Ph.D. Dissertation

Hybrid System Modeling Using the Parallel DEVS Formalism and the Modelica Language

Victorino Sanz Prat

Computer Scientist

Departamento de Informática y Automática

Escuela Técnica Superior de Ingeniería Informática

Universidad Nacional de Educación a Distancia

Madrid, 2010

Department Informática y Automática
E.T.S. de Ingeniería Informática

Title Hybrid System Modeling
Using the Parallel DEVS Formalism
and the Modelica Language

Author Victorino Sanz Prat

Degree Computer Scientist
Facultad de Informática
Universidad Politécnica de Madrid

Supervisors Alfonso Urquía Moraleda
Sebastián Dormido Bencomo

To my wife, Irene.

Acknowledgements

I would like to express my eternal gratitude to the people that has contributed, to a greater or lesser extent, to the development of this dissertation.

First of all, to my supervisors Prof. Alfonso Urquía and Prof. Sebastián Dormido, for giving me the opportunity to work at the department, to enter the modeling and simulation world, for helping me and supporting my work in all its terms. Specially, I would like to thank the support offered by Prof. Alfonso Urquía for introducing me into this research work, for guiding me during its development, for reviewing my work and always having the best advice or comment.

To Prof. François E. Cellier, Prof. Gabriel A. Wainer and Prof. Mamadou D. Seck, for inviting me to visit them in their home institutions, for offering good advices and critics, and for the shared work.

To Prof. Carla Martín and Prof. Miguel Ángel Rubio, for sharing their knowledge about Modelica and about modeling and simulation in general, for their friendship, and for being at the other end of the rope.

To Prof. José Manuel Díaz, for all his good advices and help.

To the rest of my colleagues at the department. Specially those who shared a bit their lives with me inside the 5.06, and also the one currently dedicated to pottery. Thank you for the encouraging conversations, discussions, shared coffees and jokes.

To Prof. Cesar de Prada and Prof. Daniel Sarabia, for the information provided about their model of the supermarket refrigeration system developed using EcosimPro.

To my old colleagues at Sun Microsystems, Alberto Ambroj, David Piqueras, Manuel Paniagua and Jose Antonio Sanfelix, for pushing me to leave them and start a new life at the university.

Finally, I would like to thank the immeasurable support provided by my family and friends, always encouraging me to carry on and giving me further more than the best they can. Specially to my wife, Irene, for her infinite patience, so many times not deserved, for sharing her life with me and for the received love.

Contents

List of Figures

List of Tables

List of Acronyms

ACSL	Advanced Continuous Simulation Language
ATM	Automatic Teller Machine
CMRG	Combined Multiple Recursive Generators
CSSL	Continuous System Simulation Language
DAE	Differential and Algebraic Equations
DASSL	Differential Algebraic System Solver
DESS	Differential Equation System Specification
DEVS	Discrete Event System Specification
DEV&DESS	Discrete Event and Differential Equation System Specification
DTSS	Discrete Time System Specification
EOO	Equation-Based Object-Oriented
FDTD	Finite-Difference Time-Domain
FEM	Finite Elements Method
FIFO	First In First Out
GUI	Graphical User Interface
HVF	Higher Value First
IPC	Inter Process Communication
LIFO	Last In First Out

LVF	Lower Value First
MSL	Modelica Standard Library
ODE	Ordinary Differential Equations
ONoC	Optical Network on Chip
P-DEVS	Parallel Discrete Event System Specification
QSS	Quantized State System
RNG	Random Number Generator
SoC	Systems on Chip
TCP/IP	Transfer Control Protocol / Internet Protocol

1

Introduction, Objectives and Structure

1.1 Introduction

In the physical modeling paradigm, systems are described in a modular way. A system is decomposed into subsystems, and each subsystem is described using an interface, balances of mass, energy and momentum, and material equations. The interface is used to describe the relations between subsystems. A model is considered as a constraint between model variables [Åström et al., 1998].

The object-oriented modeling methodology facilitates the description of models using acausal equations, which makes the application of the physical modeling paradigm possible. In object-oriented models, the mathematical manipulations needed to simulate the model in a digital computer (e.g., computational causality assignment, algebraic loop tearing and index reduction) are automatically performed by the modeling environment. This represents a considerable advantage when compared with the block-diagram modeling paradigm, where the model has to be manually manipulated by the modeler in order to simulate it [Åström et al., 1998]. This methodology also facilitates the design, programming, reuse and maintenance of models [Cellier, 1996].

