * / 00:00:20:000 / top(01) to sender(02)
13. Message Y / 00:00:20:000 / sender(02) / dataout /     11.00000 to top(01)
14. Message Y / 00:00:20:000 / sender(02) / packetsent /      1.00000 to top(01)
15. Message D / 00:00:20:000 / sender(02) / 00:00:20:000 to top(01)
16. Message X / 00:00:20:000 / top(01) / in1 /     11.00000 to network(03)
17. Message X / 00:00:20:000 / network(03) / in /     11.00000 to subnet1(04)
18. Message D / 00:00:20:000 / subnet1(04) / 00:00:02:987 to network(03)
19. Message D / 00:00:20:000 / network(03) / 00:00:02:987 to top(01)
20. Message D / 00:00:20:000 / top(01) / 00:00:02:987 to Root(00)
21. Message Y / 00:00:20:000 / top(01) / packetsent /      1.00000 to Root(00)
22. Message * / 00:00:22:987 / Root(00) to top(01)
23. Message * / 00:00:22:987 / top(01) to network(03)
24. Message * / 00:00:22:987 / network(03) to subnet1(04)
25. Message Y / 00:00:22:987 / subnet1(04) / out /     11.00000 to network(03)
26. Message D / 00:00:22:987 / subnet1(04) / ... to network(03)
27. Message Y / 00:00:22:987 / network(03) / out1 /     11.00000 to top(01)
28. Message D / 00:00:22:987 / network(03) / ... to top(01)
29. Message X / 00:00:22:987 / top(01) / in /     11.00000 to receiver(06)
30. Message D / 00:00:22:987 / receiver(06) / 00:00:10:000 to top(01)
31. Message D / 00:00:22:987 / top(01) / 00:00:10:000 to Root(00)
32. Message * / 00:00:32:987 / Root(00) to top(01)
33. Message * / 00:00:32:987 / top(01) to receiver(06)
34. Message Y / 00:00:32:987 / receiver(06) / out /      1.00000 to top(01)
35. Message D / 00:00:32:987 / receiver(06) / ... to top(01)
36. Message X / 00:00:32:987 / top(01) / in2 /      1.00000 to network(03)
37. Message X / 00:00:32:987 / network(03) / in /      1.00000 to subnet2(05)
38. Message D / 00:00:32:987 / subnet2(05) / 00:00:01:796 to network(03)
39. Message D / 00:00:32:987 / network(03) / 00:00:01:796 to top(01)
40. Message D / 00:00:32:987 / top(01) / 00:00:01:796 to Root(00)
41. Message * / 00:00:34:783 / Root(00) to top(01)
42. Message * / 00:00:34:783 / top(01) to network(03)
43. Message * / 00:00:34:783 / network(03) to subnet2(05)
44. Message D / 00:00:34:783 / subnet2(05) / ... to network(03)
45. Message D / 00:00:34:783 / network(03) / ... to top(01)
46.         ...
47. Message * / 00:00:40:000 / Root(00) to top(01)
48. Message * / 00:00:40:000 / top(01) to sender(02)
49. Message D / 00:00:40:000 / sender(02) / 00:00:10:000 to top(01)
50. Message D / 00:00:40:000 / top(01) / 00:00:10:000 to Root(00)
51. Message * / 00:00:50:000 / Root(00) to top(01)
52. Message * / 00:00:50:000 / top(01) to sender(02)
53. Message Y / 00:00:50:000 / sender(02) / dataout /     11.00000 to top(01)
54. Message Y / 00:00:50:000 / sender(02) / packetsent /      1.00000 to top(01)
```

FIGURE 15.4 Processor's reaction to different messages.

Then *Root* advances the global time in 10 s, and it creates an imminent message (*) with the imminent event time, which is transmitted to the *top* Coordinator (11). The *top* Coordinator knows that the *sender* should be activated at 20:000 (according to the values of the imminent list). At this moment, if two or more models were active simultaneously, *select* would be used to break the tie. All the remaining submodels (*network* and *receiver*) are passive, so it forwards the * message to the *sender* (12).

When the * message is received by the *sender,* it executes the output function. According to the model's definition, we generate two outputs: 11 through *dataout* and 1 through *packetsent*. We also schedule the next internal transition in 20 s (13–15). These three messages are transmitted to the parent Coordinator (*top*), which saves 20 + 20 = 40 in the imminent list entry for the *sender* simulator.

Then the *top* Coordinator queries the coupled model definition, and it finds out that any output messages transmitted through *dataout* should be translated into inputs for *in1* at *network*. Thus, it converts the output message Y into an input X (16). When this packet is routed to the *network* Coordinator, the translation function is queried to find out that the event should be rerouted to the *in1* port in the *subnet1* model. Thus, we reroute the X message (17).

The *subnet1* atomic model then executes the external transition function, and it responds by programming the time advance function to schedule an internal transition in 02:987 s (18). This is the imminent time for the network coupled model (i.e., the one with the smallest time in the imminent list). Thus, we retransmit the message to the parent Coordinator (*top*), which will also determine this is the imminent event (because the sender is scheduled at 40 s and the rest are passive). The *top* Coordinator also queries the coupled model definition to find out that *packetsent* is coupled to the external model (21) and thus reroutes to *Root* (which will save this value in the output file, as shown in Figure 13.43 in Chapter 13).

Root updates the global time, advancing 2:987 s, and then it creates a * message for the *top*-level Coordinator (22–24), which will route it to the *network* (the *sender* imminent time is at 40 s and the receiver is passive), which, in place, will reroute it to the *subnet1* (*subnet2* is still passive). The output function of *subnet1* is executed (25), which will transmit the packet through the *out* port and it will be forwarded to the *top* level through port *out1* of *network* (27). The *subnet1* also executes the time advance function and passivates (26). The *subnet1* and *subnet2* subcomponents are passive, so the coupled model *network* becomes passive, too (28), and a D message with time infinity is sent to the top-level Coordinator to passivate the whole coupled model. The *top* model will now query the coupling scheme to decide what to do with the message received through *out1;* it will determine this message should be transmitted to the *receiver* through the *in* port, and it does so (29). The *receiver* executes its external transition function and schedules an internal transition in 10 s (30–31).

Exercise 15.1

Explain the rest of the simulation results seen in Figure 15.4. Relate the results with those introduced in Figure 13.43 of Chapter 13.

Exercise 15.2

Analyze any of the simulation log files found in the models presented in previous chapters. Analyze the simulation results according to the algorithms just discussed.

Exercise 15.3

Using the model definition for the generator–processor–transducer (GPT) model presented in Chapter 2, apply the algorithm presented in Figure 15.2 and show how the simulation advances.

15.3 IMPLEMENTING SIMULATION ALGORITHMS IN CD++

CD++ was built as a class hierarchy in C++. As seen in previous chapters, atomic models should be programmed and incorporated to the model class hierarchy, and CD++ specification language allows defining the model coupling, Cell-DEVS rules, initial values, and external events. Figure 15.5 shows the main classes in the model hierarchy [2,3].

The two base classes, *Models* and *Processors,* provide the basic constructors for DEVS models. The abstract class *Model* is the root of this subtree. *Model* cannot be used to instantiate objects; other models are derived from this basic class. It is responsible for managing all the input and output ports

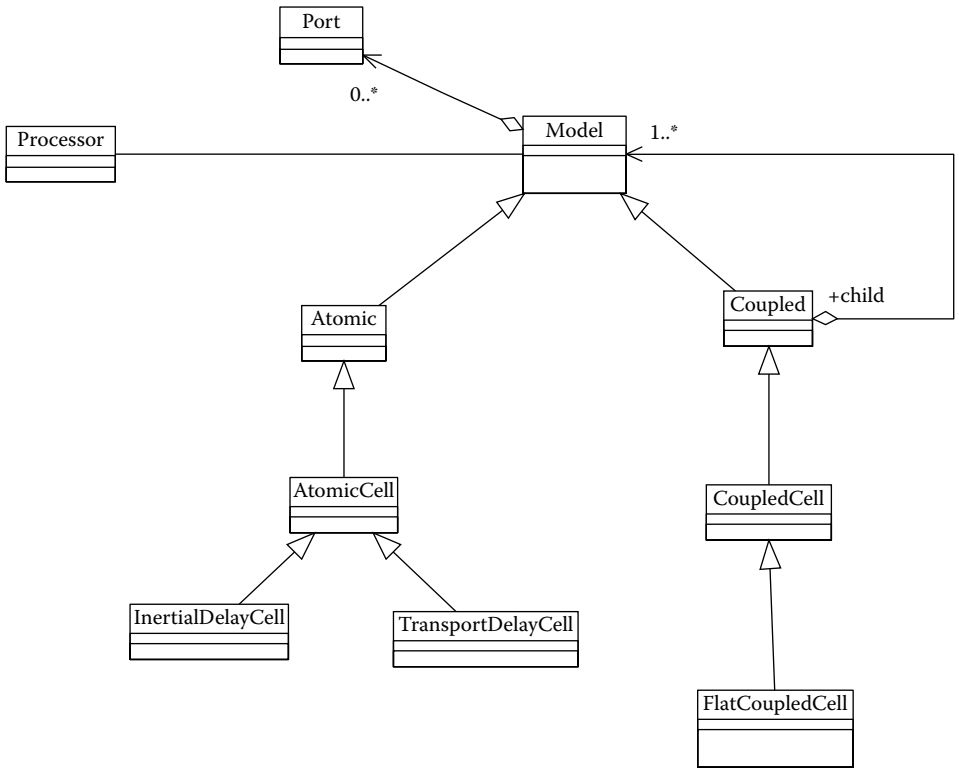


FIGURE 15.5 Basic classes defined by the tool.

(through the methods *addInputPorts, inputPorts, outputPorts*), knowing when the next event is scheduled (using the instance variables *lastChange* and *nextChange*), and knowing its identifier and its parent model (using the instance variables *ID, processor, parent, inportList,* and *outportList*).

The *Atomic* class is an abstract specialization of *Model* that represents the interface of *Atomic* models that we used in previous chapters. As we have seen, in addition to all the functions inherited from *Model,* it provides interfaces for the initialization function, the internal and external transition functions (*internalFunction, externalFunction*), and the output function (*outputFunction*). It also provides methods to query and change the model's state (*state*). These methods should be overloaded to include new models. The class also provides support and activates the time advance function (*holdIn, passivate*) (Figure 15.6).

Coupled is a specialization of *Model* that creates a composite Model (Figure 15.7). To do so, it uses a list of basic models, and it provides the means to manage that list. It is responsible for adding and managing models (*AddModel, ModelList*) and recording the dependencies between them (*addInfluence*). A coupled model is defined by its components (*children*) and the coupling relationship (specified by the instance variables *receivers* and *influences*).

AtomicCell is an abstract class derived from atomic to define Cell-DEVS models. It activates the local computing function using the cell's inputs (arriving from the cell's neighborhood and external models). When an instance of *AtomicCell* is created, the class notifies the neighbor cells about the initial value. The *neighborChange* input and *out* output ports are created. (The remaining I/O ports are created dynamically as needed, and they are stored in two lists named *in* and *output*.) The local computing function is associated with each input port in order to allow the cell to have a different behavior when a value arrives through a port. *TransportDelayCell* and *InertialDelayCell* are nonabstract subclasses of *AtomicCell* used for cells with transport or inertial delays. They overload

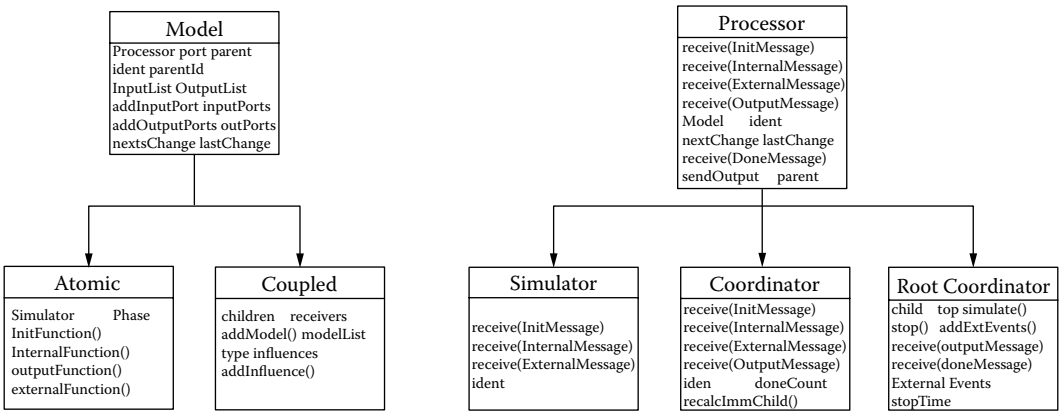


FIGURE 15.6 Basic classes.

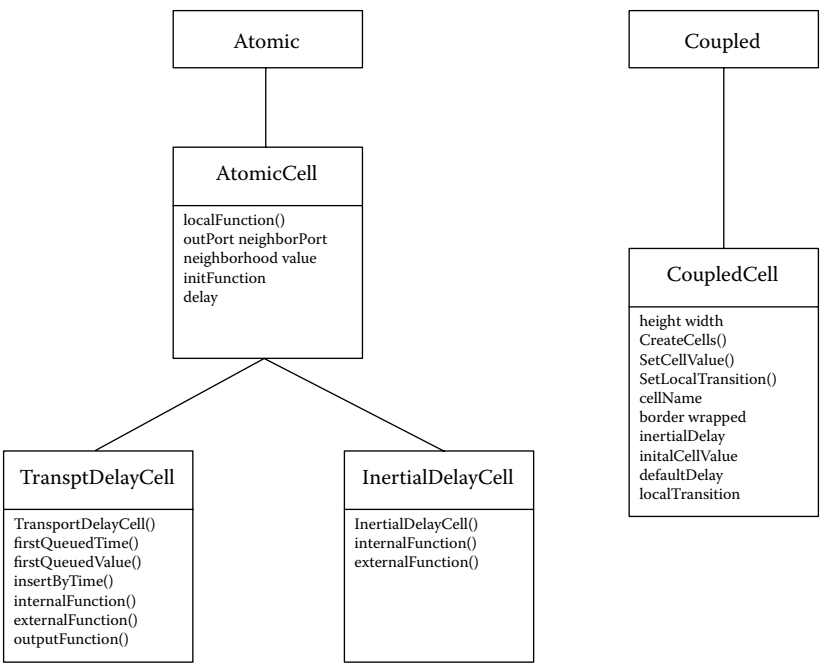


FIGURE 15.7 Atomic/coupled specializations. (From Rodriguez, D., and G. Wainer. 1999. *Proceedings of 31st SCS Summer Computer Simulation Conference,* Chicago, IL, and Barylko, A. et al. 1998. *Proceedings of Applied Modeling and Simulation,* Honolulu, HI.)

the internal transition, external transition, and output functions following the ideas introduced in Chapter 3.

CoupledCell is a specialization of *Coupled,* and it represents Cell-DEVS coupled models. It defines the cell space's dimension and size, the type of delay and border, the initial value for each cell, the local computing function, and zones with alternate behavior. It is also responsible for the creation of the lattice and for linking cells with each other using the neighborhood relationship.

The *Processor*'s subtree shown in Figure 15.8 implements the abstract simulation mechanisms presented in Section 15.2. *Processor* is the basic abstract class, and it is responsible for receiving

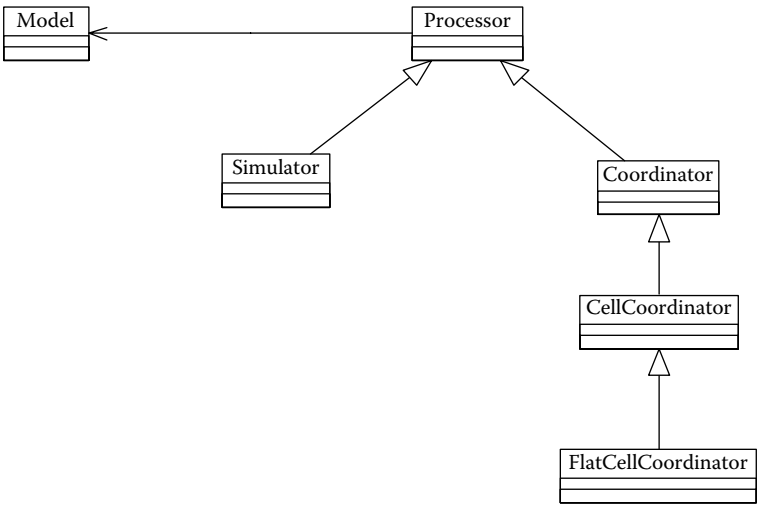


FIGURE 15.8 Processor hierarchy.

messages (*receive*), knowing the associated model and its parent *Processor,* and sending output messages to its parent *Processor* (using the *processor, model,* and *parent* instance variables). In addition to the *receive* method, *Processor* implements three methods:

- *lastChange()* reports the time of the last state change.
- *nextChange()* reports the time of the next scheduled state change.
- *absoluteNext()* reports the absolute time of the next change (*lastChange*() + *nextChange*()).

Simulator is a specialization of the *Processor* capable of executing atomic models. It manages the simulation mechanism for atomic models, receiving messages and activating the associated functions using the algorithms presented in Section 15.2.

Coordinator is also a specialization of *Processor*, and it implements the simulation mechanism for coupled models discussed earlier. The *Coordinator* derives the external messages (*receive(ExternalMessage), imminentChild, Model.influences*) by sending external X messages to the children *Processors*. In addition, it redirects the output messages considering the ports' influences and the coupling scheme (*recalcImmChild*). Finally, it forwards the internal messages to the imminent *Processor* and the initialization messages to all its children *Processors.*

RootCoordinator is also a specialization of *Processor* (only one instance can exist). It is the only *Processor* with no associated model, and it is used to start and finish the simulation and to keep a list of *ExternalEvents* and the global time (*clock*). This *processor* is related to the top level Coordinator (*child, top*). It must generate the model's output (*sendOutput*) and *initialize* the models by loading the next and last event times. It also should reset the external events list and send an initialization message to the highest-level coupled model (*simulate*).

CellCoordinator is a specialization of *Coordinator* for Cell-DEVS models. It is in charge of selecting the imminent model and managing the output messages to avoid duplicated messages being sent to the cells. The message-passing mechanism is different for Cell-DEVS models. The *CellCoupled* class is in charge of managing all the cells as children models. Therefore, when the cells are created, the specified behavior is assigned to them and they are linked with the models defined using the neighborhood relationship and the external coupling lists, as explained in Chapter 3. The *CellCoordinator* class is in charge of receiving internal and external events for these models. When an internal event is received, it will send as many *InternalMessages* as imminent children. Figure 15.9 shows a sequence diagram for the execution scenario for Cell-DEVS models.

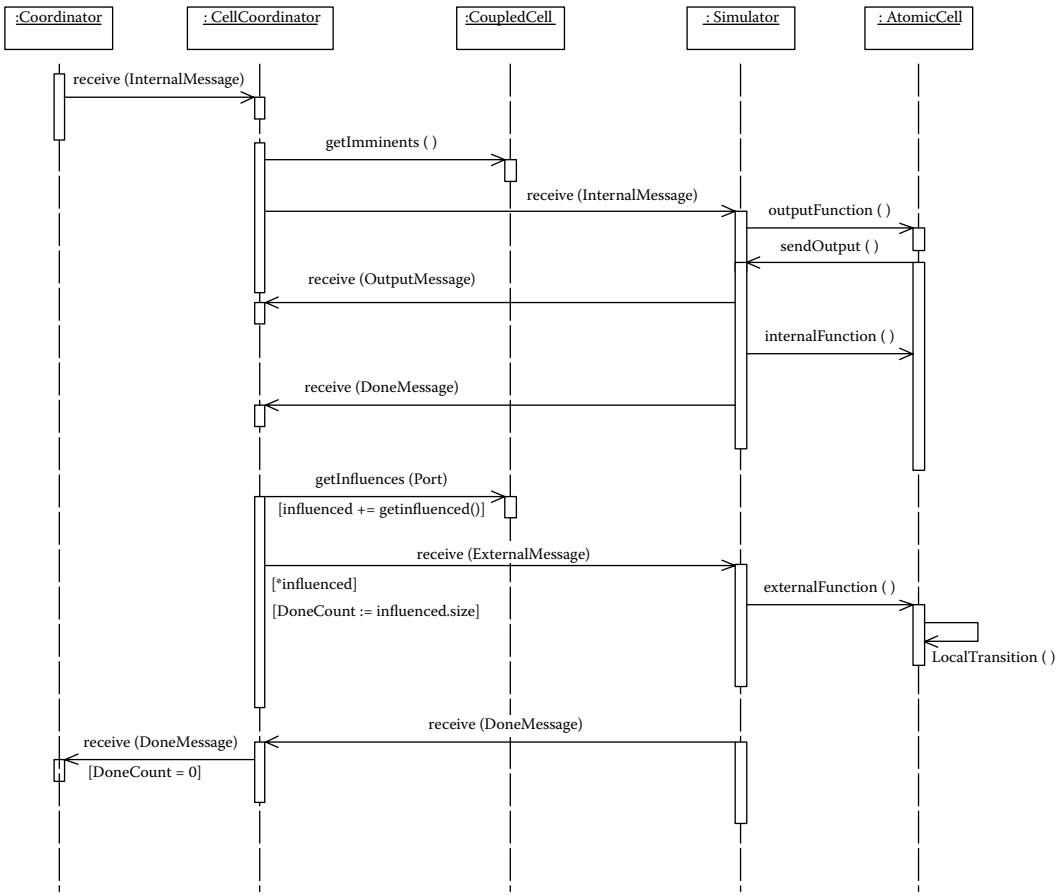


FIGURE 15.9 Interaction between Cell-DEVS messages.

The *CellCoordinator* generates imminent messages for all the imminent cells in the coupled Cell-DEVS based on the imminent list queried by the method *getImminents().* Each of the imminent simulators receives the internal message, and the corresponding atomic model runs the output function of the cell. Output messages are transmitted to the *CellCoordinator,* together with a *done* message defining the next internal transition function. The cell also executes the internal transition. The Coordinator then queries the list of influencees and, if needed, converts the output message into an input for other cell(s), triggering their external transition functions (which execute the local computing function on each cell). Finally, it generates a *done* message for the top-level Coordinator, using the minimum time for all the cells.

15.3.1 Messaging

As discussed in Section 15.2, the simulation advances through message passing. Each message includes information of the source (or destination), the event simulated time, and the content (consisting of a port and a value). Because the message-passing mechanism is encapsulated, the message distribution policy can be changed without affecting the rest of the modules. *Message* is a base class defining the possible messages that can be defined, as can be seen in Figure 15.10.

The *Message* base class defines the data about the model that generated the message and its event time. This is the root abstract class for all messages. *InitMessage* is a subclass that represents the message that the *Processors* receive when the simulation begins. *InternalMessage* corresponds to

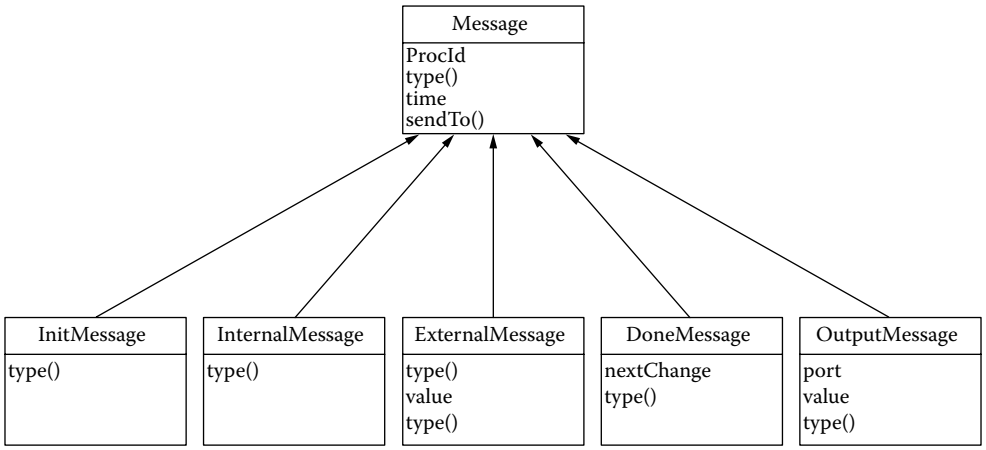


FIGURE 15.10 Message class hierarchy.

the * message in the algorithms presented earlier, and it indicates the imminent time to the imminent *Processor* that the time for an internal event has arrived. *ExternalMessage* represents the arrival of an external event (X). This message includes the input port and its value. *DoneMessage* is received by a *Processor* from one of its children *Processors* indicating the time for the child's next state change. *OutputMessage* represents the output events (Y), and it includes the output port and the value of the event.

As discussed earlier, *Processor* includes the *receive* method, which is responsible for receiving and processing the different simulation messages. The messages are sent among *Processors* through the *MsgAdmin* class, which is devoted to managing the interprocessor message passing, receiving module's invocations, and providing communication between them. This class provides a unique method to send and receive messages, providing a centralized solution that allows encapsulating of the message-passing policy. The sending *Processor* will send the message to the *MsgAdmin* using the *send* method, which will cause the message to be queued until it is sent. Sending a message is done by executing the *receive* method on the receiving *Processor.*

15.3.2 Model and Processor Administration

Models and *Processors* used during the simulation are created when the model description file is read and destroyed when the simulation finishes. For *Models* and *Processors* to be able to reference others, there must be a mechanism capable of getting a reference out of their name. This is the function of the **administrators**, which are depicted in Figure 15.11. The main functions of *ModelAdm* include:

- Creation of new models: creates a model's instance and assigns it a unique identifier. This is the only class that can create new models, and there are different methods to define different kinds of models.
- Association of model's identifiers with existing models: all the existing models are contained in a list maintained by the model's manager and created by the *registerNewAtomic()* method. The manager is in charge of referencing a model by using a unique key.

The *ProcessorAdmin* class manages all the *Processors* participating in the simulation. Because only one instance of this class exists, it must know of all the components of the simulation. This single instance is named *SingleProcessorAdmin*. It is responsible for the creation of all the *Processors* and is capable of retrieving a *Processor* from its identification.

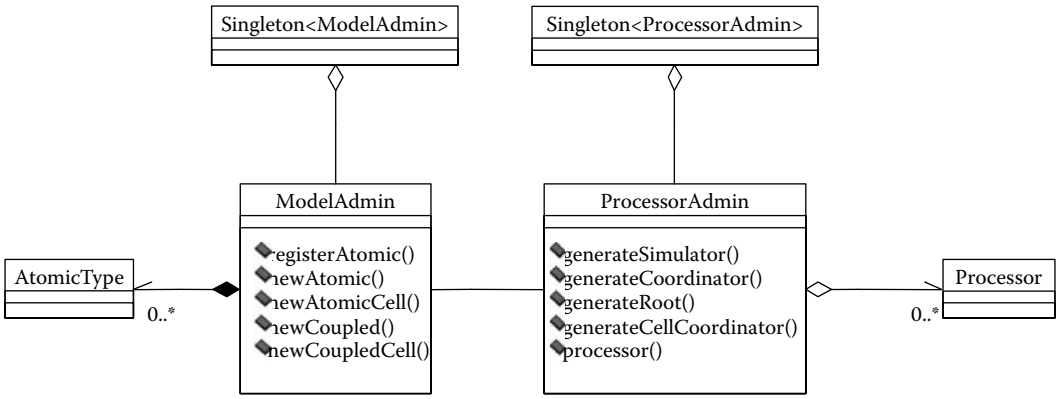


FIGURE 15.11 Class hierarchy for *ModelAdm*.

To start the simulation, the model's specification, the external input events, the time for the end of simulation, and input/output files must be defined. This information is used by the *SimLoader* class, which provides an interface to load the simulator configuration. There are two possible procedures to start the simulation. The first one is by using the class *StandAloneLoader,* which is responsible for loading the parameters by using the shell's command line. The *NetworkLoader* class is responsible for getting the same parameters by using TCP/IP services. In this way, the simulator can be executed as a simulation server, and the parameters can be loaded remotely. The simulator waits on a TCP port if when the model simulation specification is received, it executes the model, and returns the results remotely.

The *SimLoader* class (shown in Figure 15.12) is responsible for loading the model definition and execution option when the simulator is started. *SimLoader* is in charge of parsing the model specification, loading external events stream, and setting the simulation log and output as output streams. The *SimLoader* is used by the *MainSimulator* class during the initialization phase of the simulation. The main method in the *MainSimulator* class is the *run* method, which organizes the activities handled by *MainSimulator.* These include loading the model hierarchy in memory, the initial values of the cells, and the external events, and creating the simulators to execute the model. *MainSimulator* is responsible for the creation of the model tree and for establishing the links between ports using the specification. Once the hierarchy of the model is built, the simulation can begin. To do so, the external events are added, an event list is created, and the stop time is initialized.

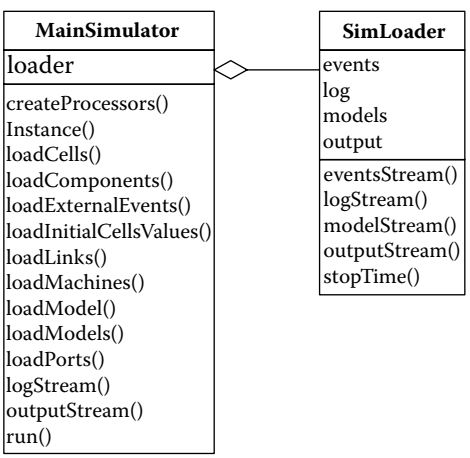


FIGURE 15.12 The *MainSimulator* and *SimLeader* classes.

15.4 INTRODUCTION TO PARALLEL AND DISTRIBUTED SIMULATION CONCEPTS

As simulated systems become increasingly sophisticated, the simulation software becomes larger and more complex. In these cases, the resources provided by a single-processor machine often become insufficient to execute these systems. *Parallel And Distributed Simulation* (PADS) deals with these issues by executing simulations over multiple processors. *Parallel Discrete Event Simulation* (PDES) studies the execution of discrete event models in parallel or distributed computers [4–7]. Parallel and distributed simulation can provide four major advantages [4,5]:

1. *Enabling execution of simulations that otherwise could not be performed.* By executing a large model after subdividing it into simpler, smaller parts, we can improve performance. Distributed environments allow the execution of simulations whose memory requirements exceed the resources available in a single computer.
2. *Geographical distribution.* It is possible to distribute the execution at different physical locations, which is particularly interesting for some applications where data or users are not in the same area.
3. *Integrating simulators based on different platforms.* Simulations can be carried out using different computers, operating systems, and simulation engines.
4. *Fault tolerance.* It is possible to increase the tolerance to failures; if a node fails, a surviving node may take over and continue the execution of the simulator.

According to Fujimoto [4], three major communities are involved in the field of parallel and distributed simulation. The first group is the high-performance computing community, whose main concern is to reduce execution time of applications by using multiple processors. Several synchronization algorithms have been developed by this community. The second group is the defense community, which is mainly interested in integrating separate training simulations to facilitate interoperability and software reuse. The third group is the gaming and Internet community. Its efforts are mostly focused on developing realistic scenarios in distributed environments. We will now give a general description about some topics of interest for the two first communities, with some basic references for the interested reader. Some of these ideas will be used in later sections.

Synchronization is a key issue when executing applications in parallel and distributed environments. Most algorithms organize the simulation components as a group of *logical processes* (LPs). LPs receive and generate time-stamped events or messages to communicate with other LPs, which might execute locally or remotely. The synchronization mechanisms ensure that each LP complies with the *local causality constraint,* which requires that events should be processed in their time-stamp order [4,5]. This guarantees the execution of events in causal order (i.e., guaranteeing that the future does not influence the past).

Two main classes of algorithms for synchronization, such as Chandy–Misra–Bryant [8–10] and Time Warp [11,12], introduced fundamental ideas that are still being applied. *Conservative algorithms* avoid violating causality constraints at all times during the execution of a simulation by processing events in strict time-stamped order. Conservative schemes must arrange for the potential causality errors. This can be done through the provision of *lookahead,* in which each model provides a time in the immediate future up to which it promises not to send input events. The minimum of such blackout times at any model or component, called the lower bound time stamp (LBTS), is the time up to which it can safely process its time-stamped inputs. Thus, simulation proceeds incrementally governed by the lookahead, which is the interval that a model or component adds to its current LBTS to obtain the blackout time sent to other models or components

Optimistic algorithms, on the other hand, allow some violations to happen but provide a mechanism to detect and recover from these situations. To do so, optimistic algorithms permit temporary time-stamped order violation that must be repaired before the final simulation output is presented.

The simulation can advance as quickly as possible, which can produce the reception of out-of-order messages. To rectify this situation, queues of already processed inputs and their outputs are maintained so that the situation can be restored to what it was just before the arrival of the old time-stamped message. Optimistic algorithms have two main advantages over conservative approaches: (1) they enable greater degrees of parallelism, and (2) they do not rely on application-specific data to determine events that are safe to process, which is usually the case in conservative approaches. Nevertheless, a higher level of overhead is involved.

Varied middleware has been used for parallel and distributed simulation efforts, including CORBA (common object request broker architecture) [13], peer-to-peer networks, TCP/IP sockets, MPI [14] and PVM [15] (message-passing interfaces designed for high-performance communication in parallel and distributed environments), Microsoft.Net, and Web Services technologies [16].

The field of defense distributed simulations has been influenced by different efforts; DIS (Distributed Interactive Simulation) [17] and the High-Level Architecture (HLA) [18] are two of the most widely used middleware applications [17]. DIS was created with the goals of being able to use a large number of computers remotely located and sharing data and compute power for advanced simulation exercises in training. The HLA is a standard specifically designed with the goal of reusing legacy simulation systems in distributed environments.

The HLA is a standard for simulation interoperability that provides a set of services that allow federates (individual simulations) to join into a cooperative federation (system of simulations). The simulations can share data (attributes) and events (interactions), and they are time driven [18]. The implementation of the HLA is done using the runtime infrastructure (RTI), a middleware whose interface is standard. Each federate has a local *ambassador* to handle access to the RTI, which must be programmed to permit the RTI to notify the federate of specific events of interest. The baseline of the HLA includes:

- HLA rules, which define the responsibilities and relationships among the components of a federation.
- Interface specification, which specifies the functional interface between federates and the RTI. It defines RTI services and identifies callback functions for each federate.
- The object model template (OMT), which provides a common presentation format for simulation object models and federation object models.

As discussed in Chapter 4, numerous DEVS simulation tools have been implemented using the middleware discussed in this section—for instance, DEVS/CORBA [19], DEVS/HLA [20], DEVSCluster [21], DEVS/Grid [22], and DEVS/P2P [23].The following sections will discuss some of the versions implemented for CD++.

15.5 CD++ PARALLEL SIMULATION ALGORITHMS

As discussed in Chapter 2, the modularity of DEVS makes it possible to separate the model from the simulation mechanism. In this section, we will show how the original abstract simulator mechanism presented in Section 15.2 was revised to run parallel DEVS (PDEVS) models in a parallel/distributed environment. PDEVS [24] was introduced to solve serialization problems with the simultaneous events in classic DEVS (also discussed in Chapter 2). The main difference is that PDEVS processes input bags and generates output bags for the model, and the *confluent transition* **function** (δ_{con}) is activated when internal and external events occur simultaneously.

As with the original definition of the abstract simulator, PDEVS *processors* are specialized into two different engines, *Simulator* and *Coordinator.* Five kinds of messages are used and can be categorized into *synchronization* messages (@, *, and *done*) and *content* messages (y and q) (Figure 15.13):

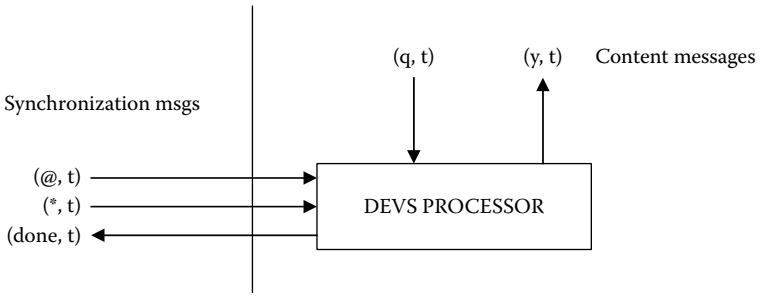


FIGURE 15.13 Messages that a DEVS processor receives and sends.

- *Synchronization messages* are sent from a parent *Processor* to its imminent children. All imminent models' output functions must be executed before any transition function. All outputs are collected and only after they have been sorted the transition functions can be activated. Message @ is used to request all imminent children to execute their output functions and to route the outputs to the corresponding inputs according to the coupling scheme. Message * tells the children to invoke their transition functions (whether it is an external, internal, or confluent transition).
- *Data messages* are sent from parent/children *Processors.* All outputs produced by a model are translated to *y* messages between a child *Processor* and its parent. External messages are sent as *q* messages.

When a simulation is running in distributed/parallel fashion, each CPU will host one or more DEVS *Processors.* Under these assumptions, Coordinator's children need not be executing on the same CPU. Because the correspondence between models and DEVS *Processors* is one to one, every coupled model is associated with only one Coordinator. Therefore, every message sent to children *Processors* running on a different CPU will require interprocess communication. Figure 15.14(a) illustrates this case. A Coordinator sends a message to its eight children distributed on two CPUs (four interprocesses messages are required for the four children running on CPU 1). If the number of children *Processors* is high, the number of messages sent across the network will be significant. This can be avoided if every coupled model has more than one Coordinator. Figure 15.14(b)

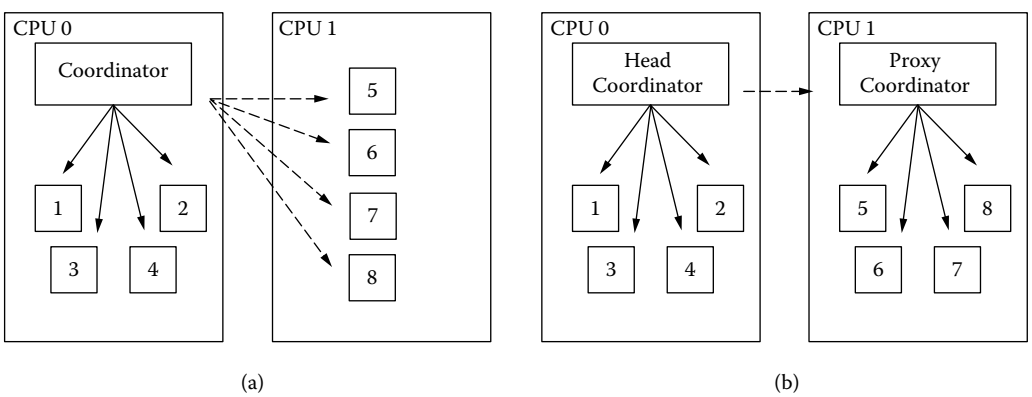


FIGURE 15.14 (a) A single coordinator sending a message to all its children Processors. Dashed lines = interprocessor messages. (b) A head/proxy pair sending messages to all children Processors.

illustrates this case. For the same coupled model, there are two Coordinators—one in CPU 0 and another in CPU 1. In this case, only one message is sent over the network.

Therefore, to reduce interprocessor messages, coupled models use a Coordinator on each CPU where a child *Processor* is running. Children *processors* send messages to a local Coordinator, which will decide how to handle the received messages. Only one of the Coordinators will receive messages from (or route messages to) the parent's model Coordinator. This specialized Coordinator is known as the *head Coordinator* and all other model Coordinators are *proxy Coordinators.*

The simulation algorithm we present here (Figure 15.15) is based on the one presented in Chow, Kim, and Zeigler [25] to simulate parallel DEVS models. We will now proceed to describe the abstract simulator mechanism for the *Simulator, head Coordinator, proxy Coordinator,* and *Root Coordinator.* The *Simulator* is responsible for invoking the atomic model's functions λ, δ_{ext}, δ_{int}, and δ_{conf}, as discussed in Chow et al. [25].

When a simulator receives the message @, it executes the atomic model's λ function and sends the output generated by this function to the parent Coordinator. We also check the message time and send a *done* message to record the current value of $ta(s)$ for the model. Messages q are simply added to the model's input bag for further processing. This will happen when the * message is received; this indicates that a model's transition function must be executed. Which function to execute will depend on the message *time* and the contents of the input bag. If $m.time < timeNext$, then it is not the time for an internal transition, and it must be the case that the bag is not empty and δ_{ext} should be executed. After executing, we empty the bag. If $m.time = timeNext$, it is the time for an internal transition. If no external messages have been received, then δ_{int} is executed; however, if there are external messages in the bag, then δ_{conf} should be called. After executing, the scheduling information is updated and a *done* message is transmitted.

When the *Head Coordinator* receives the message @ from its parent, the imminent child must be synchronized (in order to activate output functions to collect the results of the current simulation cycle before advancing). We retransmit the message @ and put the imminent children in the *synchronize* set, which will be used later to collect the results of the output functions and reroute those results to the corresponding inputs (as discussed in Chow and Zeigler [24]). We wait for all the children to return *done* messages, and the one with the smallest timestamp is transmitted to the parent Coordinator, representing the imminent time for the coupled model. When an input message q is received, it is added to the input bag for the model.

If we receive a transition message * from the parent Coordinator, we first reroute all the input messages. To do so, we take the messages in the input bag and route them using the list of influencees. If it is local, we just reroute the q message to the corresponding processor and cache it in the *synchronize* set (which keeps track of the active components). Otherwise, we find the corresponding remote Coordinator and send the message. The *proxy-sync* set is used to avoid forwarding an output message twice to a *proxy Coordinator.* This is done to reduce the number of messages sent across the network, because a *proxy Coordinator* might be the parent Coordinator for more than one of the influencees of i. If q messages are to be forwarded, then there will be one q message for each influencee of i. Finally, we send a * message to all the members of the *synchronize* set (in order to trigger their transition functions) and wait for their *done* messages. We pick the *done* message with the smallest timestamp and send it to the parent Coordinator.

Finally, when the Coordinator receives an output message y, we need to distinguish two cases:

1. The output message y is received from a child i that is not a *proxy Coordinator.* In this case, we use the translation function Zij to find all the influencees j of child i in order to determine how these outputs should be translated into inputs. If the child i is local, we use the translation function and convert the message into an input message q. If the destination is not local, we need to find the remote Coordinator where j is located (using the *FindRemoteCoordinator* method). Note that instead of forwarding a q message to a *proxy Coordinator,* a different message (y, i) is sent. This is done to reduce the number of

```
Simulator(Model:Atomic, m: message <type, source, destination, time, value>)) {
    case m.type = @
            if (m.time = timeNext) {
                send (Y, Model. (s), m.time) to the parent Coordinator
                send (done, m.time) to the parent Coordinator
            }
            else raise error

    case m.type = q  Add event q to the Model.bag

    case m.type = *
                if (timeLast   m.time < timeNext) {
                     e = m.time - timeLast
                     Model.s = Model. ext(Model.s, e, Model.bag)
                     empty Model.bag
                }
    if (m.time = timeNext  and Model.bag is empty) Model.s = Model. int(Model.s)

    if (m.time = timeNext  and Model.bag is not empty) {
                s =  con(Model.s, Model.bag)
                empty Model.bag
    }
    if m.time > timeNext or m.time < timeLast raise error
    timeLast = m.time
    timeNext = Model.ta(Model.s)
    send (done, timeNext) to parent Coordinator
}

HeadCoordinator(Model:Coupled, m: message <type, source, destination, time, value>)) {
    case m.type @ received from parent Coordinator
         if (m.time = timeNext) {
                timeLast = m.time
                for all imminent children Processors i (those with minimum timeNext) {
                        send (@, m.time) to child i
                        cache i in the synchronize set
                }
                wait until done messages have been received from all imminent Processors
                send (done, m.time) to parent Coordinator
          }
          else raise error

    case m.type q received from parent Coordinator, add event q to Model.bag

    case m.type * received from parent Coordinator
           if m.time < timeLast or m.time > timeNext raise an error
           for all q   Model.bag {
                    for all receivers of q, j in the list of influencees
                           if j is local {
                             send (q, m.time) to j
                             cache j in the synchronize set    }
                           else   {
                             s = FindRemoteCoordinator(j)
                             if s   proxy-sync set {
                                     send (q, m.time) to s
                                     cache s in the proxy-sync and in the synchronize
                                     set  }
                           }
                     clear proxy-sync set
                }

          for all i in the synchronize set send (*, m.time) to i
          wait until all (done, timeNext) are received
          timeLast = m.time
          timeNext = minimum of componentsí timeNext
          clear the synchronize set
          send (done, timeNext) to parent Coordinator

    case message y received from child i (NOT a proxy Coordinator)
                for all influencees j of child i
                        if j is a local processor     {
```

FIGURE 15.15 Abstract simulation algorithms for parallel DEVS models.

```
                                q = Model.z_{i,j} (y)
                                send (q, m.time) to child j
                                cache j in the synchronize set     }
                          else  {
                                s = FindRemoteCoordinator(j)
                                if s   proxy-sync set {
                                    send (y, i, m.time) to s
                                    cache s in the proxy-sync and in the synchronize set
                     }
                                }
                clear proxy-sync set

       case (y, i) message is received from a proxy s
       cache s in the proxy-sync set and proceed as if a y message had been received from
       child i

ProxyCoordinator(Model:Coupled, m: message <type, source, destination, m.time,
       value>)) {
    ...
    case message y received from child i
           sent_to_head = false
            for all influencees, j of child i
               if j is a local processor  {
                                send (q, m.time) to child j
                                cache j in the synchronize set  }
               else if (not sent_to_head){
                                send (y, m.time) to parent Coordinator
                                sent_to_head = true
                     }

               if self   I_i (y is to be transmitted upward) then
               if not sent_to_head send (y, m.time) to parent Coordinator

    case a message (y, i) received from parent Coordinator
               sent_to_head = true
               proceed as if a y message had been received from child i
    ...
    }