Modelica is a general-purpose modeling language, freely distributed under its own license, that allows the description of models following the object-oriented methodology. Similarly to effort performed in the 60's to describe the CSSL

standard [Augustin et al., 1967], Modelica constitutes an international effort to standardize the description of models using this methodology [Mattsson et al., 1998], in comparison with the multiple languages previously developed. The use of an standard language facilitates the exchange of models between different users and tools. Modelica includes characteristics from languages like ALLAN [Jeandel et al., 1997], Dymola [Elmqvist, 1978], NMF [Sahlin et al., 1996], ObjectMath [Fritzson et al., 1995], Omola [Andersson, 1989], SIDOPS+ [Breuneuse and Broenink, 1997] and Smile [Kloas et al., 1995].

The first version of Modelica appeared in September 1997. Since then, Modelica has been increasingly used to describe models. The industry and the research communities have widely accepted the language, as seen in the increasing number of contributions and participants in the International Modelica Conferences [Modelica, 2010]. Also, the number of Modelica libraries developed (both free and commercial) has increased. Even the language itself has been in continuous development, being the 3.1 its last version [Modelica Association, 2009].

The functionalities of the Modelica language are extended by the development of libraries of components. Modelica libraries are compilations of components jointly designed to facilitate the description of models using different formalisms or in different domains. Currently, Modelica supports the description of models from multiple domains (e.g., electrical, mechanical, thermodynamical and chemical) by means of different libraries [Modelica Libraries, 2010]. Some of the supported formalisms are the ODE, DAE, Bond Graphs [Cellier and Nebot, 2005] and System Dynamics [Cellier, 2008].

The description of discrete-event and hybrid models using Modelica is facilitated due to the included functionalities to manage time and state events [Otter et al., 1999; Mattsson et al., 1999]. Using these hybrid modeling functionalities, Modelica supports multiple discrete-event modeling formalisms, like Petri Nets [Mosterman et al., 1998], StateGraphs [Otter et al., 2005] and StateCharts [Ferreira and de Oliveira, 1999].

As already mentioned, the use of a methodology or a mathematical formalism to describe models facilitates its development, maintenance, reuse and adaptation

to different experiments and situations. The Parallel DEVS (P-DEVS) formalism [Chow, 1996] and the process-oriented paradigm [Law, 2007] are two widely used methodologies to describe discrete-event systems. Multiple contributions have also discussed their application to the description of hybrid dynamic models [Vangheluwe, 2000; Giambiasi and Carmona, 2006; Kelton et al., 2007].

The Classic DEVS formalism was described by Zeigler [1976]. P-DEVS is an extension of the Classic DEVS formalism that allows simultaneous occurrences of events, removing the restriction of their sequential management. DEVS has been also considered as the equivalent of the differential equations formalism for describing discrete-event systems [Zeigler, 1989]. Other discrete-event system modeling methodologies and formalisms can be described using DEVS [Vangheluwe, 2000].

The process-oriented paradigm is one of the three "world-views" [Kiviat, 1969] commonly used to describe discrete-event systems. It provides an intuitive mechanism to describe a system as a series of interconnected processes, instead of occurrences of events. Process-oriented models represent systems as a flow of entities that flow through the processes of the system using the available resources [Law, 2007]. Arena, which is internally implemented using the SIMAN language [Pegden et al., 1995], is a simulation environment widely used in academia and industry to describe discrete-event systems following the process-oriented approach [Kelton et al., 2007].

The feasibility of describing Classic DEVS models in Modelica was demonstrated in Fritzson [2003]. Also, a Modelica library called ModelicaDEVS [Beltrame and Cellier, 2006] was developed for modeling continuous-time systems using the Classic DEVS formalism and the QSS integration algorithms [Kofman, 2004]. These DEVS implementations in Modelica define the communication between models as a change in the value of a boolean variable, and using the mentioned hybrid modeling functionalities to perform the management of discrete-events. However, as it will be discussed, the description of P-DEVS and process-oriented models require additional mechanisms currently not present in Modelica. The description of the P-DEVS model communication mechanism in

Modelica is not straightforward. None of the existing Modelica libraries support the P-DEVS formalism, neither the process-oriented approach to develop models of discrete-event systems in Modelica. The use of P-DEVS to formally describe discrete-event systems in Modelica could facilitate the construction, maintenance and reuse of models and the introduction of other formalisms, like the process-oriented approach, into Modelica.