RootCoordinator(Model:TopCoupled, m: message <type, source, destination, time, value>)) {
       load queue of external events and sort them by arrival time.
       t = minimum of timeNext of topmost Coordinator and timeNext of queue.
       while t
         if t = timeNext of queue
                for all q in queue with time=t send (q , time) to topmost Coordinator

         if t = timeNext of topmost Coordinator
                send (@, t) to topmost Coordinator
                wait until done is received from it

        send (*, t) to topmost Coordinator
        wait until done is received from it
    }
```

FIGURE 15.15 (continued).

messages sent across the network. A *proxy Coordinator* might be the parent Coordinator for more than one of the influencees of *i*. If *q* messages are to be forwarded, then there will be one *q* message for each influencee of *i*. Instead, just one (*y, i*) message is sent across the network and it will be the responsibility of the *proxy Coordinator* to generate the appropriate *q* messages.

2. If the output message (*y, i*) is forwarded from a *proxy Coordinator* that received *y* from a local child *i,* we cache *s* in the *proxy-sync set* and proceed as if a *y* message had been received from child *i*. When the output events are routed down to children *Processors,* if the message is to be forwarded to a *proxy Coordinator,* the *Z* translation will not be applied. Instead, the original *q* message will be sent. Therefore, care must be taken not to forward a message twice to a *proxy Coordinator.* Here again, the *proxy-sync* is used for that purpose.

The *Proxy Coordinator,* introduced next, differs from the *Head Coordinator* in only one way: When a message needs to be sent to a *processor* that is not local, it will be sent to the *Head Coordinator* (in Figure 15.15, we only show the differences between the Head and Proxy Coordinators). There is no difference in how both *Head* and *Proxy Coordinators* handle a message @. However, for a *proxy Coordinator,* the set of children *Processors* is made by the set of local child *Simulators* and the set of local child *head Coordinators* only.

When an output event is received from a child *i,* the *proxy Coordinator* sorts the message to the influencees of *i*. If any influencee is local, a *q* message is sent. If there are nonlocal influencees, then the output event is sent to the *head Coordinator,* which will then sort the message to other *proxy Coordinators* if necessary. Only one *y* message should be forwarded to the *head Coordinator.* When the *proxy Coordinator* receives an output event that has been forwarded by the *head Coordinator* on behalf of child *i,* it will handle the event as if *i* had been local, but no *y* messages will be sent back to the *head Coordinator* if there is a nonlocal influencee. This is to avoid infinite loops of messages being sent back and forth.

The *Root Coordinator* is responsible for advancing the virtual simulation time and handling external events, which are stored in a sorted queue of events.

Exercise 15.4

Suppose we want to execute the GPT example in three different CPUs. Follow the simulation algorithm presented in Figure 15.15 and analyze the execution of the model using the head–proxy Coordinators.

15.6 FLAT COORDINATORS

As discussed in Kim et al. [26], the hierarchical structure of the simulator (which creates a one-to-one correspondence between model components and simulation objects) can increase the communication costs of message passing. Figure 15.16 shows a sample model with a few components. If the *Root Coordinator* has to schedule an event to lowermost simulators (*#4* and *#5*); the overhead incurred by message passing can be considerable. The same phenomenon is produced if *#5* sends an output to *#3*. The number of intermediate Coordinators can be arbitrarily high, so this overhead can be high. A flat simulator simplifies the underlying structure while keeping the same model definition and preserving the separation between model and simulator [27]. Studies have shown that flat simulators can outperform hierarchical mechanisms [28–31].

We introduced a *flat Coordinator* for CD++ in which there is only one *Processor* in the hierarchy to replace the hierarchical *Coordinators* and *Simulators*. The *flat Coordinator* is in charge of all the tasks for simulating the atomic components, including scheduling and port mapping among its children. The structure of the model is shown in Figure 15.16 using the *flat* coordinator: the resulting hierarchy is simplified as shown in Figure 15.17.

In order to execute the simulation properly, the *flat Coordinator* stores information for the atomic models handled. This includes I/O ports, links, time of next event, and time of the last event

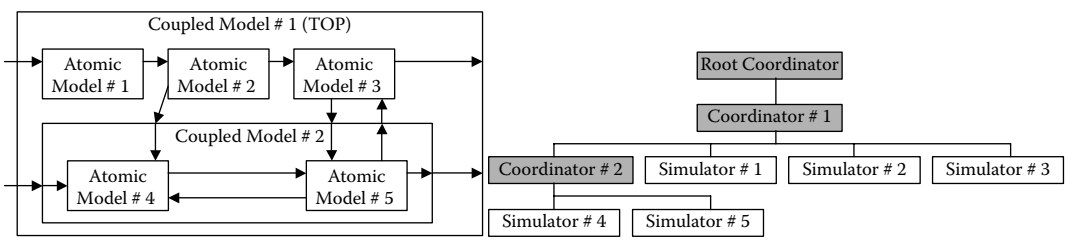


FIGURE 15.16 A sample model and hierarchical structure of the simulator.

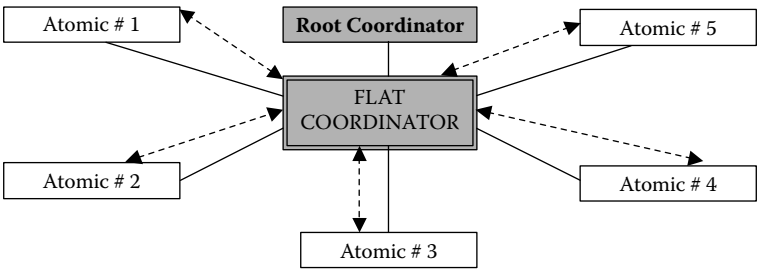


FIGURE 15.17 Flat processors' hierarchy.

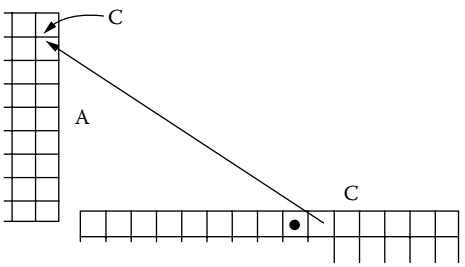


FIGURE 15.18 Model M's initial state.

processed, as well as a queue of pending events. Implementation details about this technique for parallel CD++ can be found in Glinsky and Wainer [28,29].

Wainer and Giambiasi [32] and Wainer [33] introduced a flat simulation algorithm for Cell-DEVS models that reduces the number of interactions carried out in the cell space. We will show the behavior of such algorithms using the example presented in Figure 3.8 in Chapter 3. We first show the hierarchical version. Let us suppose first that in simulated time 10, the model's state is as in Figure 15.18. There is only one active cell in the model: cell [1,9] of model C. At present there is only one message waiting to be processed in the event-list of the Root Coordinator, and the contents of the data structures of the simulators and Coordinators are as shown in Figure 15.19. (The figure does not include the information about each cell's simulator in order to make it easier to read.) The simulation begins when the Root Coordinator creates the following message:

$$< *, \text{Root}, 10 >$$

This message is sent to the M Coordinator, which queries its list of imminent children. There, the Coordinator selects model C (the first of its imminent list) and sends the message <*, M, 10> to its Coordinator. Coordinator C receives this message and selects its imminent child (in this case, cell [1,9]) by consulting its imminent children list. It also verifies that the time of the next event is equal to the simulated time included in the message.

Message <*, C, 10> is sent and arrives at the simulator C_{19}, which will execute the internal transition function. As a first step, the time of next event *tn* is verified. Because it is the same as that of the * message, the arrived message is correct and the output function is executed. If we analyze the model's rules in Chapters 3 and 14, the result of the local computing function is $s' = 0$. Because $s = 1$, the cell's state has changed and its present value should be output. Thus, a Y message is generated with the values < Y, C(1,9), 10, 0 >, and it is transmitted to the parent Coordinator C. After this, the event times corresponding to the Coordinator are updated:

$tl = tn = 10$; and
$tn = tl + D(s) = 10 + \infty = \infty$.

Root Coordinator: **Clock**= 10; **Associated coordinator**: Coordinator M.	Coordinator M **Parent**: Root Coordinator **Children**: { A, B, C, D, E } **Associated coupled model:** {M} **Waiting list**: { } **Imminent child:** { C } ***tn*:** 10; ***tl*:** 0.
Coordinator A **Parent**: Coordinator M **Children**: { A_{11}, A_{12}, ... } **Associated coupled model:** {A} **Waiting list**: { } **Imminent child:** { } ***tn*:** ∞; ***tl*:** 0.	Coordinator B **Parent**: Coordinator M **Children**: { B_{11}, B_{12}, ... } **Associated coupled model:** {B} **Waiting list**: { } **Imminent child:** { } ***tn*:** ∞; ***tl*:** 0.
Coordinator C **Parent**: Coordinator M **Children**: { C_{11}, C_{12}, ... } **Associated coupled model:** {C} **Waiting list**: { } **Imminent child:** { [1,9] } ***tn*:** ∞; ***tl*:** 0.	Coordinators D,E **Parent**: Coordinator M **Children**: { } **Associated coupled model:** {D} ({E }) **Waiting list**: { } **Imminent child:** {} ***tn*:** ∞; ***tl*:** 0.

FIGURE 15.19 Initial contents of simulators' and coordinators' data structures.

The Y message is received by the Coordinator C, which queries its coupling scheme. Because this message is local to the C model, it should be translated into an X message and sent to the following cells: (1,10), (1,8), (2,8), (2,9), (2,10), (4,8), (4,9), and (4,10). The <X, C(1,9), 10 > is sent to each of these cell simulators, and the cell numbers are added into the Coordinator's waiting list.

In each simulator, the message time is checked to see whether it is between the last event and next event times. This is correct, so the elapsed time and the event times are updated:

e = t – *tl* = 10 – 0 = 10, where t is the present time;
σ = *tn* – e = 10 – 10 = 0;
tl = *tn* = 10; and
tn = *tl* + D(s) = 10 + 2 = 12.

Then each simulator executes the local computing function and creates a message < done, C(i,j), 12 > (because the transport delay is of two time units in all of them). When these messages are received, the Coordinator eliminates the children from the wait-list and adds the message in its imminent list. In this case, all the messages have the same time, and the select function is used to choose the imminent children (messages are queued by using this criterion):

Imminent children: { (1,10), (1,8), (2,8), (2,9), (2,10), (4,8), (4,9), (4,10) }

When the waiting list is empty, all the influencees have finished with the execution of the external transition function. Therefore, the smallest imminent child is ready for execution. The first element of the imminent children list is chosen and the message < done, C(1,10), 12 > is sent to the Coordinator M. In this way, if more than one imminent model exist in the hierarchy, one of them is chosen. These procedures are repeated for each of the imminent children of Coordinator C.

When we execute the output function on cell (1,10), we generate a message < Y, C(1,10), 12, 1 > sent to the Coordinator C. This message is transmitted to the neighboring cells, and it influences the links that are external to the C model. Therefore, it should be retransmitted to the parent Coordinator M to execute the *Zij* function, which will send the change to the other models. When the message is received by the M Coordinator, it queries the external coupling function and determines that the

coupling $Y(1,10)_C \rightarrow X(2,10)_A$ should be used. Therefore, it generates the following message: < X, A(2,10), 12, 1 >. The simulation process continues.

If we use a flat simulator, we use a *Next-Events* list to record the next scheduled events and a *New-state* list to record the cells that have changed (in order to keep the cell space updated). In our previous example, all the *Next-Events* lists for the Coordinators are empty, except for model C:

$$Next\text{-}Events = \{(1,9), 10\}.$$

When the Coordinator executes, it takes the first event of the list and updates the simulation time: $tl = tn = 10$ and $e = 0$. In this case, when the local computing function is executed, the result obtained is *New-state* = 0. Because Cells[1,9].state = 1, the new state should be stored in the *New-state*s list. Therefore, *New-state*s = { (1,9), 0 }. After this, because the cell is not in the *Ylist* and inertial delays are not used, we make the next event time = 10 + 2 and

$$Next\text{-}Events = \{ <(1,10), 12>, <(1,10), 12>, <(1,8), 12>, <(2,8), 12>, <(2,9), 12>, <(2,10), 12>, <(4,8), 12>, <(4,9), 12>, <(4,10), 12> \}.$$

After this, the *Next-Events* list does not include any other events with time 10. Therefore, we can update the cell space with the *New-states* list information by making C[1,9] = 0; *New-states* = {}.

When the next events have been updated, a message < done, 10, C > is sent to the upper-level Coordinator. The Coordinator detects that C is the imminent child; therefore, after the root Coordinator updates the global clock, a * message is sent to the C Coordinator that continues with the simulation.

The simulator cycles, and it makes $tn = 12$, $tl = 10$, and $e = 0$. Cell (1,10) is chosen from the *Next-Events* list. In this case, *New-state* = 1, and Cells[1,9].state = 0; therefore, *New-states* = { <(1,10), 1> }.

The cell is in the *Ylist* for the model; therefore, a Y message is created and sent to the Coordinator M. The Coordinator will react in the same way explained for the hierarchical models. When message < Y, C(1,10), 1, 12 > arrives at the M Coordinator, it is translated into the message < X, A(2,10), 12, 1 > that will be transmitted to the flat Coordinator A. This Coordinator will insert it into the *Next-Events* queue.

15.7 IMPLEMENTATION OF DISTRIBUTED DEVS SIMULATION ALGORITHMS IN CD++

The algorithms presented in the previous section were implemented on a Beowulf cluster [27,34] and in a distributed environment using Web Services [35]. This section introduces the implementation of the algorithms for DEVS *Processors* presented in the previous section for simulation in parallel and distributed environments. Figure 15.20 shows the class hierarchy implemented.

In order to implement the PDEVS algorithms, the *Coordinator* receives synchronization and content messages and reacts accordingly using the Coordinator algorithms described in Section 15.5. The message bag associated with the *Coordinator* is processed through the method *sortExternalMessages,* which is invoked at the time of receiving an *internal message (*).* This causes the messages in the bag to be forwarded to their destinations. The method *sortOutputMessages* is invoked whenever a child sends an *output message* to its parent Coordinator. This results either in the message being translated into *external message(s)* sent to the local destination(s) or an *output message* being forwarded upward in the class hierarchy. The *calculateImminentChild* is responsible for evaluating the imminent children *Processors* by examining the minimum time of the next state change.

Figure 15.21 shows the definition of the head and proxy Coordinators, which are implemented by extending the *Coordinator* class and integrating them into the *Processor* class hierarchy. Both override the *receive* method used to process the different messages received by the *Processors.*

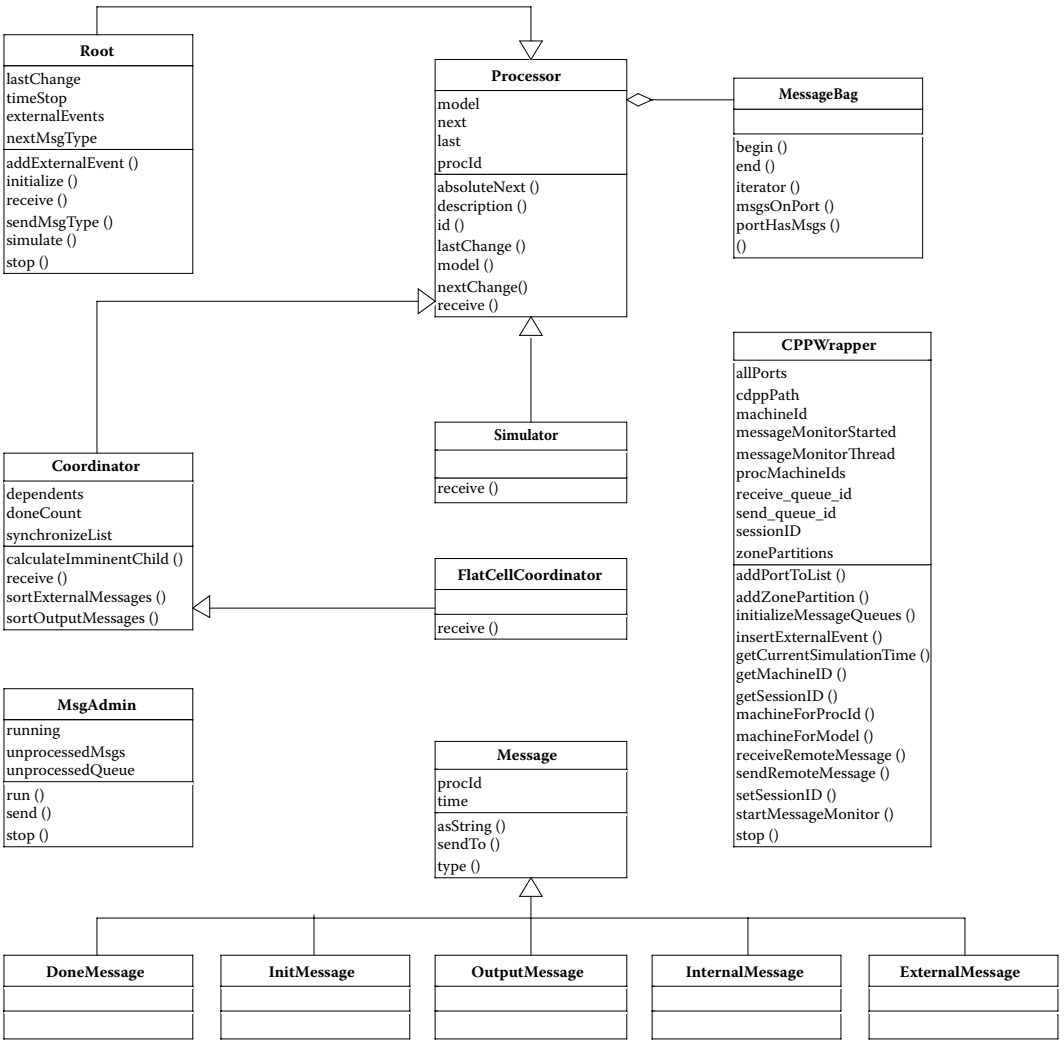


FIGURE 15.20 The simulation class hierarchy.

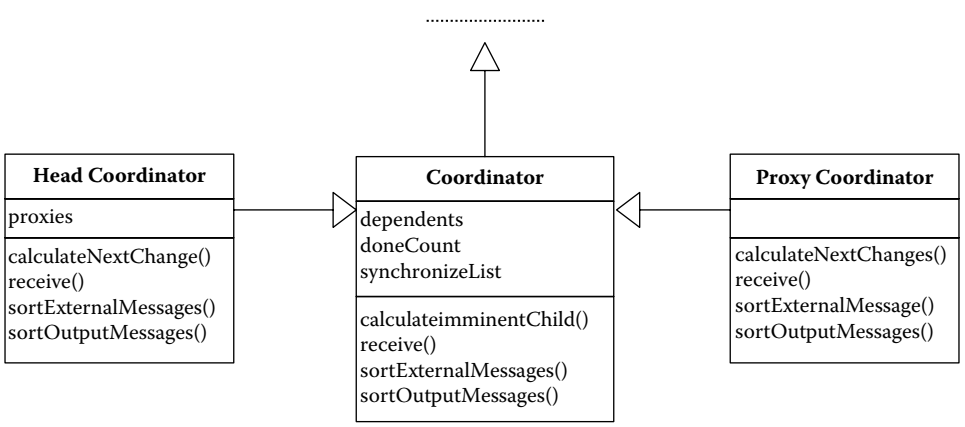


FIGURE 15.21 Head and proxy Coordinator classes.

In addition, they implement the *sortExternalMessages* and *sortOutputMessages*. The *sortOutput Messages* method is triggered when receiving an *output message* from a child *Processor*. The *sortExternalMessages* method is triggered when the Coordinator receives an *internal message* from its parent Coordinator. It causes the Coordinator to process all the messages in its bag by forwarding them to their destinations either locally or remotely. The *calculateNextChange* method is used to evaluate the imminent children *Processors*, and its behavior is different for each Coordinator. In the case of the head Coordinator, it considers the local children *Processors* in addition to the remote proxy Coordinators; in the case of the proxy Coordinator, it only considers the local children *Processors*.

15.8 CD++ REAL-TIME SIMULATOR

Hard Real-Time Systems are highly reactive artificial systems that deliver data from and to devices interacting with the surrounding environment (another artificial/natural system) within tight deadlines (usually ranging in millisecond scales). Because the decisions taken by these applications can lead to catastrophic consequences for assets or lives, correctness and timeliness are critical. Real-time systems' correctness depends not only on the logical results of computation but also on the time at which the results are produced. If a system delivers the correct answer after a certain deadline, it could be regarded as an unsuccessful response. Simulation and real-time systems are related in different ways, mainly:

- Simulation has been used for testing Real-Time systems models; that is, simulations of Real-Time systems are useful for validation and verification of these systems.
- Advanced simulation systems have Real-Time constraints. Simulations with hardware in the loop (for instance, flight or driving simulators embedded in moving platforms, and Live–Virtual–Constructive simulation environments) usually have Real-Time requirements (because the simulator must interact with humans and hardware components within specified deadlines). Such Real-Time simulators are complex Real-Time applications that must handle events in a timely fashion, where timing constraints must be stated and validated.

DEVS simulators (and other simulation tools) have been widely used for validation and verification of Real-Time applications [36]. Recently, DEVS has been used as a framework for Real-Time System construction and validation [19,31,37,38]. Real-Time DEVS [37] helps to expand each model of the system for executing in a real-time environment. CD++ simulation engines have also been modified in order to provide real-time responses [39,40]. These new features allow interaction between the simulator and the surrounding environment, receiving inputs from specialized devices (such as sensors and timers), and providing outputs through ports connected to devices such as motors, transducers, and valves.

In this case, the *Root Coordinator* manages the advance of time along the simulation. This Coordinator must wait until the physical time reaches the next event time to initiate the new cycle. In order to be able to study timeliness of the models, we provide extended facilities. Typically, a model has to react to an external event within a given time to produce an output in order to solve a given problem. For this reason, a way to indicate a deadline time for an external event is provided in the real-time extension of the toolkit. When a model is executed, the simulator is able to check whether the deadlines are met [39,41].

The Real-Time engine of CD++ uses the real-time clock to trigger the processing of discrete events in the system. Figure 15.22 outlines the simulator's architecture. The Root Coordinator manages the interaction with external events (in this case, an experimental frame in charge of testing the model) and returns outputs. The simulator also keeps track of the *number of missed deadlines* and the *worst-case response time* throughout the execution, for further analysis. The *number of missed deadlines*

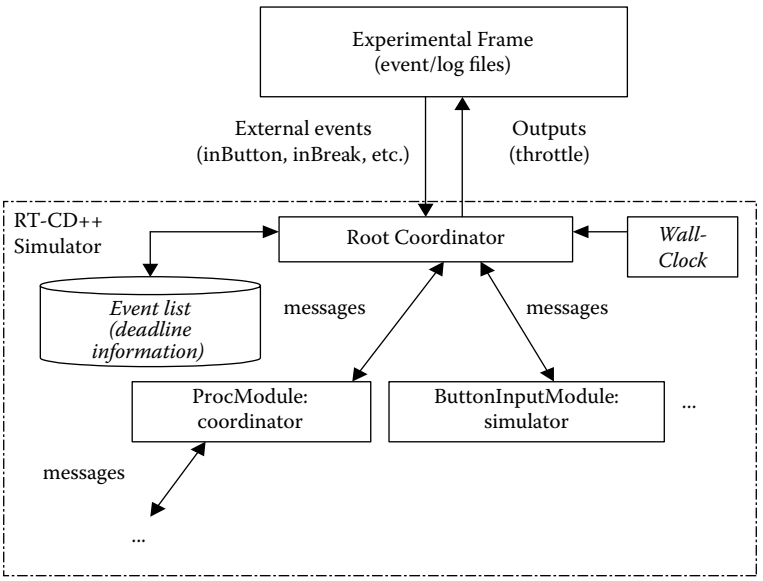


FIGURE 15.22 Real-time simulation in CD++.

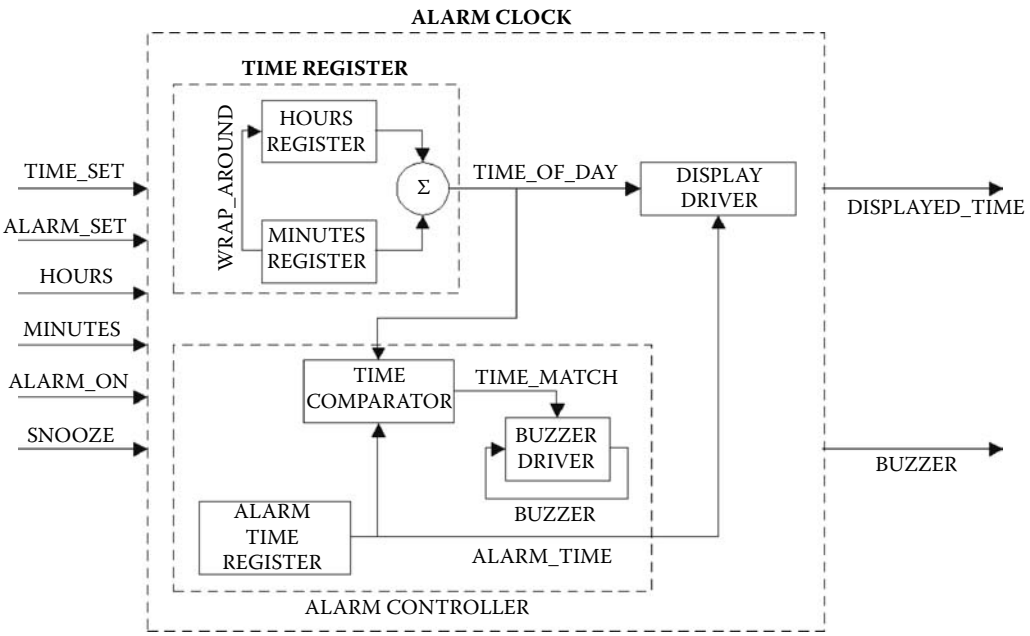


FIGURE 15.23 Alarm clock conceptual model. (From Jacques, C., and G. Wainer. 2002. *Proceedings of Summer Computer Simulation Conference,* San Diego, CA.)

represents the number of deadlines that have been missed along the entire execution of a model. On the other hand, the *worst-case response time* represents the maximum time between the arrival of an event and the output that the model produces in response, in the entire simulation process.

Figure 15.23 shows an example of a simple alarm clock presented in Jacques and Wainer [42] and found in *./alarm_clockRT.zip.* The *ALARM CLOCK* coupled model has six input signals

```
Real time       Message time        Port        Value
01:00:000       01:00:000       DISPLAY_TIME    00:01
02:00:000       02:00:000       DISPLAY_TIME    00:02
03:00:000       03:00:000       DISPLAY_TIME    00:03
...
30:00:000       30:00:000       DISPLAY_TIME    00:30
30:00:000       30:00:000       BUZZER_ON           1
31:00:000       31:00:000       DISPLAY_TIME    00:31
32:00:000       32:00:000       DISPLAY_TIME    00:32
```

FIGURE 15.24 Excerpt from the output file of the alarm clock.

representing the push buttons and switch positions that exist in the real system. *TIME_SET* is used in combination with *HOURS* and *MINUTES* to set the time of day. *ALARM_SET* is used in conjunction with *HOURS* and *MINUTES* to set the desired alarm time. The buzzer will sound if *ALARM_ON* is set at that time. *SNOOZE* stops the buzzer for a period of 10 min, after which the buzzer will automatically sound again if *ALARM_ON* is set. The model also has two outputs: *DISPLAY_TIME* represents the four-digit display while *BUZZER_ON* represents the output of the buzzer speaker. Figure 15.24 is an excerpt from the output file produced by the simulation of this model.

As time passes, the actual time is obtained through the *DISPLAY_TIME* port. Furthermore, the buzzer is turned on at *00:30* and this is notified through the *BUZZER_ON* port. It is important to point out that actual output times are equal to their corresponding message times.

In references 29 and 43–45, the reader can find advanced results for the RT-DEVS simulation engine.

15.9 DYNAMIC STRUCTURE DEVS

In many cases, it is useful to allow the models to adapt to changes in the environment dynamically. As discussed in Chapter 2, dynamic structure DEVS (DSDE) [46–49] allows addressing some of these issues. DSDE divides models into two groups: Basic and Network models. Basic models are atomic structure units that cannot be split. Network models are coupled components composed of multiple basic structure models and interconnections that involve structural changes. A Network Executive is a modified Basic model in charge of conducting structural changes in the network. The Network Executive stores all possible states of structural changes and their corresponding component sets in each structural state [46].

The dynDEVS formalism [48] uses two kinds of dynamic DEVS models: dynDEVS (atomic) and dynNDEVS (coupled). A dynDEVS model can be interpreted as a set of DEVS models with the same interface plus a function (called ρ_α, the model transition function) that determines which DEVS model succeeds the previous one. Agents associated with dynDEVS or dynNDEVS models hold the worldview knowledge of their corresponding models and environments, and the agents are responsible for launching structural changes and conducting the changing process.

Shang and Wainer [50] introduced a simulation algorithm that integrates the dynamic DEVS simulation into CD++. Detailed information about CD++ dynamic DEVS models and their implementation can be found in references [50–52].

Our proposal stems from both DSDE and dynDEVS algorithms. We apply the DSDE formal specifications and parts of the dynDEVS simulation algorithm. In DSDE, a Network Executive conducts the dynamic structural changes. We follow the same idea to provide ground for user-defined model design and simulation (state transition functions, structural transition functions, and output functions). However, we do not attach a Network Executive to a network model. A different mechanism is devised to launch dynamic structural changes and to link regular state transitions of models and structural transitions.

The simulation algorithm uses new message types:

- *sc**: a message to request a structural change from a Simulator to its supervised Coordinator or from a Coordinator to its parent Coordinator (any Simulator or Coordinator can issue this message);
- *sc*: a structural change message sent from a Coordinator to its children, who sent out a request for structural changes, indicating that the children can carry out the structural changes; and
- *start*: an initializing message sent by a Root Coordinator after a structural change; after receiving the *done* messages from the models experiencing structural changes, the Root Coordinator sends the *start* message to start a new simulation phase. This message is used to initialize newly added models and to get the next imminent event time for all the new models.

Figure 15.25 describes the simulation algorithms used in this case.

The Root Coordinator, Coordinator, and Simulator described previously contain extended versions of the regular DEVS simulation engine. The Root Coordinator is able to process the structural change requests and to issue structural change commands. We incorporated the function of the network executive mentioned in DSDE into the two abstract simulators: Coordinator and Simulator. In this way, the dynamic structure algorithm can be integrated into the regular simulation processes. The Coordinator must know all possible states of structural changes and migrations between those states. The structure transition function in the Coordinator is applied to execute those migrations. The structure transition function in the simulator executes structural changes within the associated atomic model. There are three steps in the structural change process:

- *Requests for structural change*: These requests always rise in an internal * message. When receiving a * message, a Simulator evaluates its states or its simulation time. If they are imminent, the Simulator will send a request message *sc** to its parent Coordinator for a structural change. Structural change is a chain of activities, and these activities may span a period. Some changes can be initiated by Simulator but others cannot, such as adding a new atomic model, deleting an existing model, or adding a new link between two of them. For these cases, the corresponding Coordinator (and not the Simulator) launches the structural change.
- *Structural change processing*: Both Simulator and Coordinator perform structural change processes employing structural transition functions, which are introduced especially for dynamic structure simulations. In the Simulator, the structural transition function $\delta_{st}(s, time, e, bag)$ is used to calculate the next structural state of an atomic model. A new state is determined by the current model state (imminent state), the elapsed time since the immediately preceding state, the input bag, and global simulation time. In the Coordinator, the structural change message *done* from the initial model triggers the structure transition function. When a simulation involves multiple levels, the structural change should be executed from the bottom to the top.
- *Structural change end*: At this stage, the simulation returns to the regular DEVS simulation process without losing any unprocessed information. It is under the control of the Root Coordinator. After receiving all *done* messages in response to their corresponding *start* messages, the Root Coordinator knows the time for the next imminent event. Then global time is advanced and simulations are stepped to a new stage.

In Figure 2.7 in Chapter 2, we showed the case of dynamic reconfiguration in an automated manufacturing system. We showed the reconfiguration of the ES workstation due to duty shifts. ES

```
Simulator(Model:Atomic, m: message <type, source, destination, time, value>)) {
// Use Figure 15.15 for @ and * messages

    When receive start message from parent
      initialize the new models and new ports for the models experienced structural
      change;
      timeLast = m.time;
      timeNext = timeLast + Model.ta(Model.s)
       s = s 0 (the initial state of the model)
       Send (done, timeNext) to parent Coordinator

    When receive (sc,m.time) from parent Coordinator
        if m.time > timeNext   or m.time < timeLast   raise error
         e = m.time - timeLast
        Model.s = Model.   sc(s, m.time, e, StruCommand)
        Send (done,m.time) message to parent Coordinator }

Coordinator(Model:Coupled, m: message <type, source, destination, time, value>)) {
 // Use Chow et al. (1994) algorithm for @ message

When * is received
    if m.time > timeNext   or m.time < timeLast   raise error
    // use Chow et al. (1994) for basic processing of *

    wait (done, timeNext) from all i in synchronize set.
    if (sc*, timeNext) received and m.time=minimum  {
         procId = msg.procId;
         capture the associated model (id) into the structured set.
         Send (sc*, timeNext) to its parent Coordinator.}
    Else {
        Wait for (done, timeNext) messages
        timeLast = m.time
        timeNext = minimum of components' timeNext's
         clear the synchronize set
         Send (done,timeNext) to parent Coordinator }

When receive sc message from its parent
   If (timeLast <= m.time <= timeNext) {
     Backup current model set & links supervised by the Coordinator.
     Get the corresponding model in the structured set;
     if (model type is atomic) {
            Send sc to SA;
            Clear the structured set;
            Capture SA into structured set;
            Wait for done;
            When done is received
            Send done to its parent;
    }
    else {
            Send (sc, m.time) to the model;
            Wait for done;
            When done is received   {
                Catch SA into the structured set
                Send (sc, m.time) to SA
                Wait for done
    }
   Update the structure info.

   if m.time<timeLast or m.time>timeNext) raise error

When receive (start,m.time) from parent
   Send (start, m.time) message to i
   ...  // {i|i   D'}. D' is the model set after structural change.
   Wait (done, timeNext) from all components i
   Select timeNext = minimum of components' timeNext's
   Send (done, timeNext) to parent Coordinator
}

RootCoordinator(Model:TopCoupled, m: message <type, source, destination, time, value>)) {
    load queue of external events and sort them by arrival time.
    m.time = minimum of timeNext   of topmost Coordinator and timeNext    of queue.
```

FIGURE 15.25 Simulation algorithm.

```
While m.time
    Send (@, m.time) message to topmost Coordinator
    Wait for done message
    Send (*, m.time) message to topmost Coordinator
    if (sc*,timeNext) received from topmost Coordinator {
        Send (sc, m.time) to topmost Coordinator
        Wait for done message
        Send (start, m.time) to topmost Coordinator
        Wait for (done, timeNext) message
    else
        Wait for (done, timeNext) message
}
```

FIGURE 15.25 (continued).

and ES′ represent the workstation (daytime and nighttime shifts) considered as two structural states of the basic model ς. χ is the network executive. Assume that the working time is 30 min at daytime and 40 min at night. Daytime duty is from 8:00 a.m. to 5:00 p.m. every day. The structural change is implemented as follows:

1. When ES reaches the critical time points t_c = {t| t∈[8:00am, 5:00pm], 5:00pm – t <= 30mins} or {t| t∈[5:00pm, 8:00am], 8:00am – t <= 40mins}, it calculates *tn* for the coming structural change, and it sends a (*sc**, *t*) message to the Top Coordinator.
2. If *tn* is the minimum, the Top Coordinator sends the message (*sc**, *t*) to the Root Coordinator.
3. The Root Coordinator advances the current simulation time to *tn* and issues an (*sc*, *t*) message.
4. The Top Coordinator receives (*sc*, *t*) and sends it to the basic model ς.
5. The Simulator associated with the basic model ς calculates the new structural state using the structural transition function δ_{st}, and a (*done*, *t*) message is sent back.
6. The Root Coordinator sends a (*start*, *t*) message to initialize the new simulation stage. When (*done*, *t*) messages are received, a new regular simulation stage begins.

Figure 2.8 in Chapter 2 showed the workstation PS coupled model, which includes the atomic models Controller, Color, Chrome, and Painter. The atomic model chrome is optional, and painting selection is determined by the controller model. Cars on the conveyor are painted with specific colors; if the current batch of cars on the conveyor needs both color and chrome to be painted, the atomic model Chrome should be added into PS automatically. Thus, there are the two structural states of the network model Θ. When the simulator associated with the atomic model "Controller" detects this change, the following structural change happens:

1. The Simulator associated with the basic model Controller sends the (*sc**, *tn*) message to its parent Coordinator, which is associated with the network model Θ. (Here, *tn* can be the current time, which means the structural change will happen immediately.)
2. The Coordinator retrieves the model-changing list according to the message value in (*sc**, *t*) from the Simulator Controller. The model-changing list should be: (Controller, *tn1*), (Chrome, *tn2*), (Painting, *tn3*). The Coordinator then sends the (*sc**, *t*) message upward.
3. At time *tn1,* when the Coordinator receives the (sc, t) message from its parent, it forwards the message to its corresponding Simulator under its supervision (and also to the Simulators associated with the Controller, Chrome, and Painter because *tn1* = *tn2* = *tn3*).
4. When the Simulators receive the (*sc*, *t*) message, the corresponding structural changes are executed and (*done*, *t*) messages are sent back to the Coordinator.
5. When all structural changes in the Simulators supervised by the Coordinator Θ finish, the Coordinator Θ begins to execute structural changes on its own level. In this case, new links are created among Controller, Chrome, and Painter. A (*done*, *t*) message is returned to the upper level Coordinator.

6. A message (*start*, *t*) is issued by the Root Coordinator once the structural changes finish. After computing the next imminent time *tn,* the simulation time advances to *tn*. Then the Root Coordinator issues a (@*, t*) message and a new simulation phase begins.

15.10 DISTRIBUTED SIMULATION WITH WEB SERVICES

Madhoun, Feng, and Wainer [35] introduced a Web Services–based implementation of CD++, which exposes the functionality of the tool as a Web Service and allows for executing simulations through Web Service technologies. Web Services are group of standards and languages aiming to facilitate developing, publishing, and discovering Web-enabled applications. A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network, using an interface described in a machine-understandable format (specifically Web Service Description Language, WSDL [53]). Client systems interact with the Web Service in a manner prescribed by its description using SOAP messages [54], which are typically implemented using HTTP with an XML serialization in conjunction with other Web-related standards [16].

In order to integrate the Web Service technologies with CD++, a Web Service wrapper was developed to make its functionality accessible by Web Service clients. The Web Service interface performs the following activities:

- receiving the required files, including C++ and header files (in the case of DEVS models), a model definition file, and an external input file, to define the model and execute the simulation;
- executing the simulation, providing the client with the ability to monitor the progress; and
- sending the simulation results to the client, including external output files, simulation logs, and debug information.

The simulation service was split into the *Web Service components* (which handle the Web Service activities) and the *simulation components* (which interact with CD++ by accessing and manipulating its internal objects and data structures). Both parts interact with each other though message queues (through the *WrapperProxy*), as described in Figure 15.26.

The *Web Service components* of the simulation service are compiled into Java archive files and deployed in an Axis server. When the server is started, it loads the *JavaWrapper* (the backbone of

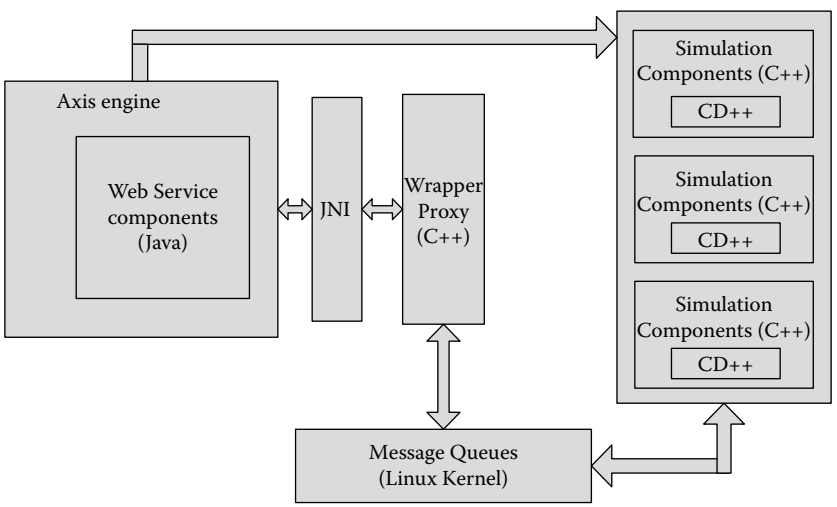


FIGURE 15.26 Implementing the simulation service using JNI and message queues.

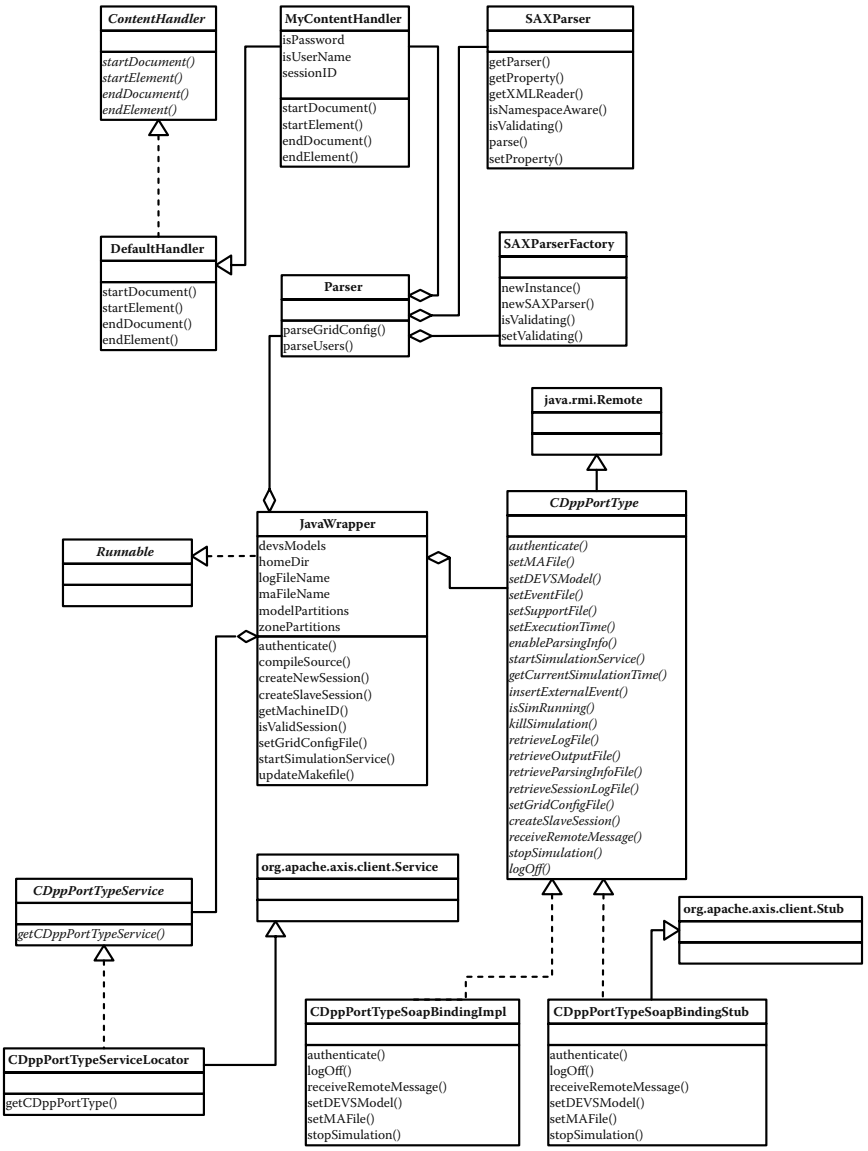


FIGURE 15.27 Web Service components' UML diagram.

the Web Service components presented in Figure 15.27), the server-side stubs, and the client-side stubs. In addition, when the *JavaWrapper* class is loaded, it loads the *WrapperProxy.*

Some of the operations performed by the *JavaWrapper* class include:

- User authentication (method *authenticate*)
- Session initialization: *createNewSession* creates a working space for new sessions, including a *JavaWrapper* instance to handle the newly created session.
- Setting the model definition: The methods *setMAFile, setEventFile, setDEVSModel,* and *setSupportFile* are used for defining the model. The *setMAFile* is used to submit the coupled model definition, *setEventFile* to submit the external events file, and *setDEVSModel* to submit the source and implementation files for DEVS atomic models; *setSupportFile* sets the initial values file for Cell-DEVS models.

- Setting the configuration information for distributed sessions: *setGridConfigFile* is used to send the *grid configuration file*; once the method is executed, it parses the file and save the information contained in it in the *JavaWrapper* instance created for the session.
- Starting the simulation: *startSimulationService* is used to start the simulator. This includes initialization, such as compiling the submitted DEVS models (if any) with the source code of the simulator, sending the model definition to proxy CPUs, and starting the proxy sessions.
- Checking the status of the simulation: The method *isSimRunning* is used to check the status of the simulation process. In addition, *killSimulation* is used to end the simulation process (if needed).
- Retrieving the results of the simulation: *retrieveLogFileName* and *retrieveOutputFileName* are used for retrieving the log and output files, respectively. In the case of running distributed simulations, *JavaWrapper* will utilize the services running on the proxy CPUs in order to retrieve and archive all the log files into one file that can be sent to the user.
- Logging off: The method *logoff* is used to log the current user off and invalidate the session. This method will cause the *JavaWrapper* class to reclaim the resources used by the session and to send messages to the proxy sessions to do the same.

Some of the methods defined in *JavaWrapper* are native methods implemented in C++ that constitute the *WrapperProxy* component of the service (see Figure 15.18) and are implemented as procedures written in C/C++ in order to access Linux message queues. These methods are interfaced to the *JavaWrapper* class using the Java native interface (JNI):

- *initializeNewSession* creates two message queues for each session to act as a communication channel between the Web Service and simulation components of the service.
- *getCurrentSimulationTime* is used to query the simulator for the current execution time.
- *insertExternalEvent* inserts external events in the simulation while the simulation is running.
- *startMessageMonitor* starts the message monitor that keeps checking for messages from the simulator.
- *getMachineID* gets the ID of the CPU running the simulation.
- *machineForModel* returns the ID of the CPU running a particular session of a distributed simulation.
- *sendRemoteMessage* sends remote messages between CPUs in distributed simulation sessions. It takes a message and passes it to the Web Service components to be sent as a SOAP message.
- *receiveRemoteMessageByProxy* receives remote messages when running distributed simulations. It gets SOAP message content from the Web Service components and passes it to the simulator.
- *stopSimulationSession* ends the simulation.
- *addZonePartition* defines Cell-DEVS model partitions.

The client- and server-side stubs are required for the deployment and utilization of the simulation service. The *CDppPortTypeSoapBindingImpl* represents the server-side stub; when the Axis server receives a request from the client in the form of a SOAP message, it does some processing on the message and extracts the attributes necessary to execute the service. Then, it invokes a method in the *JavaWrapper* class corresponding to the operation requested by the client. *CDppPortTypeService* and *CDppPortTypeServiceLocator* are used to locate the Web Service using its Unified Resource Locator (URL). The former is an interface implemented by the latter, and it is usually used at the beginning of any invocation process of the Web Service. *CDppPortTypeSoapBindingStub* is a client-side stub that can be used by the program accessing the simulation service. It is used to access and set up proxy sessions while running distributed simulations.

```
<wsdl:message name="setDEVSModelRequest">
        <wsdl:part name="in0" type="soapenc:string"/>
        <wsdl:part name="in1" type="apachesoap:DataHandler"/>
        <wsdl:part name="in2" type="soapenc:string"/>
        <wsdl:part name="in3" type="apachesoap:DataHandler"/>
</wsdl:message>
<wsdl:message name="setDEVSModelResponse">
        <wsdl:part name="setDEVSModelReturn" type="soapenc:string"/>
</wsdl:message>
<wsdl:message name="isSimRunningResquest"/>
<wsdl:message name="killSimulationRequest"/>
```

FIGURE 15.28 A *message* definition for *SetDEVSModel*.

In order to "consume" the simulation service, we need access to the service interface. This is defined as a WSDL document. WSDL documents usually contain a *type* element to define nonstandard parameter types of the messages exchanged between the Web Service and the client. The *message* element defines the request and response SOAP messages. For instance, Figure 15.28 shows the request and response messages for the *setDEVSModel* operation.

setDEVSModel takes four arguments (through the message *setDEVSModelRequest*): *in0,* the name of the header file defining the DEVS model class; *in1,* a *DataHandler* object representing the file (sent as a SOAP attachment); *in2,* the name of the C++ file containing the class implementation; and *in3,* a *DataHandler* object representing the C++ file. *DataHandler* provides a consistent interface to data available in many different formats; in our case, it represents a file that is serialized by the client into a SOAP attachment and is deserialized to a file on the server side. The *setDEVSModelResponse* message represents the return type for *setDEVSModel,* which is a string stating whether the operation was successful or not.

The *portType* defines a collection of operations, each having an input and output. In this case, the input is the *setDEVSModelRequest* message and the output is the *setDEVSModelResponse* message (Figure 15.29).

The *binding* element defines the binding of the Web Service SOAP messages to an actual protocol (HTTP or SMTP). In addition, it defines the encoding style (RPC/message) and encoding type (encoded/literal). Figure 15.26 shows a partial definition of the binding of the simulation service to HTTP (http://schemas.xmlsoap.org/soap/http). The *binding* element lists the operations implemented in the service with the input and output messages for each one.

The *service* element groups a number of ports together. Each port links a *binding* definition of a specific *portType* to URL to be used to access the service (Figure 15.30). In Figure 15.27, the simulation service binding *SimulationServiceSoapBinding* is linked to the *SimulationService* port,

```
<wsdl:portType name="CDppPortType">
        <wsdl:operation name="setDEVSModel" parameterOrder="in0 in1 in2 in3">
                <wsdl:input message="impl:setDEVSModelRequest"
                        name="setDEVSModelRequest"/>
                <wsdl:output message="impl:setDEVSModelResponse"
                        name="setDEVSModelResponse"/>
        </wsdl:operation>
```

FIGURE 15.29 The *portType* definition of the simulation Web Service.

```
<wsdl:service name="CDppPortTypeService">
        <wsdl:port binding="impl:CDppServiceSoapBinding" name="SimulationService">
        <wsdlsoap:address location="http://localhost:8080/axis/Service/CDppPortType"/>
        </wsdl:port>
</wsdl:service>
```

FIGURE 15.30 The *service* definition of the simulation Web Service.

which in turn is assigned the URL http://localhost:8080/axis/Service/SimulationService, used by the clients to access the simulation service.

Further details about the Web-services version can be found in references 35, 55, and 56.

15.11 INTERFACING DEVS SIMULATORS: CD++ AND DEVS C#

DEVS C# is a DEVS engine created in the University of Arizona's ACIMS laboratory and programmed in C# .NET [57]. In DEVS C#, models are written as C# classes that extend the atomic class. A DEVS C# model consists of input and output ports, a constructor, an initialization function, internal and external transition functions, and an output function. Similarly, the DEVS C# atomic class provides to CD++ a set of services to all models extending the class. The `hold(time)` function programs the time advance function. `TimeNext` and `TimeLast` get and set the time of the next event and the time of the previous event, respectively. The `TimeCurrent` property gets and sets the current time. Figure 15.31 is the code for a simple model representing a timer.

The preceding C# class extends the base type Atomic. For this simple case, the constructor is empty. For more complex models, member variables may be set and helper functions can be invoked. In the *init* function, *hold(time)* is called, specifying when the first internal transition will occur (in 3.3 s). The external transition function is empty in this case (in this model, state change only occurs due to timer expiration). The internal transition function calls the *hold(time)* function, setting the time until the next internal transition. The output function displays the time of the alarm, utilizing *TimeCurrent* to get the current time.

A split simulation is a source system whose components have been broken into two or more groups prior to execution. These groups of components (component groups hereafter) will run under separate simulators that may or may not be implemented using the same simulation engine (CD++ or DEVS C#, specifically). Different mechanisms have been used for making these simulators interact. One way is to split the execution of the Coordinators/simulators among multiple *Processors* within a single simulation engine. This is the approach taken by DEVS/HLA, DEVS/CORBA, or parallel CD++. In this section, we show the use of a model wrapper to make two different simulation tools interact at a high level. The wrapper hides the component and provides a means of communication with components modeled in the other environment [57].

```
public class SimpleTimer : Atomic{
   public SimpleTimer() { }
   public override void init(){    hold(3.3); }    // Initialize the SPTimer
   public override void delta_int(){  hold(3.3); }  // Internal transition function.
   public override void delta_ext(double e, Bag<PortValue> x){}  // External transition
                                                                    function.
   public override void output_func(Bag<PortValue> y){
   Console.WriteLine("Alarm at " + TimeCurrent);       // Output function
   }
}
```

FIGURE 15.31 The timer model.

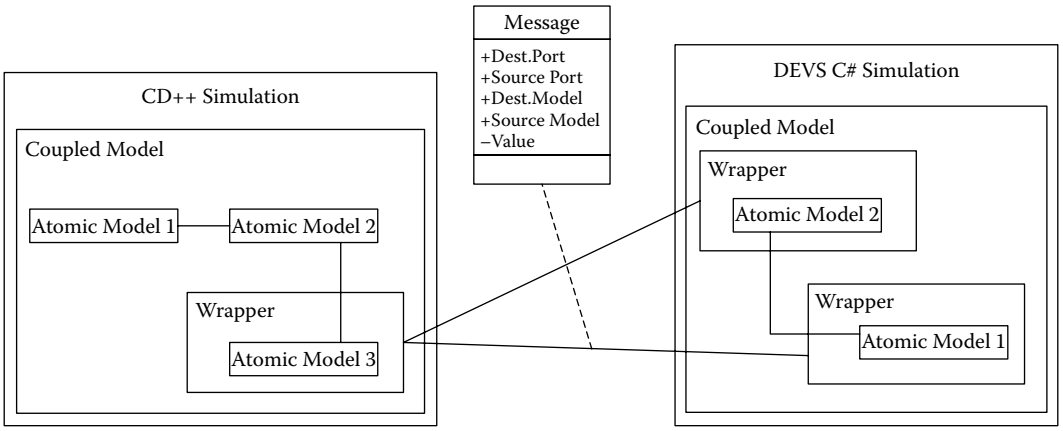


FIGURE 15.32 Connection between a CD++ simulation and a DEVS C# model using wrappers.

```
public override void output_func(Bag<PortValue> y){ // Output function
   y.Add(new PortValue(portOut, m_count.ToString()));
   m_wrapper.send(portOut.Name, "transducer", "arriv", m_count.ToString());
}
```

FIGURE 15.33 DEVS C# output function.

After receiving a message from another component's wrapper, the receiving wrapper must pass the message to its component so that the simulation can progress. CD++ wrappers and DEVS C# wrappers have different means of passing a received message to their encapsulated component (in this experiment, we used TCP/IP sockets). The diagram in Figure 15.32 represents a simple split simulation comprising two component groups. The component group on the left is modeled in CD++, and it contains three atomic models. The component *atomic model 3* is encapsulated in a CD++ wrapper. *Atomic model 3* can send messages to and receive messages from components connected to this wrapper. The component group on the right is modeled in DEVS C#, and it contains a DEVS C# coupled model. Both components in this group are encapsulated in DEVS C# wrappers in order to communicate with the CD++ group.

CD++ and DEVS C# send messages between simulations in a similar manner. In each atomic model's output function, the message is passed to the component's wrapper, which will forward it to another wrapper. This creates an explicit, loose coupling between the component groups. The code fragment from a component in DEVS C# in Figure 15.33 shows how this coupling is achieved. First, the bag of port values (here called *y*) passed by reference to the function is appended to include a message from the generator's *portOut* port containing a value (`m _ count.ToString()`). The DEVS C# simulator will use the coupling defined to deliver this message to all of its intended recipients. Next, we have a message being sent from the wrapped component to a component in another component group. This is done by invoking the wrapper's *send* function. When a component modeled in CD++ needs to send a message to another component group, a similar call is made from the component's output function to its wrapper.

Received messages are handled differently in CD++ wrappers than they are in DEVS C# wrappers. In CD++ wrappers, all messages are routed from the wrapper directly to the atomic model. The wrapper calls the atomic model's external function, passing the received message as the argument. The following is a fragment of the CD++ wrapper's *receive* function, showing how received messages are handled:

```
m_model.externalFunction(receivedMessage);
```

The variable *m_model* is the component encapsulated by the wrapper. The variable *receivedMessage* is the message received from the other wrapper.

In contrast, DEVS C# wrappers have a reference to their component's simulator. This means messages can be injected directly into the simulation by the wrapper. This results in the receipt of the message by its intended recipient models. The following is a fragment from the DEVS C# wrapper's listen function, which listens for and handles messages as they arrive:

```
PortValue pv = new PortValue(port, value);
m_wrappedSim.inject(pv);
```

The first line shows the creation of a *PortValue,* using the port and value received from the other wrapper. The second line shows the injection of the *PortValue* into the simulator, here named *m_wrappedSim.*

Prior to initiating the split simulation, each of the DEVS C# wrappers must know the IP address of the CD++ wrapper to which it will connect. Upon execution of the DEVS C# simulation, each wrapper will try to connect to the CD++ wrapper specified. When a connection can be made, two sockets are created, providing asynchronous communication between the two models (i.e., both models can be sending a message at the same time). Upon execution of a CD++ wrapper, two listening sockets are created and the wrapper waits for a connection from a DEVS C# wrapper. After a connection has been established and both sockets are ready for communication, the simulation is initiated and started. Messages are passed between the wrappers until the simulation is completed. At this time, the model where completion has been decided or detected sends out a termination message and all wrapper connections are closed.

The generator, processor, transducer (GPT) model presented in Chapter 2 was built using this method. The generator/processor component group was defined in CD++, and the transducer was created in DEVS C#. Both were interconnected using the model's wrapper, as seen in Figure 15.34. The generator creates a job (in this case, it is an integer starting at 0 and increasing by 1 with each job created). The job is sent from the generator to the processor via the CD++ coupling and to the transducer via the wrapper. Upon receiving a job, the transducer adds a timestamp, which will be used for calculation when completed job messages arrive (Figure 15.35).

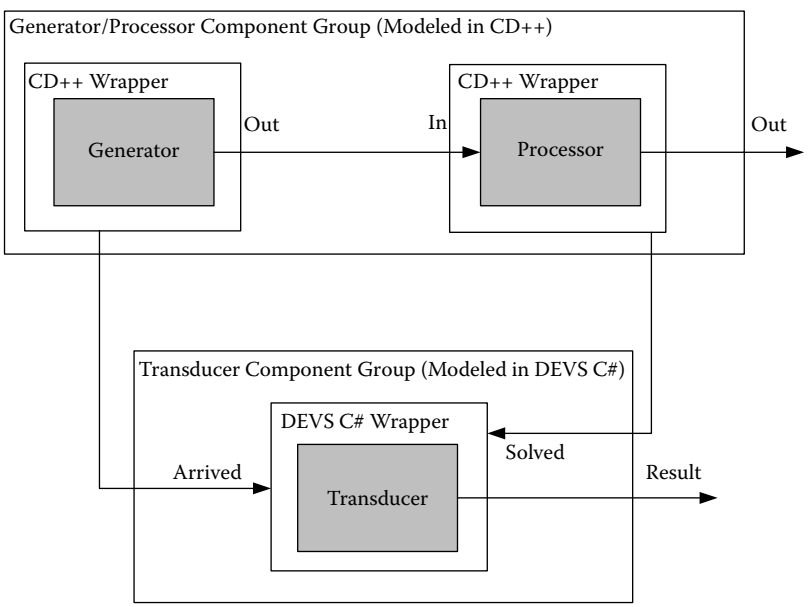


FIGURE 15.34 The split GPT model.

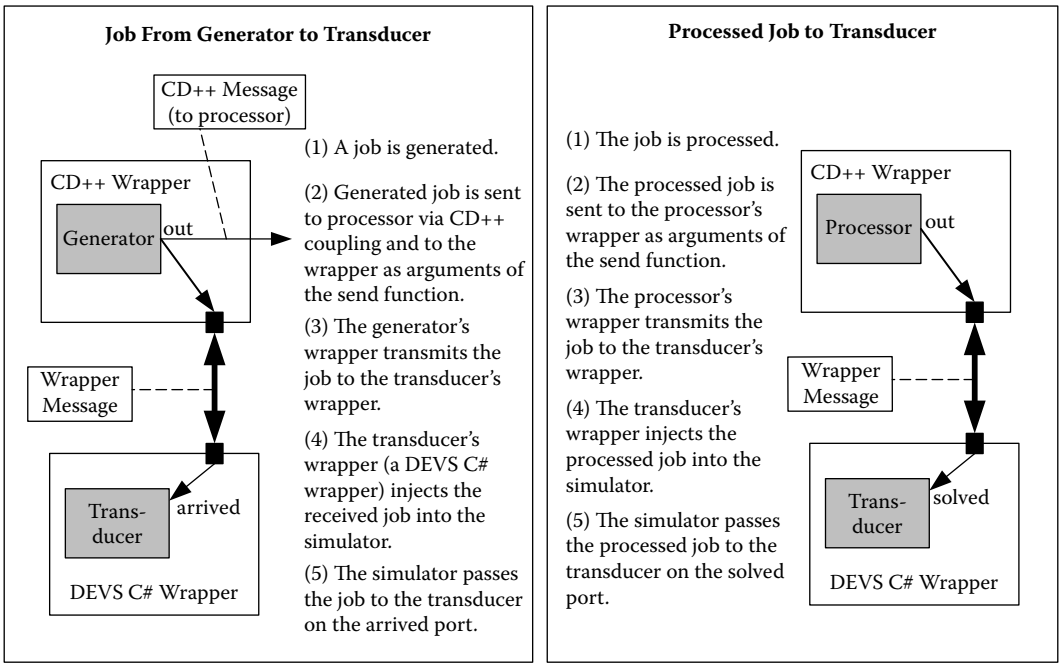


FIGURE 15.35 (a) Path that a job takes from the generator to the transducer through the wrappers; (b) path that a processed job takes from the processor to the transducer.

Upon arrival at the processor, the processor waits for the amount of processing time set during initialization. Upon completion of processing, the processor forwards the job message to the transducer via the wrapper. Upon arrival at the transducer, an elapsed time for the job is calculated. Figures 15.36 and 15.37 show output from the two environments for a short period of simulation of the split GPT model.

Figure 15.36 shows the output generated by CD++ representing the message flow of the generator/processor component group or the first 20 s of simulation. Figure 15.37 shows the DEVS C# output for the transducer component group over the same 20-s period. Information for each message

```
Message */00:00:00:000/Root(00) to top(01)
Message */00:00:00:000/top(01) to generator(02)
Message Y/00:00:00:000/generator(02) / out / 0 to top(01)
Message D/00:00:00:000/generator(02) / 00:00:10:000 to top(01)
Message X/00:00:00:000/top(01) / in / 0 to processor(03)
Message D/00:00:00:000/processor(03) / 00:00:10:000 to top(01)
Message D/00:00:00:000/top(01) / 00:00:10:000 to Root(00)
Message */00:00:10:000/Root(00) to top(01)
Message */00:00:10:000/top(01) to generator(02)
Message Y/00:00:10:000/generator(02) / out / 1 to top(01)
Message D/00:00:10:000/generator(02) / 00:00:10:000 to top(01)
Message X/00:00:10:000/top(01) / in / 1 to processor(03)
Message D/00:00:10:000/processor(03) / 00:00:10:000 to top(01)
Message D/00:00:10:000/top(01) / 00:00:10:000 to Root(00)
Message */00:00:20:000/Root(00) to top(01)
Message */00:00:20:000/top(01) to generator(02)
Message Y/00:00:20:000/generator(02) / out / 2 to top(01)
Message D/00:00:20:000/generator(02) / 00:00:10:000 to top(01)
Message X/00:00:20:000/top(01) / in / 2 to processor(03)
Message D/00:00:20:000/processor(03) / 00:00:10:000 to top(01)
Message D/00:00:20:000/top(01)/00:00:10:000 to Root(00)
```

FIGURE 15.36 CD++ results of a short split simulation.

```
0 transducer's ext:         -- {portAriv:0} -->
10 transducer's ext:        -- {portSolv:0} -->
10 transducer's ext:        -- {portAriv:1} -->
20 transducer's ext:        -- {portSolv:1} -->
20 transducer's ext:        -- {portAriv:2} -->

End Time     : 20
Jobs Arrived : 3
Jobs Solved  : 2
Total TA     : 20
Average TA   : 10
Throughput   : 0.1
```

FIGURE 15.37 DEVS C# results for the split simulation.

is formatted to take two lines. The first line shows the time of the event, the name of the model, and the function triggered (internal, external, or confluent). The second line shows the previous state, the port name and value on the port and the new state. For this example, the states are blank because the transducer has only one non-passive state and it is unnamed. Following termination, the transducer displays its results, showing the end time of the simulation, the number of arrived and solved jobs, the total and average time advance, and the processor's throughput.

15.12 SUMMARY

This chapter has introduced the different existing simulators available for CD++. We first introduced the basic simulation algorithms for single-processor DEVS models. We explained in detail how these simulation algorithms are implemented in the stand-alone version of CD++ (which can be downloaded and modified by users interested in changing or adapting the simulation engine), including the definition of DEVS and Cell-DEVS models.

This independence between the simulation and modeling software components allowed us to create varied simulation engines. We briefly presented the main ideas on the implementation of

- a CD++ parallel simulator, which executes DEVS models in parallel in distributed memory architectures;
- flat Coordinator algorithms (to accelerate simulation time);
- a real-time simulator (to provide interaction with external devices in real time);
- a dynamic structure DEVS simulator (to allow dynamic reconfiguration of the models);
- a distributed simulation engine built on Web Services technology; and
- a mechanism for interfacing two different DEVS simulators (C++ and DEVS/C#).

Having separate entities for the models and their simulators offers the advantage of isolating the simulator architecture from the model structure so that changing the simulator internals does not affect the model's definition. In addition, it facilitates the use because the modeler needs only to define the model without any knowledge of the simulator (as we demonstrated in previous chapters).

REFERENCES

1. Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation,* 2nd. ed. New York: Academic Press.
2. Rodriguez, D., and G. Wainer. 1999. New extensions to the CD++ tool. *Proceedings of 31st SCS Summer Computer Simulation Conference,* Chicago, IL.
3. Barylko, A., J. Beyoglonián, and G. Wainer. 1998. GAD: A general application DEVS environment. *Proceedings of Applied Modeling and Simulation,* Honolulu, HI.
4. Fujimoto, R. M. 1999. *Parallel and distribution simulation systems.* New York: John Wiley & Sons.
5. Fujimoto, R. 1990. Parallel simulation of discrete events. *Communications of the ACM* 33 (10): 30–53.

6. Nicol, D., and R. Fujimoto. 1994. Parallel simulation today. *Annals of Operations Research* 53:249–285.
7. Banks, J., J. S. Carson, B. L. Nelson, and D. Nicol. 2005. *Discrete-event system simulation,* 4th ed. Upper Saddle River, NJ: Prentice Hall.
8. Chandy, K., and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM* 24:198–206.
9. Chandy, K., and J. Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* SE-5:440–452.
10. Bryant, R. E. 1977. Simulation of packet communication architecture computer systems. Technical report, UMI order number: TR-188. Massachusetts Institute of Technology.
11. Jefferson, D. 1987. Distributed simulation and the time warp operating system. *Proceedings of 11th Symposium on Operating Systems Principles*, Austin, TX.
12. Jefferson, D. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7:404–425.
13. Henning, M., and S. Vinoski. 1999. *Advanced CORBA programming with C++*. Reading, MA: Addison–Wesley.
14. Dongarra, J. J. 1995. *High-performance computing: Technology, methods and applications.* Amsterdam: Elsevier.
15. Sunderam, V., A. Geist, J. Dongarra, and R. Manchek. 1994. The PVM concurrent computing system: Evolution, experience and trends. *Parallel Computing* 20:531–545.
16. Alonso, G. 2003. *Web Services: Concepts, architectures and applications.* New York: Springer–Verlag.
17. IEEE Std 1278. 1995. IEEE standard for modeling and simulation. Distributed interactive simulation (DIS).
18. IEEE Std 1516.1-2000. 2001. IEEE standard for modeling and simulation. High-level architecture (HLA)—Federate interface specification, i–467.
19. Cho, Y. W., X. Hu, and B. Zeigler. 2003. The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems. *Simulation* 79 (4): 197–210.
20. Sarjoughian, H. S., and B. P. Zeigler. 2000. DEVS and HLA: Complementary paradigms for M&S? *Transactions of the SCS* 17:187–197.
21. Kim, K., and W. Kang. 2004. CORBA-based, multithreaded distributed simulation of hierarchical DEVS models: Transforming model structure into a nonhierarchical one. *Proceedings of ICCSA 2004, International Conference,* Assisi, Italy, 167–176.
22. Seo, C., S. Park, B. Kim, S. Cheon, and B. P. Zeigler. 2004. Implementation of distributed high-performance DEVS simulation framework in the grid computing environment. *Proceedings of High Performance Computing Symposium, Advanced Simulation Technology Conference,* Arlington, VA.
23. Cheon, S., C. Seo, S. Park, and B. P. Zeigler. 2004. Design and implementation of distributed DEVS simulation in a peer-to-peer network system. *Proceedings of DASD, Advanced Simulation Technology Conference,* Arlington, VA.
24. Chow, A. C., and B. Zeigler. 1994. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. *Proceedings of Winter Simulation Conference,* Orlando, FL.
25. Chow, A. C., D. H. Kim, and B. P. Zeigler. 1994. Abstract simulator for the P-DEVS formalism. *Proceedings of AI, Simulation, and Planning in High Autonomy Systems,* Gainesville, FL.
26. Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1996. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS* 13 (3): 135–154.
27. Glinsky, E., and G. Wainer. 2006. New parallel simulation techniques of DEVS and cell-DEVS in CD++. *Proceedings of Annual Simulation Symposium,* Huntsville, AL, 244–251.
28. Glinsky, E., and G. Wainer. 2002. Performance analysis of real-time DEVS models. *Proceedings of Winter Simulation Conference,* San Diego, CA.
29. Glinsky, E., and G. Wainer. 2002. Performance analysis of DEVS environments. *Proceedings of Artificial Intelligence, Simulation and Planning.* Lisbon, Portugal.
30. Kim, K., W. Kang, B. Sagong, and H. Seo. 2000. Efficient distributed simulation of hierarchical DEVS models: Transforming model structure into a nonhierarchical one. *Proceedings of 33rd Annual Simulation Symposium,* Washington, D.C.
31. Cho, S., and T. G. Kim. 2001. Real-time simulation framework for RT-DEVS models. *Transactions of the Society for Computer Simulation International* 18:203–215.
32. Wainer, G., and N. Giambiasi. 2001. Application of the cell-DEVS paradigm for cell spaces modeling and simulation. *Simulation* 76 (1): 22–39.
33. Wainer, G. 1998. Discrete-event cellular models with explicit delays. PhD thesis, Université d'Aix-Marseille III, France.

34. Troccoli, A., and G. Wainer. 2003. Implementing parallel cell-DEVS. *Proceedings of 36th IEEE/SCS Annual Simulation Symposium,* Orlando, FL.
35. Madhoun, R., B. Feng, and G. Wainer. 2007. Web-service-based distributed CD++. *Proceedings of AIS 2007, Artificial Intelligence, Simulation and Planning,* Buenos Aires, Argentina.
36. Schulz, S., J. W. Rozenblit, M. Mrva, and K. Buchenriede. 1998. Model-based codesign. *Computer* 31:60–67.
37. Hong, J., H. Song, T. G. Kim, and K. H. Park. 1997. A real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems: Theory and Applications* 7:355–375.
38. Kim, T. G., S. M. Cho, and W. B. Lee. 2000. DEVS framework for systems development: Unified specification for logical analysis, performance evaluation and implementation. In *Discrete event modeling & simulation: Enabling future technologies,* ed. H. S. Sarjoughian and F. Cellier. New York: Springer–Verlag.
39. Glinsky, E., and G. Wainer. 2002. Definition of RT simulation in the CD++ toolkit. *Proceedings of Summer Computer Simulation Conference,* San Diego, CA.
40. Li, L., T. Pearce, and G. Wainer. 2002. An experience in hardware–software codesign using the DEVS formalism. *Proceedings of EuroSim Industrial Simulation Symposium 2003,* Valencia, Spain.
41. Glinsky, E., and G. Wainer. 2004. Modeling and simulation of systems with hardware-in-the-loop. *Proceedings of Winter Simulation Conference,* Washington, D.C.
42. Jacques, C., and G. Wainer. 2002. Using the CD++ DEVS toolkit to develop Petri nets. *Proceedings of Summer Computer Simulation Conference,* San Diego, CA.
43. Glinsky E., and G. Wainer. 2004. Model-based development of embedded systems with RT-CD++. *Proceedings of the WIP Session, IEEE Real-Time and Embedded Technology and Applications Symposium,* Toronto, ON, Canada.
44. Wainer, G., E. Glinsky, and P. MacSween, 2005. Model-driven architecture of real-time systems. In *Model-driven software development—Research and practice in software engineering,* vol. II, ed. S. Beydeda and V. Gruhn. New York: Springer–Verlag.
45. Yu, H., and G. Wainer. 2007. E-CD++: An environment for developing embedded DEVS applications. Carleton University, Dept. of Systems and Computer Engineering.
46. Barros, F. J. 1995. Dynamic structure discrete event system specifications: A new formalism for dynamic structure modeling and simulation. *Proceedings of Winter Simulation Conference,* Arlington, VA, 781–785.
47. Barros, F. 1998. Abstract simulators for the DSDE formalism. *Proceedings of Winter Simulation Conference,* Washington, D.C., 407–412.
48. Uhrmacher, A. M. 2001. Dynamic structure in modeling and simulation: A reflective approach. *ACM Transactions on Modeling and Computer Simulation* 11:206–232.
49. Uhrmacher, A. M., and J. Himmeelspach. 2004. Processing dynamic PDEVS models. *Proceedings of 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04),* Volenlam, the Netherlands.
50. Shang, H., and G. Wainer. 2006. A simulation algorithm for dynamic structure DEVS modeling. *Proceedings of Winter Simulation Conference,* Monterey, CA.
51. Shang, H., and G. Wainer. 2008. Dynamic structure DEVS: Improving the real-time embedded systems simulation and design. *Proceedings of 40th IEEE/SCS Annual Simulation Symposium,* Ottawa, Canada.
52. Kgwadi, M., H. Shang, and G. Wainer. 2008. Definition of dynamic DEVS models—Dynamic structure CD++. *Proceedings of Poster Papers Workshop, SpringSim 2008,* Ottawa, Canada.
53. Christensen, E., F. Curbera, G. Meredith, and S. Weerawarana. 2006. *Web Service description language (WSDL) 1.1.* URL: http://www.w3.org/TR/wsdl
54. Gudgin, M., M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. 2006. *SOAP version 1.2 part 1: Messaging framework.* URL: http://www.w3.org/TR/soap12-part1/
55. Madhoun, R., and G. Wainer. 2007. Performance analysis of Web-based CD++. Presented at DEVS Symposium 2007, Norfolk, VA,
56. Wainer, G., Q. Liu, J. Landais, L. Quinet, and M. K. Traoré. 2008. Performance analysis of Web-based distributed simulation in DCD++: A case study across the Atlantic Ocean. Presented at SCS High Performance Computing and Simulation Symposium (HPCS 2008), Ottawa, Canada.
57. Lombardi, S., G. Wainer, and B. P. Zeigler. 2006. Interoperation of DEVS models in DEVS/C# and CD++. *Proceedings of SISO Fall Interoperability Workshop,* Huntsville, AL.

16 Mechanisms for Three-Dimensional Visualization

16.1 INTRODUCTION

Current simulation practices rely on close cooperation between application domain specialists (sometimes with limited experience in software development) and software specialists (with limited expertise in the domain of application). This cross-domain communication often leads to a fair amount of difficulties in model specification, validation, and verification. Recent development of advanced computer graphics has provided a unique opportunity to address these problems, enabling the adoption of modeling and simulation (M&S) technology among many organizations. The availability of adequate visualization mechanisms (generic and flexible to show different results in an intuitive and user-friendly manner) can ease the analysis of complex system behavior while enabling advanced training facilities, including simulations with hardware-in-the-loop and live-constructive-virtual simulation for defense training [1].

This chapter introduces different visualization mechanisms and tools that have been integrated into the CD++ environment in an attempt to help with these goals. Although CD++Modeler provides a basic Graphical User Interface (GUI) that supports two-dimensional visualization, in order to be able to show complex behavior intuitively, we developed mechanisms to interface CD++ with a variety of three-dimensional visualization and rendering tools. This includes VRML (Virtual Reality Modeling Language) [2], Autodesk Maya [3], OpenGL (Open Graphics Library) [4], and Blender [5]. Many of these tools are open source, and they provide a platform for investigating three-dimensional visualization and its relation with discrete-event simulation environments. The result is an integrated simulation and visualization environment that is able to meet the diverse needs of different users, to ease the validation and verification of continuously evolving models.

16.2 THREE-DIMENSIONAL ANIMATION USING CD++/VRML

VRML is a Web-based graphics language for describing interactive three-dimensional objects and virtual worlds created using a scene-graph structure [2]. It defines a universal interchange file format, and it allows users to interact with a scene through viewpoints, movement, and rotations. Although VRML has now been superseded by extensible 3D (X3D)—an open ISO standard for real-time three-dimensional computer graphics [6]—the technique still enjoys widespread use in education and research communities. The Java programming language has been integrated with VRML to enhance the animation and interactivity of three-dimensional visual models and scenes, resulting in the Java/VRML connection in the VRML specification 2.0. The external authoring interface (EAI) provides a Java Application Program Interface (API) that enables a Java applet to update and control the contents of a three-dimensional VRML scene [7].

A VRML *scene graph* consists of an ordered collection of nodes hierarchically grouped that represent objects and their properties. The scene graph also generates events in response to environmental changes and user interaction and participates in event routing mechanisms through which the effect changes are propagated to other nodes.

The *node* is the basic VRML structure, which represents a visual object with certain properties (shape, color, light, position and orientation, viewpoints, subscenes, and sensors for user inputs). A VRML *scene* consists of a group of nodes of different types organized hierarchically [7]. *Grouping nodes* combine several other nodes into a common entity. Each grouping node defines a coordinate space for its children and provides events (or methods) to add or remove nodes from it. A *transform node* is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors. It is used to move, rotate, and scale a visual entity. An *inline node* is another type of grouping node that reads the data of its children from a given location specified by a URL. Different *sensor nodes* (e.g., `CylinderSensor`, `PlaneSensor`, `ProximitySensor`, `TimeSensor`) are used to generate events in response to user actions or expiration of a timer. A *shape node* represents a visual object using two fields: an *appearance node* that specifies the material and texture attributes of the object and a *geometry node* that is rendered using the specified appearance characteristics. Each shape node can be manipulated by modifying its parameters (radius of a sphere, height and width of a box, etc.).

Wainer and Chen [8] introduced a visualization facility (referred to as CD++/VRML), which enables visualization of Cell-DEVS models in a VRML virtual world. Users can navigate and interact with the nodes in the scene using an external Java Applet. The tool can be found at http://www.sce.carleton.ca/faculty/wainer/vrmlGUI/index.html. The tool's source code is also available for use and modification.

16.2.1 Integrating CD++ and VRML for Interactive Three-Dimensional Visualization

CD++/VRML is based on predefined VRML constructions. The visualization starts with an empty VRML root file embedded in an HTML Web page, representing an empty scene to which the simulation data can be added as child nodes. These nodes can then be manipulated by an external Java applet according to the simulated model behavior recorded in a CD++ log file. The applet then acts as the interface between CD++ and the VRML virtual world, providing a set of functions for the user to control the scene. Major functionalities of the applet are as follows [8]:

- adding or removing a node from the VRML scene;
- modifying the shape and color of individual nodes and controlling their visibility;
- defining the coloring scheme for different value ranges of the nodes;
- changing the scale of the nodes as well as the intervals between them;
- navigating in the VRML scene;
- editing the scene and individual nodes;
- loading and rendering a simulation result stream; and
- using a set of viewpoints that permit users to switch to different viewing areas in the scene.

As illustrated in Figure 16.1, these functionalities are arranged into several categories, each of which is implemented in a separate panel. The link between the CD++ environment and a VRML scene is implemented in the class `ReadDrwFile`, which takes care of retrieving simulation data from the log input stream. The data retrieval is activated by the `NavigatePanel` whenever the user moves to the next result, which can be the data with the next (forward) or previous (backward) simulated virtual time in the result streams or the data with a user-specified virtual time. When the applet starts, the `InfoPanel` is presented, allowing the user to choose the result stream to be visualized and to define a coloring scheme for the nodes.

`NavigatePanel` (the main class of the toolkit) keeps track of the data currently displayed in the scene, a history of recently navigated nodes, and the name of every node displayed. This information changes whenever the VRML scene is updated during the navigation. For Cell-DEVS models, `NavigatePanel` initiates the scene as an array of nodes corresponding to the size of the cell space. Users can modify the attributes of individual nodes and examine data from different

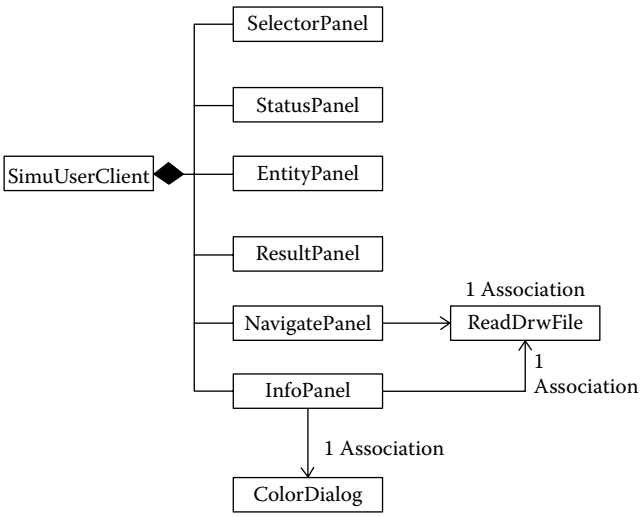


FIGURE 16.1 Applet panels for visualizing simulation data in a VRML scene. (From Wainer, G., and W. Chen. 2003. *Simulation* 79:626–647.)

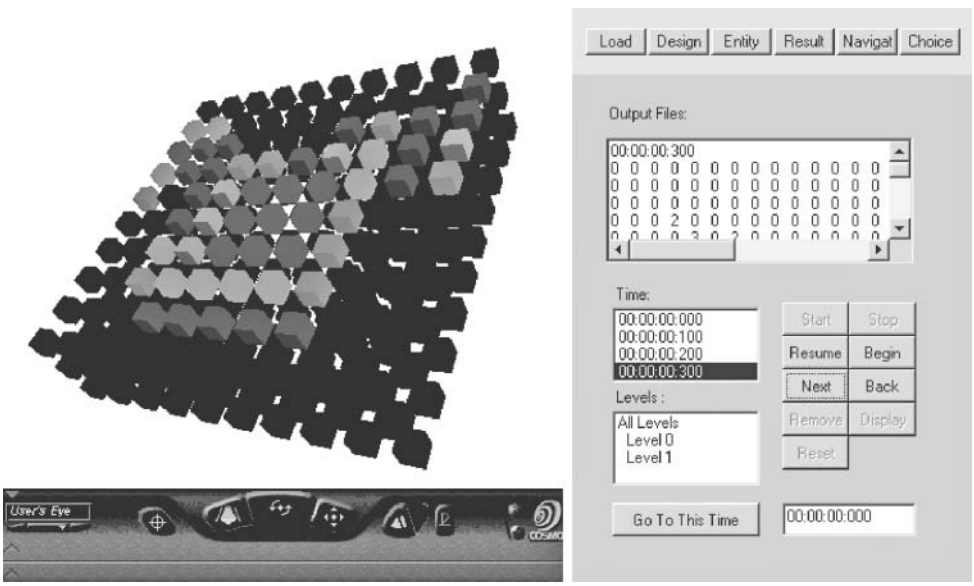


FIGURE 16.2 Graphical user interface of the *NavigatePanel*.

viewpoints. Users can also specify a node as the current node and add/remove layers in the scene. Figure 16.2 shows the GUI of the `NavigatePanel`.

By using the methods defined in the `EntityPanel`, users can edit the attributes (e.g., shape, color, and size) of currently selected nodes. Nodes can also be added or removed from the scene. `ResultPanel` provides different methods for controlling the navigation. When these methods are invoked, the corresponding functions in the `NavigatePanel` are activated to change the scene accordingly. The main methods include:

- `start`, to initiate the execution;
- `resume`, to continue execution if it has been stopped;
- `next/back`, to step forward/backward in the animation sequence;
- `stop`, to halt the animation at a given stop time;
- `display`, for continuous animation until the end of the result stream; and
- `goto`, to allow the user to jump to any given simulated time.

Figures 16.3 and 16.4 demonstrate some of the major capabilities of CD++/VRML using the three-dimensional heat diffusion model that was introduced in Chapter 5. As mentioned earlier, users can choose to represent visual objects using different geometries such as boxes, spheres, cones, or cylinders, as we can see in Figure 16.3.

A user can edit the attributes of individual nodes in order to highlight a particular node or group of nodes of interest for further examination. For complex models, users can also delete nodes from the scene to focus on the remaining data. Figure 16.4(a) shows the effect of editing a single node in the scene, and Figure 16.4(b) illustrates the removal of a layer of cells from the scene. Finally, users can create multiple instances of the VRML scene derived from the same set of simulation data so that different viewing areas can be examined using different graphical metaphors at the same time.

Exercise 16.1

Using the VRML GUI interface, visualize the simulation results of the following examples: (1) three-dimensional heat, (2) a bouncing ball, (3) fire, and (4) a maze.

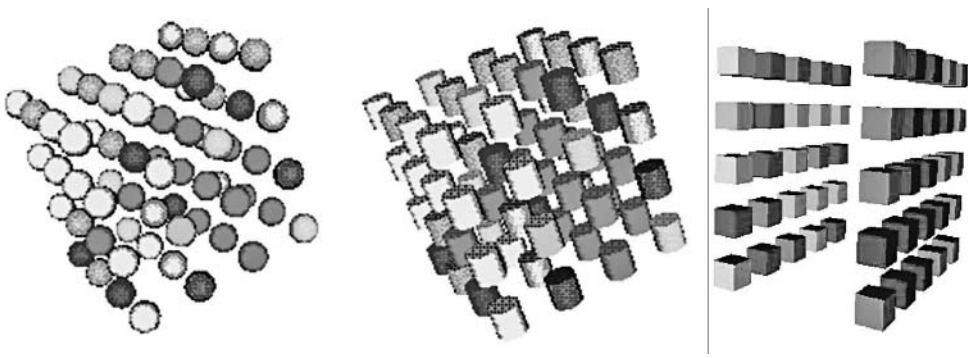
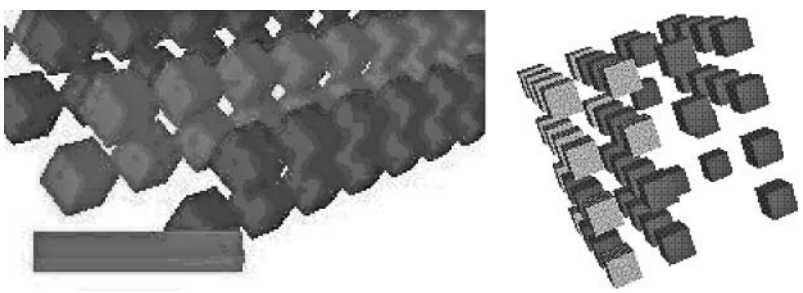


FIGURE 16.3 Three-dimensional cell space visualization using different geometries and coloring schemes.



(a) (b)

FIGURE 16.4 Node editing and scaling in the scene.

Exercise 16.2

For each of the models in Exercise 16.1, create different palettes for color visualization. Navigate the model, changing the geometries used and the size of the elements in the simulation.

16.2.2 Graphical Modeling and Visualization of Urban Traffic with MAPS

In Chapter 14, we introduced the ATLAS language for modeling traffic. A simulation project in traffic using ATLAS is carried out as a sequence of the procedures shown in Figure 16.5.

Initially, a graphical model should be defined with the MAPS tool [9], which allows users to draw city sections directly through a GUI and automatically translates the imagery into valid ATLAS models (text based). These models are, in turn, compiled into Cell-DEVS models for simulation in CD++ using the traffic simulation compiler (TSC). This graphical environment eliminates the need for learning the ATLAS language and reduces the model development time. The simulation results of CD++ are then presented with a customized three-dimensional visualization facility that presents the traffic network and moving vehicles in a user-friendly manner [9].

The MAPS subsystem was built on top of JHotDraw [10], an open source, two-dimensional Java GUI framework for developing structured drawing editors. The parser is responsible for translating the graphical version of a city section into a valid ATLAS specification. It first removes and stores the crossings of the traffic network and the city-level decorations (e.g., railways) and then loops through the roads to identify the intersections. New intersections are automatically created as needed and the roads are cut into appropriate segments, each having its parameters set based on the road configuration (e.g., whether parking is allowed or not, the position at which the road segment crosses a railway line). The process continues until all the roads and lanes in the imagery are parsed.

The VRML-based output module included in MAPS is used to reconstruct the specified city section in VRML virtual worlds and to animate traffic flows in realistic three-dimensional graphics according to the simulation results. The output module, found at http://www.sce.carleton.ca/faculty/wainer/vrmlGUI/index.html, uses the generated ATLAS file to reconstruct a static scene of the city section in a VRML virtual world. It then determines the location and direction of each vehicle at a particular point in time, based on a CD++ log file. A three-dimensional car shape is displayed on the screen at the appropriate cell of a road segment for the time duration, as indicated in the log file. When that time expires, the car is moved to the next cell per the simulation results. In this way, the animation shows a microscopic view of the traffic flow on the road network. Users can navigate through the virtual city section using the VRML navigation panel that we have introduced in the previous section.

The three-dimensional visualization environment translates ATLAS constructions and simulated vehicles into VRML objects or nodes. Different shapes are used to represent the vehicles and various constructions such as the cells of a road segment, crossings, and traffic signs. Figure 16.6 depicts some of the VRML objects used to represent these entities. Note that a one-to-one correspondence is established among the Cell-DEVS model components, the ATLAS constructions, and the VRML visual objects.

In order to visualize the ATLAS file properly, the output module needs to calculate the length and rotation angle for each road segment and then place it at the appropriate position in the VRML

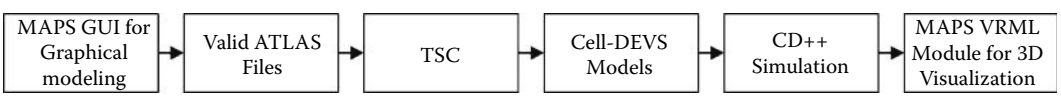


FIGURE 16.5 Procedure and techniques for urban traffic modeling and simulation.

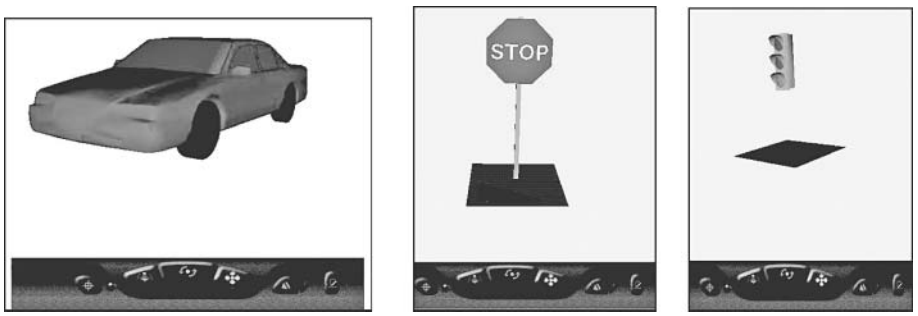


FIGURE 16.6 VRML visual objects for vehicles and crossings. (From Wainer, G. 2007. *Software Practice and Experience* 37:1377–1404.)

scene. If a segment is parallel to a coordinate axis, the angle does not need to be calculated, and the length is computed as

$$length = \sqrt{(P_{1x} - P_{2x})^2 + (P_{1y} - P_{2y})^2} \tag{16.1}$$

where (P_{1x},P_{1y}) and (P_{2x},P_{2y}) are the two end points of the segment.

On the other hand, if a segment does not run parallel to a coordinate axis, then the corresponding VRML object must be rotated by an angle that is obtained as follows:

$$angle = \tan^{-1}\left(\frac{P_{2y} - P_{1y}}{P_{2x} - P_{1x}}\right) \tag{16.2}$$

Once the length and angle are determined, the output module maps the segment to a visual object that is stretched from the given start point and scaled and rotated according to the calculated values.

Exercise 16.3

Compute the length of segment `A = (0,0), (10,10), 1, straight, go, 40, 300, parkNone`.

Exercise 16.4

Compute the length of every segment in the example introduced in Figure 14.34, Chapter 14.

The output module parses the log file generated by the CD++ simulator to animate the traffic flows in a VRML virtual world. The entering of a car in a particular cell is detected by an output message (Y) with a value of 1 from the *out* port of the cell; the departure of a car from a cell is indicated by a Y message with value 0. In the former case, the output module creates a VRML car shape and rotates it by the same amount as the cell to which the car belongs. In the latter case, the car shape is removed from the current cell and the output module looks ahead in the log file to find an entering event scheduled at the destination cell. The car shape will be put at the new location after the delay time given in the Y message, as seen in Figure 14.39 in Chapter 14.

16.3 ADVANCED TECHNIQUES FOR VISUALIZATION OF DEVS AND CELL-DEVS MODELS IN CD++

Although CD++/VRML provides a three-dimensional visual M&S environment, VRML is a standard that is no longer widely supported. We have developed mechanisms to integrate the CD++

environment with a variety of both commercial and open source visualization and rendering techniques, including Autodesk Maya, OpenGL, and Blender. In this section, we will elaborate on these advanced techniques and demonstrate their capabilities.

16.3.1 CD++/Maya—High-Performance Three-Dimensional Visualization Engine for CD++

Autodesk Maya [3] is one of the leading commercial software packages for three-dimensional modeling, animation, and visual effects. The Maya software interface is fully customizable and allows users to extend their functionality from within Maya by providing access to the Maya embedded language (MEL). With MEL, users can tailor the GUI to fulfill their specific needs and to develop in-house tools. We have used MEL to create CD++/Maya, a three-dimensional visualization engine [12] that enables interoperability between DEVS-based M&S tools and advanced generic visualization environments. Users can create static scenes (providing the necessary background for three-dimensional animation of the simulation results) and use DEVS and Cell-DEVS to generate a log stream to record the events during a simulation. The CD++/Maya engine then loads and initializes the predefined scene file based on the model definition and overlays three-dimensional animation on top of the static scene according to the log data. Figure 16.7 shows the major modules defined in the visualization engine [12].

The `logFileAnimation` module serves as an interface between CD++ and Maya. Two modes are available for analyzing the simulation data: *direct analysis* and *animation*. The former allows advanced users to take a close look at the content of the log file for debugging and model verification purposes, and the latter presents an intuitive three-dimensional animation for the selected model. The `readFile` module displays the content of log files in a script editor window, as shown in Figure 16.8. The `animator` module reads the log file, creates Maya objects based on the log data, and runs the animation in three dimensions. The `translateTime` module is in charge of matching the animation time line with the simulated virtual time as presented in the log file. Finally, the `maFileReader` module is used internally by the visualization engine to retrieve the initial cell values of Cell-DEVS models.

We will now show the visualization of different models introduced in previous chapters in order to exemplify the features of the tool. All the examples in this section have been included for visualization at http://www.sce.carleton.ca/faculty/wainer/vrmlGUI/maya.html. For instance, Figure 16.9 shows a snapshot of the animation maze-solving algorithm introduced in Chapter 5. As we can see, the advanced three-dimensional rendering and visual effects open up new possibilities for increasing the representational validity of the model's behavior.

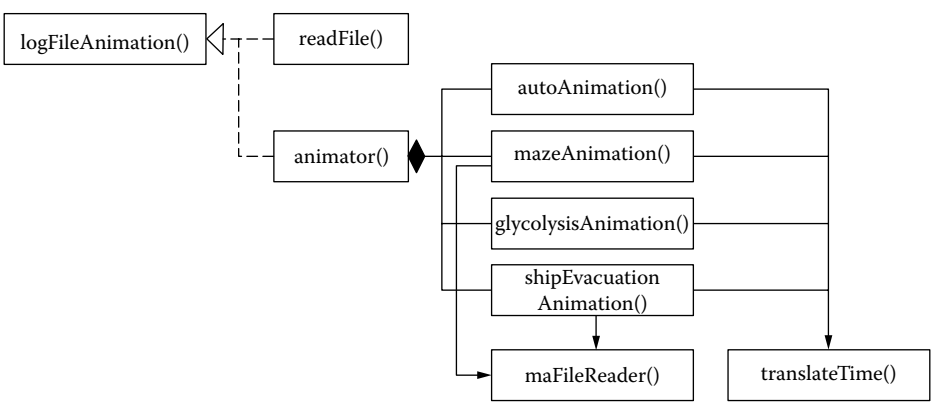


FIGURE 16.7 Major modules defined in the CD++/Maya visualization engine [12].

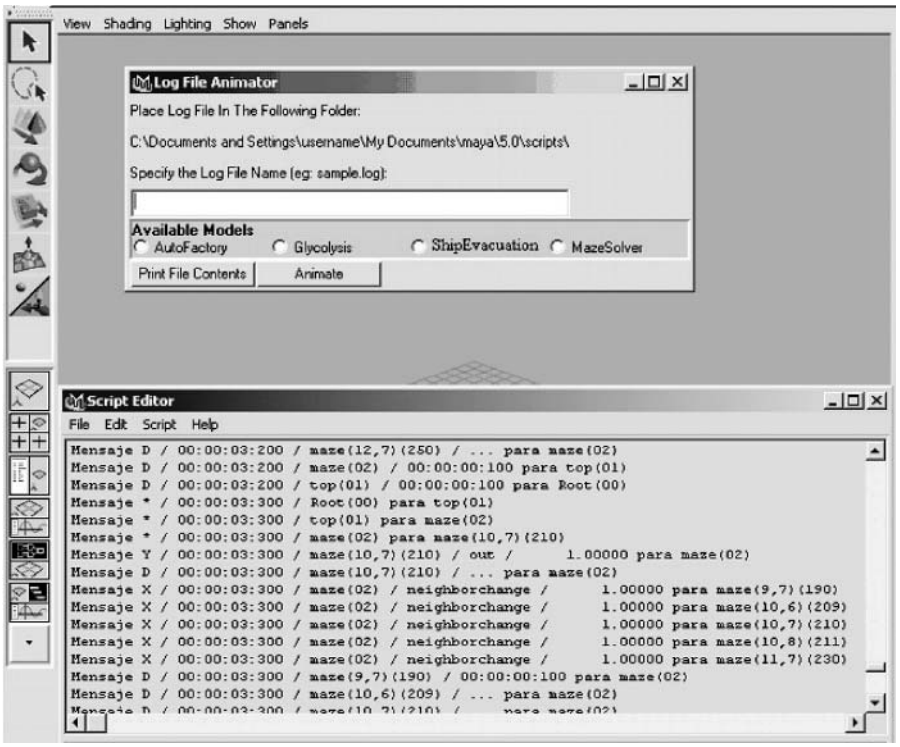


FIGURE 16.8 The script editor window displays the contents of the log file [12].

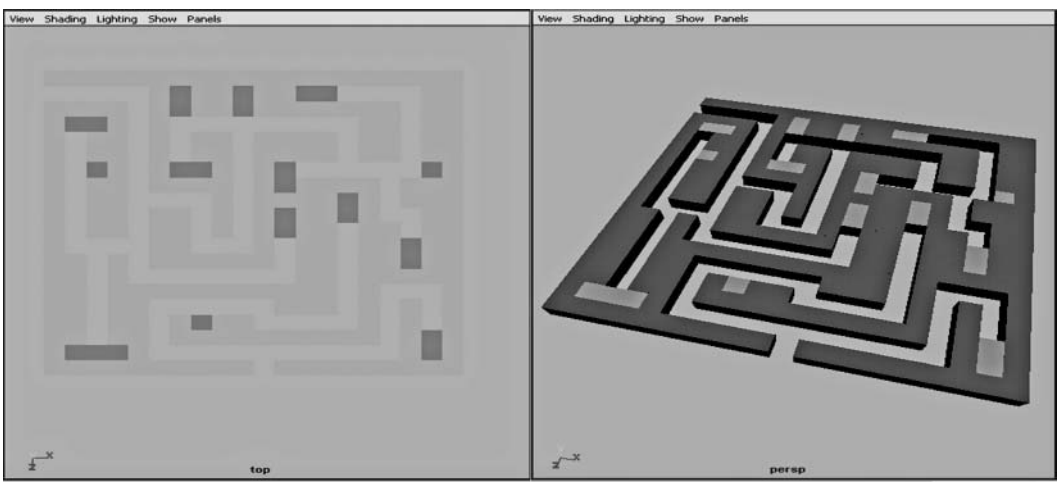


FIGURE 16.9 CD++/Maya animation from different perspectives [12].

We introduced a model representing a manufacturing plant for vehicles that is trying to maintain a suitable production level by coordinating the operation of its various assembly lines (the model can be found in *./auto.zip*). The structure of this model is depicted in Figure 16.10. The automobile parts are produced by four assembly lines (i.e., chassis, body, transmission case, and engine, respectively).

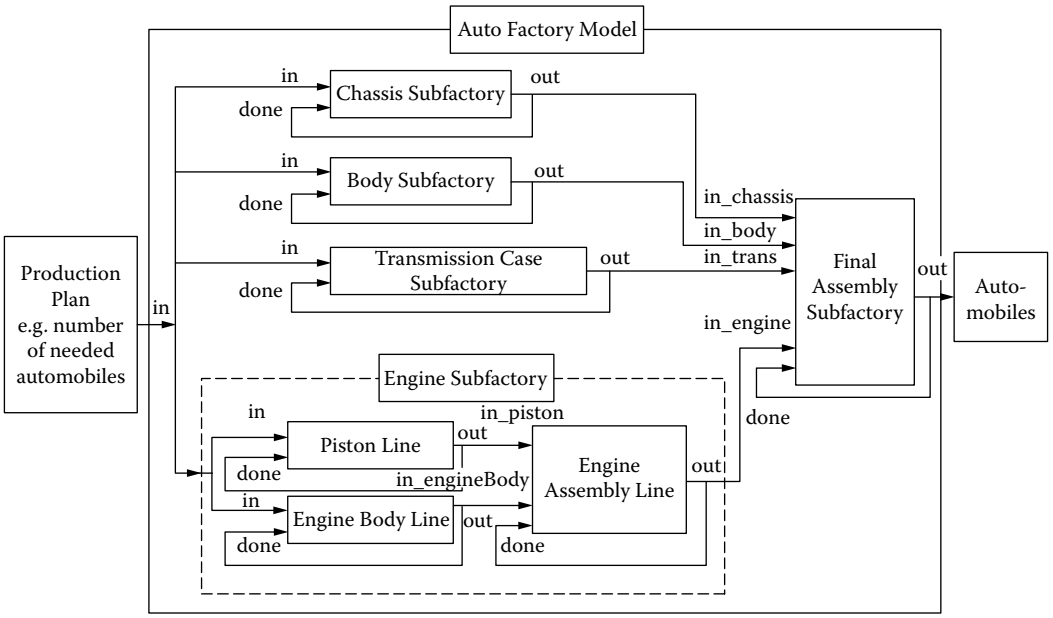


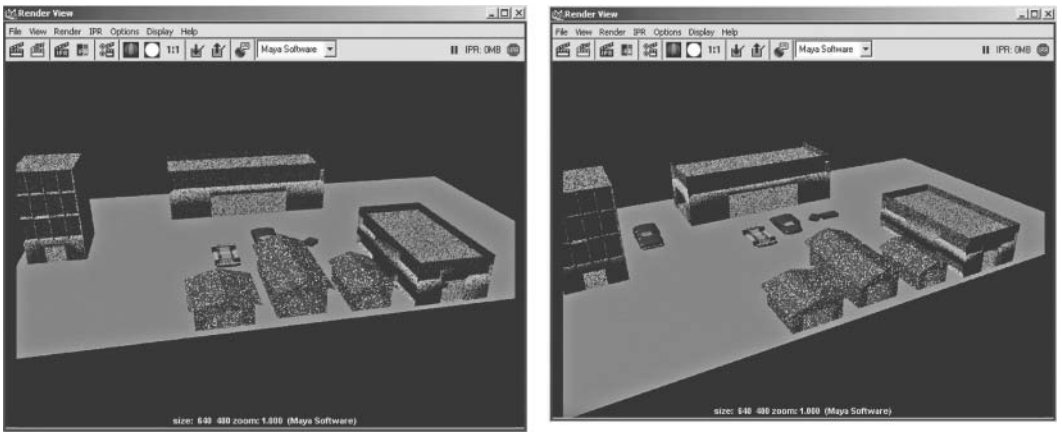
FIGURE 16.10 A car manufacturing coupled model [12].

```
X/00:000/top/in/2 to chassis
X/00:000/top/in/2 to body
X/00:000/top/in/2 to trans
X/00:000/top/in/2 to enginesubfact
D/00:000/chassis/02:000 to top
D/00:000/body/02:000 to top
D/00:000/trans/02:000 to top
X/00:000/enginesubfact/ in/2 to piston
X/00:000/enginesubfact/ in/2 to enginebody
...
Y/02:000/chassis/out/1 to top
D/02:000/chassis/... to top
X/02:000/top/done/1 to chassis
X/02:000/top/in_chassis/1 to finalassem ...
*/02:000/top to enginesubfact
*/02:000/enginesubfact to enginebody
Y/02:000/enginebody/out/1 to enginesubfact
D/02:000/enginebody/... to enginesubfact
X/02:000/enginesubfact/done/1 to enginebody
X/02:000/in_enginebody/1 to engineassem
D/02:000/enginebody/02:000 to enginesubfact
D/02:000/engineassem/02:000 to enginesubfact
```

FIGURE 16.11 Car manufacturing model-generated log file [12].

A car is then assembled using one component from each of these assembly lines. Further, an engine comprises four pistons and one engine body.

Figure 16.11 is generated when the model is executed by the CD++ simulator. The input and output trajectories of the DEVS model can be animated in two dimensions using CD++Modeler



(a) (b)

FIGURE 16.12 Animation of the car manufacturing model in CD++/Maya [12].

as we have discussed in Section 3.4 in Chapter 3. Although this allows a detailed analysis of the simulation data, it is still rather abstract and elusive for general users to have an intuitive and global comprehension of the simulated phenomenon.

Figure 16.12 illustrates two snapshots of the animation of the car manufacturing model in CD++/Maya at different virtual times, found at http://www.sce.carleton.ca/faculty/wainer/vrmlGUI/maya.html. In the log file, the availability of a part is represented as a Y message sent from the corresponding assembly line to the final assembling warehouse. Such an activity is shown in the animation as a three-dimensional icon that stands for the specific automobile part moving between the entities in the virtual world. For example, the animation in Figure 16.12(a) shows that three auto parts are made available at virtual time 02:000, and Figure 16.12(b) shows a finished car moving out of the assembling warehouse at virtual time 04:000.

The glycolysis model introduced in Chapter 8 was used to create molecular visualizations (which play a central role in chemistry and biology research due to their effectiveness in revealing information on complex molecular structure and dynamics). Figure 16.13 illustrates the animation of step 6, which begins at the presence of three molecules of nicotinamide adenine dinucleotide (NAD+).

Finally, we show the application of CD++/Maya to the emergency evacuation model introduced in Chapter 10. Figure 16.14 illustrates the Maya scene file created specifically for the evacuation model. This static scene constitutes a realistic visual framework of the building under study and provides the background for the three-dimensional animation.

Figure 16.15 shows the animation for the emergency evacuation at different virtual times and from different viewpoints. Each human figure is represented by a three-dimensional icon. The engine retrieves the log data and relocates human icons based on the coordinates and values of the cells. The resulting frame-based motion of human figures allows tracing each individual throughout the building and gaining deeper insight into the evacuation process as a whole. The three-dimensional rendering also gives more details about the building than the symbolic two-dimensional animation.

Figure 16.16 shows an extended version of the evacuation model applied to the streets of downtown Montreal (where the SAT building is located). The figure shows the Cell-DEVS simulation of the pedestrians in the area and the three-dimensional visual results with CD++/Maya.

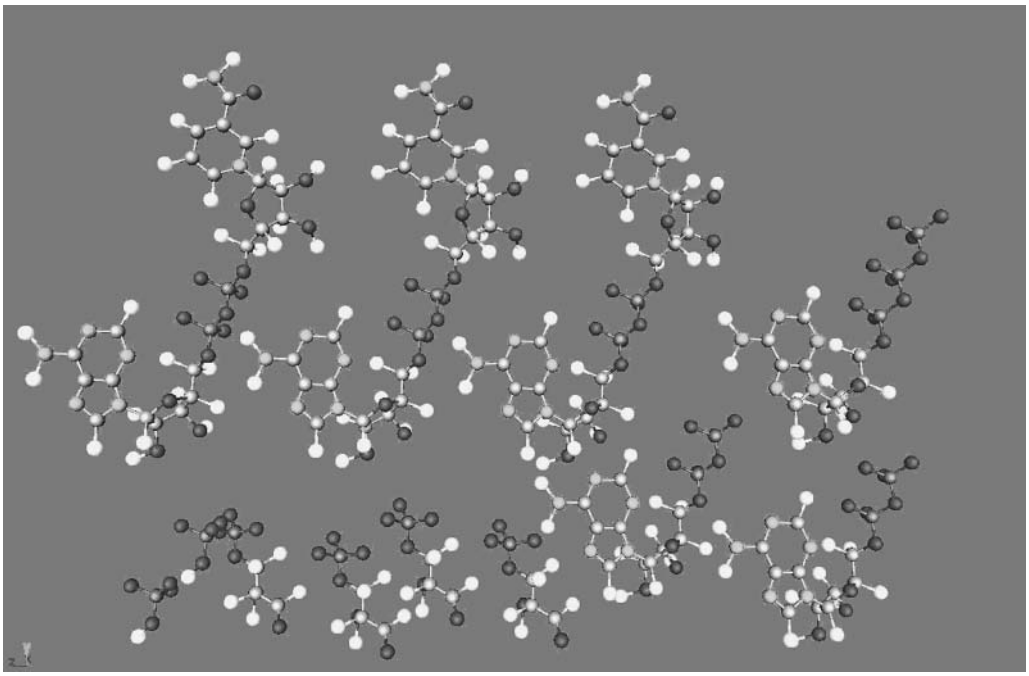


FIGURE 16.13 Animation of the glycolysis process in CD++/Maya. (From Djafarzadeh, R. et al. 2005. *Proceedings of 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference,* San Diego, CA.)

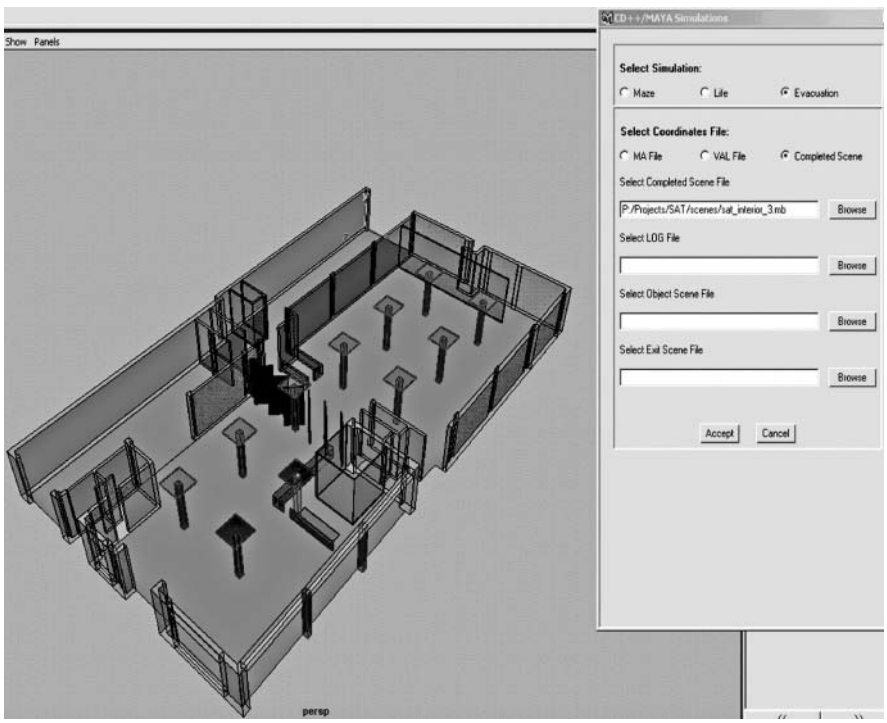


FIGURE 16.14 Background for the evacuation model using CD++/Maya.

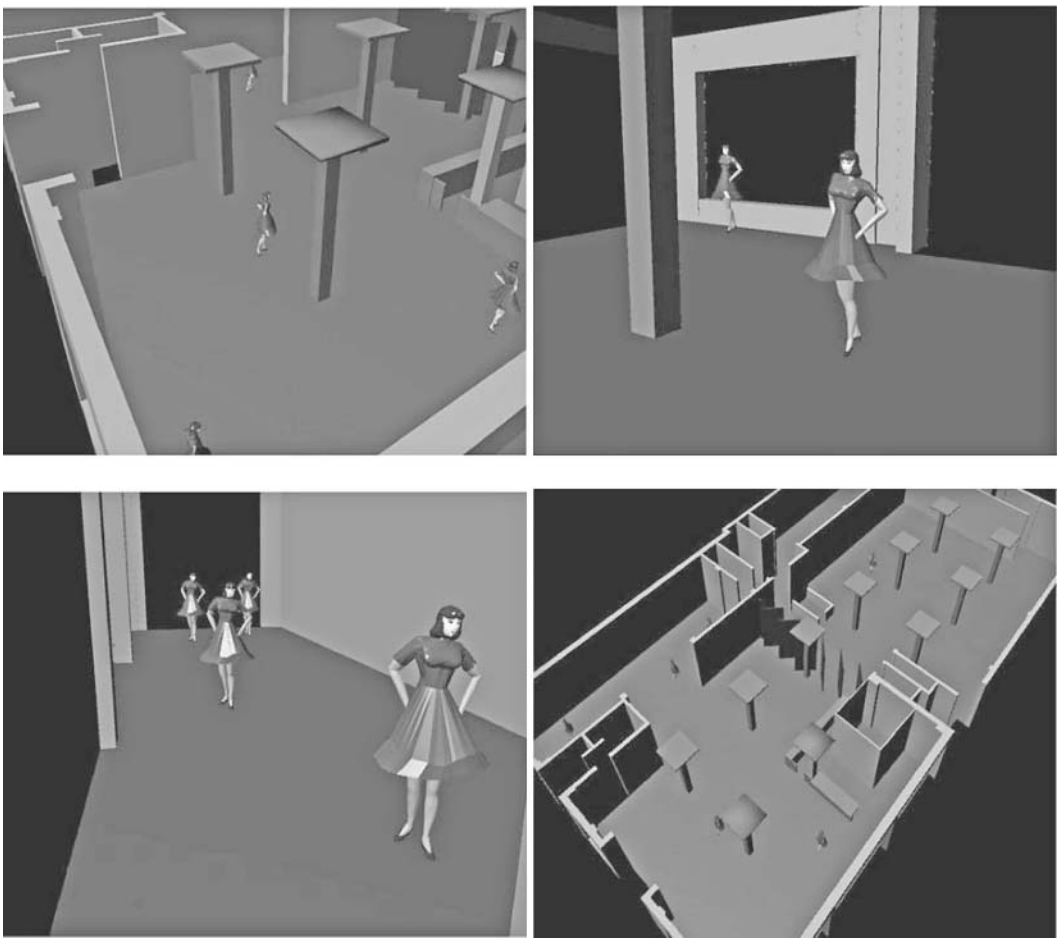


FIGURE 16.15 Animation of the emergency evacuation model using CD++Modeler and CD++/Maya.

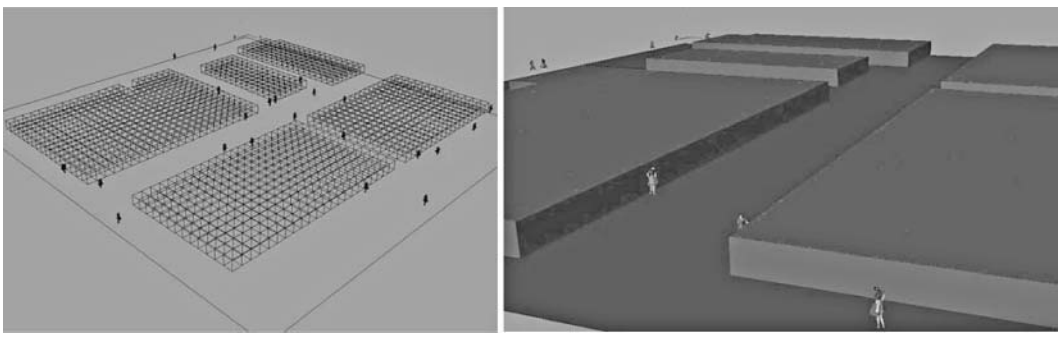


FIGURE 16.16 Animation of the crowd model using CD++Modeler and CD++/Maya.

16.4 DEVSVIEW—OPENGL-BASED TOOL FOR VISUALIZATION OF DEVS AND CELL-DEVS MODELS

Although CD++/Maya is a powerful visualization engine for creating advanced three-dimensional animation of both DEVS and Cell-DEVS models, its cost can be very high (in terms of software

installation size, advanced hardware requirements, and licensing issues). This makes it ideal for large projects where high performance and visualization quality are the major concerns. In order to provide an alternative low-cost visualization method for academic purposes, we developed an open source toolkit, referred to as DEVSView [14], for visualizing CD++ simulation results based on OpenGL [4]. OpenGL is a standard specification for developing cross-platform and language-neutral applications that has been widely used in a broad range of applications (computer-aided design, virtual reality, flight simulation, etc.). We used GLUT (OpenGL utility toolkit), a cross-platform windows-based library for writing portable OpenGL programs [15].

Visualization in DEVSView consists of visual models that are directly translated from the atomic and coupled components, creating a one-to-one relationship between visual models and the corresponding simulated components. Each visual model uses a state transition system and an event animation system that can be manipulated to match the external and output messages in CD++ log files. DEVSView provides:

- controls to define and play back three-dimensional animations;
- customizable visualization of DEVS and Cell-DEVS models, which is done by associating each visual model with a state transition system (including user-defined visual states and transition rules); and
- customizable animation of events, defined by an event animation system for each visual model (creating user-specified visual effects upon the arrival of events).

At the beginning of the visualization, DEVSView parses the CD++ log file to create a visual model for each component. The visual models are then customized to follow a visual state transition system or to produce animations. Animations evolve through events transmitted to visual models involved in the message exchanges (event messages contain information about the source/destination visual models, the input/output ports through which the event is sent, simulated virtual time of the event, and its value). Based on this information, state transition rules determine how arriving events affect the visual model, and event animation rules decide the kind of animation produced for the event. Figure 16.17 shows a visual model called `pinver` (pin verifier) for the animation of the ATM model presented in Chapter 6. The visual model is currently in its *idle* state and hence displayed as a three-dimensional box (which is specified in the state transition rules). Users can edit the properties of a visual model (e.g., visual state, shape, color, label) by using the state editing panel on the left-hand side.

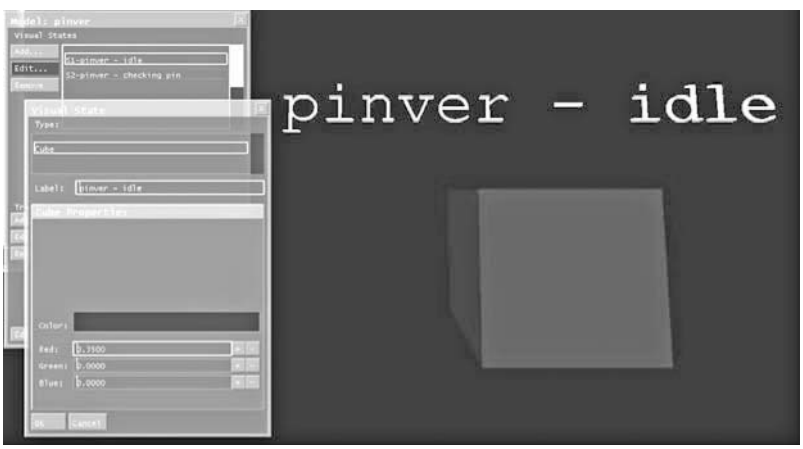


FIGURE 16.17 A visual model and the state editing panel. (From Wenhola, W., and G. Wainer. 2006. *Proceedings of DEVS Symposium, Spring Simulation Conference,* Huntsville, AL.)

Formally, a visual model is defined by

- its name;
- a list of input/output ports to exchange events with other visual models;
- its location in the three-dimensional coordinate system, orientation, and size;
- a list of visual states (one of them marked as current, defining the visual appearance of the model);
- a visual state transition system (defined as a state machine with visual states and the transitions between them), as determined by a list of state transition rules; and
- an event animation system that generates visual effects based on a list of user-specified event animation rules.

For Cell-DEVS models, all the cells share the list of visual states, transition rules, and event animation rules. This information sharing reduces file size and memory consumption, facilitates visual model definition, and improves animation performance.

The visual state transition system and event animation system process the events received by a visual model to generate the desired animation based on user-defined rules. The granularity or detail of the animation is controlled by a *value rule* mechanism that filters incoming events; an event is passed to the state transition and event animation rules only if it satisfies certain criteria. By controlling the value rules, users can focus on the animation of only those events of interest, while removing the events of less importance from the scene. Three types of value rules are supported [14]:

- **All values:** No filter is applied.
- **Equal value:** An event passes only if its value is equal to a user-specified constant.
- **Range of values:** An event passes if its value is within a given range.

Complex three-dimensional scenes containing many different visual objects may require a significant time to render. Determining which objects need to be refreshed in a frame is important for the efficiency and performance of the animation. *View culling* is the process of calculating which objects are currently in view and therefore require rendering. We used an efficient culling algorithm based on octtrees [16], a tree data structure that represents a three-dimensional space by recursively subdividing it into eight subspaces. The visual objects in a three-dimensional scene are assigned to the smallest (fittest) regions that can contain them completely, and each region is implemented as a node in the octtree. The initial space and the division of its immediate subspaces are illustrated in Figure 16.18.

The proposed culling algorithm can accurately match a graphical object to its fittest region, improving rendering performance. To do so, the octtree data structure is traversed to check whether

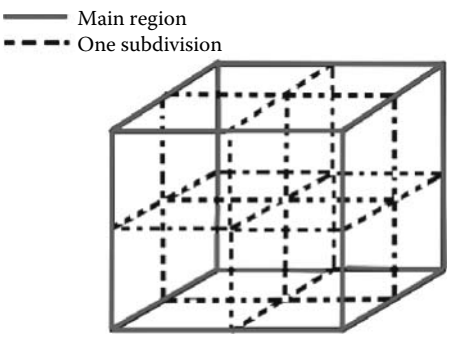


FIGURE 16.18 Partition of three-dimensional space in octtree. (From Wenhola, W., and G. Wainer. 2006. *Proceedings of DEVS Symposium, Spring Simulation Conference,* Huntsville, AL.)

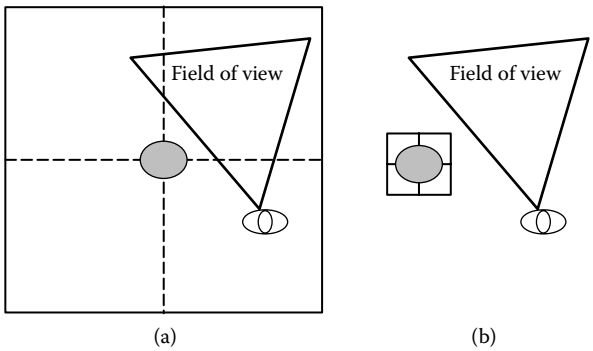


FIGURE 16.19 Illustration of the *view culling* algorithm in DEVSView. (From Wenhola, W., and G. Wainer. 2006. *Proceedings of DEVS Symposium, Spring Simulation Conference,* Huntsville, AL.)

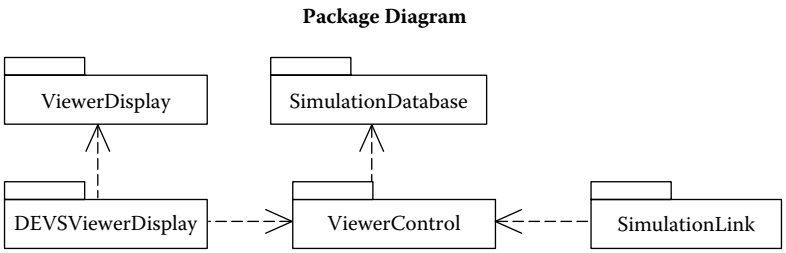


FIGURE 16.20 Package diagram for the DEVSView toolkit. (From Wenhola, W., and G. Wainer. 2006. *Proceedings of DEVS Symposium, Spring Simulation Conference,* Huntsville, AL.)

a node is in the current view or not. If a node is out of view, the entire subtree rooted at that node is pruned. On the other hand, if a node is completely in view, then the subtree extending from that node is visible and no further visibility check will be performed for its descendants. Consider, for instance, a small object located at the center of the root region, as shown in Figure 16.19(a). Because the object is associated with the root region, it can be culled only when the root node is pruned, despite the fact that it may rarely be in view. If the small object can be added to the eight fittest regions that contain it completely, as illustrated in Figure 16.19(b), then the scene will be culled much more efficiently in the octtree.

DEVSView includes three main components (Figure 16.20):

- a log parser in charge of extracting the atomic and coupled model components and the events from CD++ log files;
- a GUI that provides the animation control mechanisms and allows users to specify the graphical representation of the visual models; and
- a scene database that can efficiently organize visual models in the three-dimensional space.

`DEVSViewerDisplay` is responsible for converting user inputs into commands that can be processed by `ViewerControl`. `DEVSViewerDisplay` also controls the rendering of all three-dimensional objects, using the services provided by `ViewerDisplay` for event-driven functionalities. `SimulationLink` serves as the linkage between DEVSView and CD++. It parses the CD++ log file and notifies `ViewerControl` of new events and visual models. `ViewerControl` processes the requests from both `DEVSViewerDisplay` and `SimulationLink`. It translates the requests into a sequence of interactions with the `SimulationDatabase`, which stores the events, visual models, and related information for the visualization.

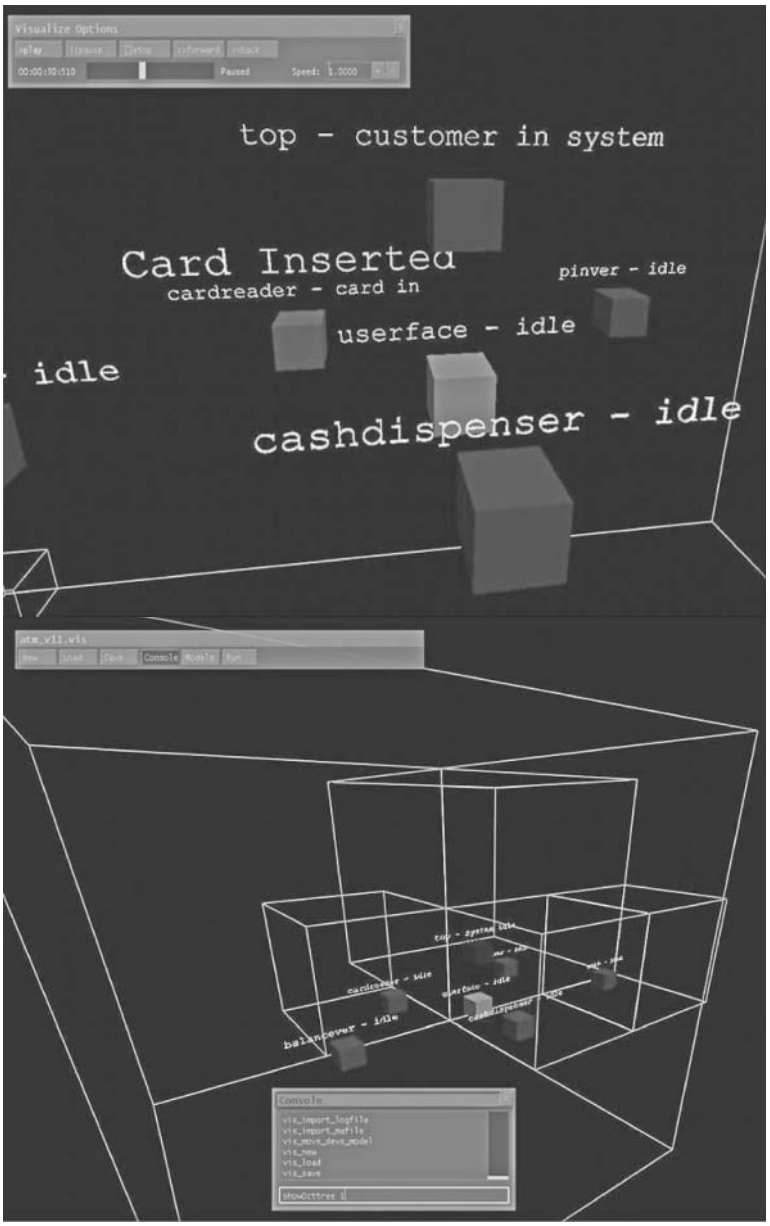


FIGURE 16.21 DEVSView animation of the ATM model. (From Wenhola, W., and G. Wainer. 2006. *Proceedings of DEVS Symposium, Spring Simulation Conference,* Huntsville, AL.)

Further details of the design and implementation of these can be found in Khan and Wainer [14], and the tool and its source code can be found at http://www.sce.carleton.ca/faculty/wainer/students/View/index.html.

Figure 16.21 shows a three-dimensional visualization of the ATM model discussed earlier. Figure 16.21(a) shows the animation when a customer inserts a debit card into the ATM machine. The event animation is shown as a three-dimensional text effect "card inserted" just beside the `CardReader` visual model. Accordingly, the `CardReader` model transitions to the "card in"

state and the `top` model changes to the "customer in system" state, while all other models are idle. The octtree regions allocated to the visual models are outlined in Figure 16.21(b).

Exercise 16.5

Modify the event animation for the ATM model. Change the colors used and the signs associated with each of the events.

Exercise 16.6

Create three-dimensional versions of the following models using DEVSView: (1) a multitask server, (2) a processor-buffer-transducer, (3) vending, and (4) an elevator.

Figure 16.22 shows the three-dimensional animation of a Cell-DEVS model that represents the design of Persian tapestry, found in *./PersianTapestry.zip*. This visualization shows the multiple layers used by the three-dimensional model in creating the shapes in the tapestry and the final design obtained.

Exercise 16.7

Modify the color scheme of the Persian tapestry model.

Exercise 16.8

Modify the color scheme and event animation for the bouncing ball simulation visualization found on the DEVSView Web page.

Exercise 16.9

Create a three-dimensional version of the same models discussed in Exercise 16.1 using DEVSView.

16.5 CD++/BLENDER

Blender [5] is an OpenGL-based, freely available three-dimensional modeling and animation software package being actively developed and widely used in a broad array of applications. It has a mature and robust feature set similar in scope and depth to other high-end three-dimensional applications such as 3ds Max [3] and Maya. Using the Blender software suite, we developed an extension for visualization of DEVS models, known as CD++/Blender, in an attempt to combine the advantages of CD++/Maya and DEVSView (i.e., software availability and visualization quality) in an integrated environment. CD++/Blender has a relatively small installation footprint, and it can be used to create advanced animations of complex models. A Python script parses CD++ log files and generates customizable visualization based on the same design as DEVSView. The design and implementation of the toolkit will not be reiterated in this section. Instead, we demonstrate the capabilities of CD++/Blender with different examples.

Figure 16.23 shows a screenshot of the CD++/Blender GUI. As usual, users can specify the model definition file and CD++ log file to be used during the visualization (animation is shown in the main window, based on a predefined three-dimensional scene file as well as the events recorded in the log file). Users can customize the animation, navigate the three-dimensional scene, and generate videos. One of Blender's strengths is that the GUI is entirely drawn in OpenGL and the content of every window can be panned, zoomed, and moved around just like other visual objects so that the screen can be organized to the user's taste for each specialized task. In addition, users can modify the GUI.

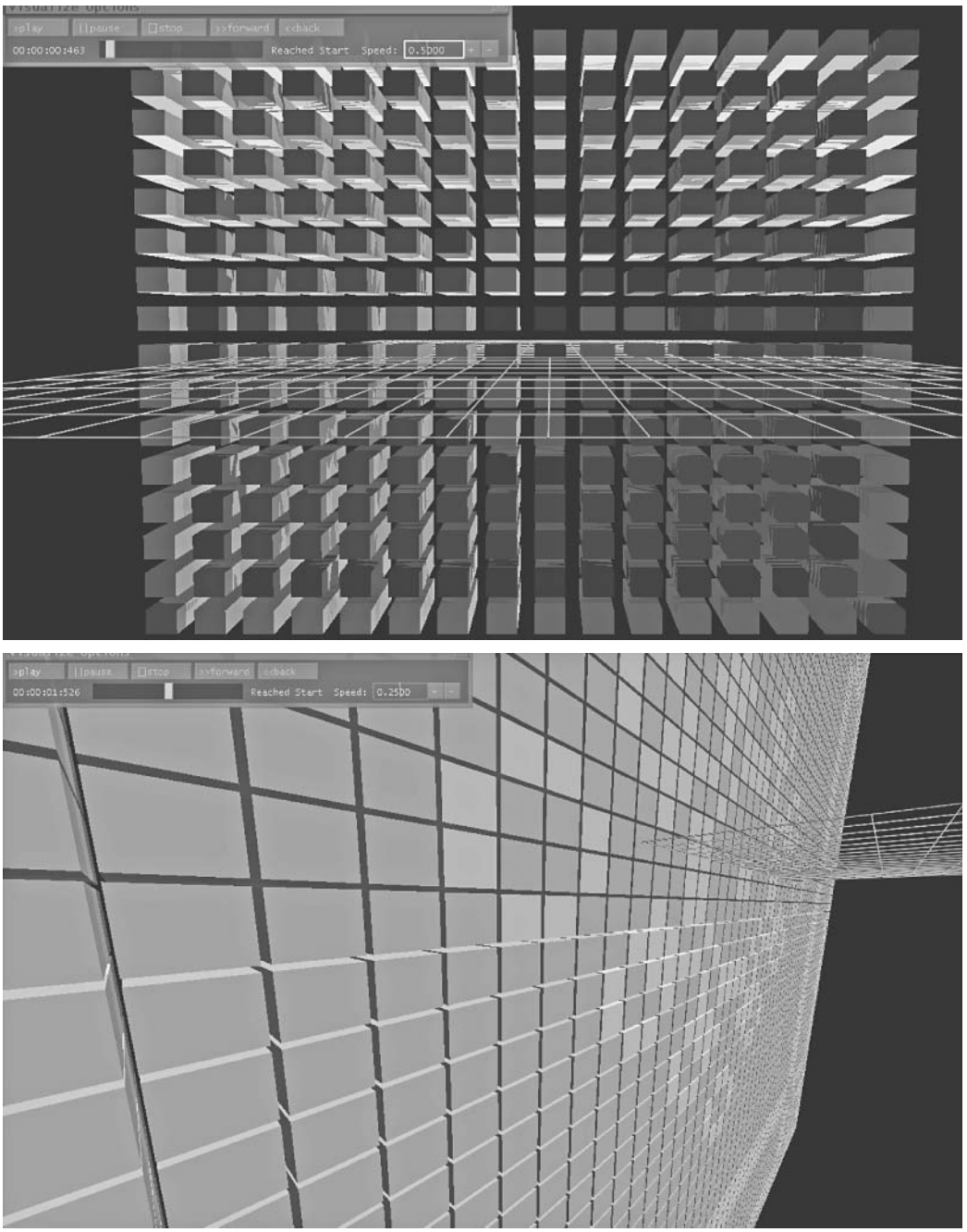


FIGURE 16.22 Animation of the Persian tapestry Cell-DEVS model.

Figure 16.24 shows how to create a three-dimensional scene file in CD++/Blender. For each scene, users can create props, and dress and paint them with different materials and textures using predefined layouts. It is also possible to define multiple scenes within a single Blender file, allowing them to share and reuse common visual objects to reduce the resulting file size.

Users can add various visual objects, ranging from simple shapes to three-dimensional human avatars. Users can run the animation forward or backward and examine the simulation results from

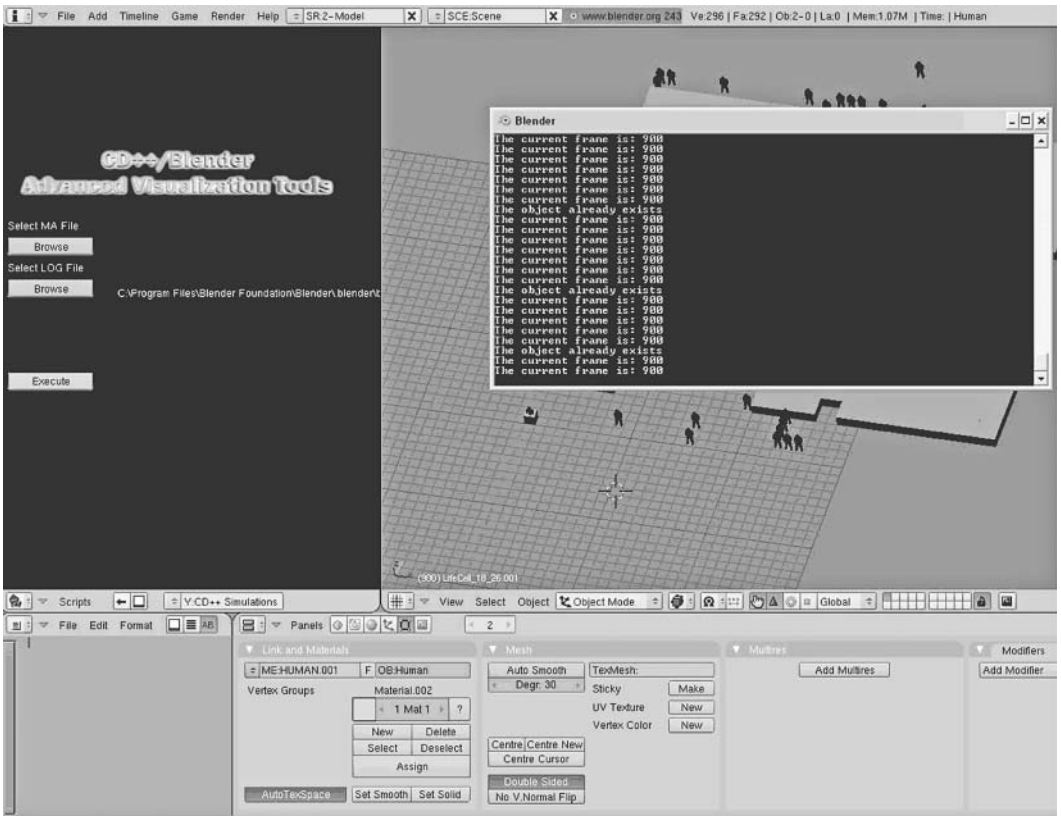


FIGURE 16.23 The CD++/Blender graphical user interface.

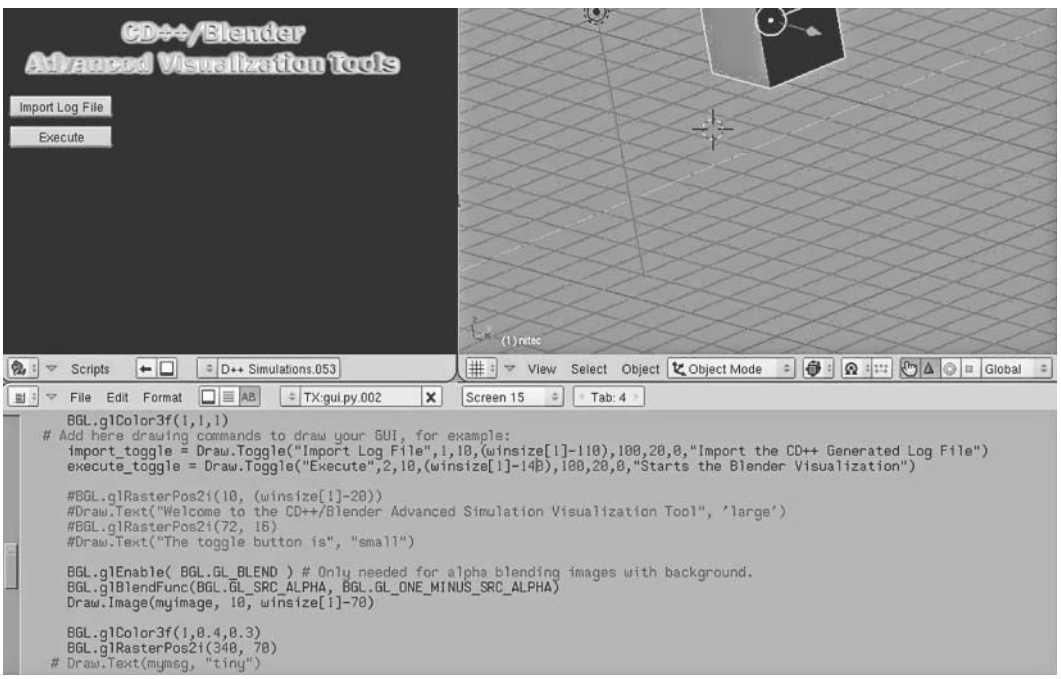


FIGURE 16.24 Creating plan file in CD++/Blender.

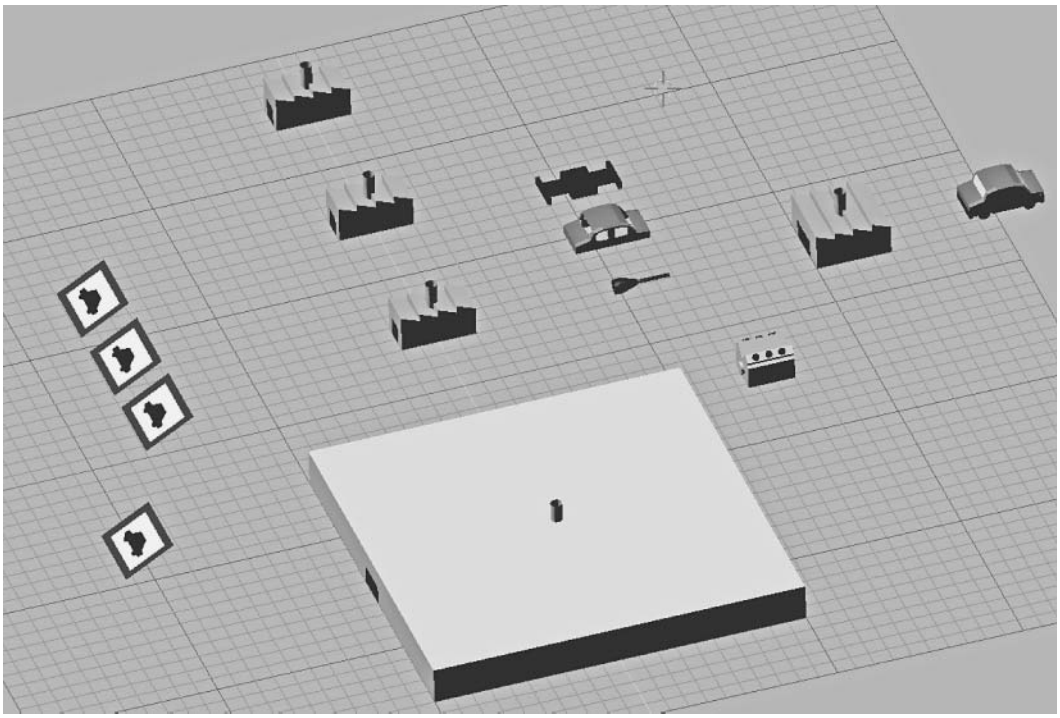


FIGURE 16.25 Animation in CD++/Blender from different viewpoints and layers.

different viewpoints by navigation in the three-dimensional virtual world. For large models, three-dimensional scenes may become exceptionally confusing due to the increased complexity. This problem is solved in CD++/Blender by virtue of Blender's native support of layers. Each layer in the scene groups the related visual objects of interest so that only the selected layers (or groups of visual objects) are rendered at any one time. This technique provides a better overview of the animation and allows the user to examine the simulation data with varying granularities. Figure 16.25 illustrates a frame of the car manufacturing model animation discussed earlier.

Exercise 16.10

Build three-dimensional visualizations using CD++/Blender for Exercises 16.1 and 16.9.

16.6 SUMMARY

This chapter introduced a detailed discussion of varied visualization facilities currently available for CD++. These facilities provide users with a variety of mechanisms to facilitate the M&S process, thereby promoting the adoption of cutting-edge M&S technologies by a wider community of practitioners and researchers. Although CD++Modeler provides some basic facilities (allowing application specialists to construct models and analyze simulation data), the two-dimensional facilities might not be adequate for complex applications, training, analysis, and live simulations with man and hardware in the loop.

CD++/VRML enables basic animations of Cell-DEVS models in a three-dimensional virtual world, and it can be expanded to provide more advanced visual results and tailored for special-purpose application domains (as we did for the urban traffic control case). In order to fulfill the needs of different user communities, we also integrated the CD++ environment with both commercial and open source software packages and developed a set of advanced toolkits for high-performance

animation of complex DEVS and Cell-DEVS models, including CD++/Maya, DEVSView, and CD++/Blender. Following the DEVS modular approach, the resulting architecture can be easily extended and adapted, facilitating the validation and verification of continuously evolving models and making them suitable for efficient online decision-making.

REFERENCES

1. Choi, W. 2008. Study on L-V-C (live-virtual-constructive) interoperation for the national defense M&S (modeling & simulation). *ICISS International Conference on Information Science and Security,* Seoul, Korea, 128–133.
2. Ames, A., D. Nadeau, and J. Moreland. 1997. *VRML 2.0 source,* 2nd ed. New York: John Wiley & Sons, Inc.
3. Pardew, L., and M. Tidwell. 2006. *Autodesk Maya and Autodesk 3ds Max side-by-side.* Boston: Course Technology Press.
4. Segal, M., and K. Akeley. 2006. The OpenGL graphics system: A specification. Silicon Graphics, Inc.
5. Roosendaal, T., and S. Selleri. 2004. *The official Blender 2.3 guide: Free 3D creation suite for modeling, animation, and rendering.* San Francisco, CA: No Starch Press.
6. Behr, J., P. Dähne, and M. Roth. 2004. Utilizing X3D for immersive environments. *Proceedings of Web3D '04, Ninth International Conference on 3D Web Technology,* Monterey, CA, 71–78.
7. Roehl, B., and J. Couch. 1997. *Late night VRML 2.0 with Java.* Hightstown, NJ: Ziff–Davis Publishing Co.
8. Wainer, G., and W. Chen. 2003. A framework for remote execution and visualization of cell-DEVS models. *Simulation* 79:626–647.
9. Wainer, G., S. Borho, and J. Pittner. 2001. Defining and visualizing models of urban traffic. *Proceedings of 1st Mediterranean Multiconference on Modeling and Simulation,* Genoa, Italy.
10. Sourceforge. 2007. *JHotDraw Open-Source Project.* URL: http://sourceforge.net/projects/jhotdraw
11. Wainer, G. 2007. Developing a software toolkit for urban traffic modeling. *Software Practice and Experience* 37:1377–1404.
12. Khan, A., and G. Wainer. 2005. Advanced visualization of DEVS and cell-DEVS models in Maya. *Proceedings of the SISO Spring Interoperability Workshop,* San Diego, CA.
13. Djafarzadeh, R., T. Mussivand, and G. Wainer. 2005. Modeling energy pathways in cells. *Proceedings of 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference,* San Diego, CA.
14. Wenhola, W., and G. Wainer. 2006. DEVSView: A tool for visualizing CD++ simulation models. *Proceedings of DEVS Symposium, Spring Simulation Conference,* Huntsville, AL.
15. Kilgard, M. J. 1996. The OpenGL utility toolkit (GLUT) programming interface: API Version 3. Available: http://www.Opengl.org/resources/libraries/glut/glut-3.spec.pdf
16. Jackins, C. L., and S. L. Tanimoto. 1980. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics & Image Processing* 14:249–270.