1.2 Objectives

The main objective of this Ph.D. dissertation is to include new functionalities in the Modelica language to describe the discrete-event part of hybrid models using the P-DEVS formalism and the process-oriented paradigm. The use of these methodologies combined with the object-oriented modeling methodology, supported by Modelica, will facilitate the description of hybrid dynamic models. To achieve this objective, the completion of the following tasks is proposed:

1. The identification and analysis of the requirements to describe P-DEVS and process-oriented models in Modelica. This task constitutes the foundations of the work presented in this dissertation.

2. A new Modelica library, named DEVSLib, will be designed and developed to facilitate the description of discrete-event models using the P-DEVS formalism. The definition of new atomic and coupled models using this library will be as close as possible to the formal P-DEVS specification of the model.

3. The communication between P-DEVS models follows a message passing structure. The current Modelica mechanism for model communication is based on the connection of model variables, so it does not facilitate the transmission of structured information between models. A message passing communication mechanism in Modelica will be proposed, to facilitate the description of P-DEVS models. This mechanism could be used to commu-

nicate structured information between models using multiple configurations (1:1, 1:N, N:1). Therefore:

   (a) A message passing mechanism in Modelica will be proposed, together with the language modifications needed to facilitate the communication of models using the proposed mechanism.

   (b) A partial implementation of the proposed message passing mechanism will be developed, using the current Modelica functionalities. This mechanism will be used in the development of the DEVSLib library.

4. The P-DEVS formalism will be used to describe the behavior of some of the components of the SIMAN language. This formal specification will facilitate the implementation of these components in Modelica, using the new DEVSLib library. In this way, the P-DEVS formalism will be used to support process-oriented modeling in Modelica.

5. Two new libraries, named SIMANLib and ARENALib, will be designed and implemented in Modelica to support the description of discrete-event models using the process-oriented modeling paradigm. These two new libraries will provide similar model description and analysis functionalities to the Arena simulation environment and the SIMAN language. The SIMANLib library will reproduce the functionalities of the Create, Dispose, Queue, Seize, Delay, Release, Branch, Count, Tally and Assign SIMAN blocks. These blocks have been selected because they constitute the basic components required to describe the majority of the processes found in logistic systems. The ARENALib library will reproduce the functionalities of the Create, Dispose, Process, Record, Decide and Assign modules of the Arena's BasicProcess panel. These components, similarly to Arena, will be constructed using the components of the SIMANLib library.

6. Discrete-event system models are sometimes stochastic. Another task of this dissertation will be to provide the Modelica language with stochastic modeling functionalities. Neither the Modelica language nor the Modelica

Standard Library include any random number generation functionality. A new free Modelica library, named RandomLib, will be developed to facilitate the generation of uniform random numbers and random variates. The functionalities included in RandomLib will replicate the stochastic modeling functionalities included in the Arena simulation environment, in order to facilitate the validation of the developed models.

7. Interface models will be developed and included in the implemented libraries to make them compatible with the rest of the Modelica libraries. This is, to allow the connection of discrete-event models constructed using DEVSLib, SIMANLib and ARENALib with models constructed using other Modelica libraries. These interface models will translate the discrete-event messages generated by the P-DEVS models into discrete-time signals, and discrete-time and continuous-time signals into messages. The interaction between process-oriented components and continuous-time models will also be studied. Some process-oriented components will be extended with additional functionalities to interact with continuous-time models, and construct hybrid process-oriented models.

8. The description of hybrid control systems will also be studied. In this case, discrete-event controllers will be described using the P-DEVS formalism and the continuous-time plants using other Modelica libraries. Both parts will be connected using the developed interface models.

9. Finally, a set of models to illustrate the use of the developed libraries will be implemented. These models will include both discrete-event stochastic models and deterministic hybrid models. Two models, of an automatic teller machine and the predator-prey interactions described by Lotka and Volterra, will be used to describe the construction of discrete-event models using the P-DEVS formalism in Modelica, by means of the DEVSLib library. Two hybrid models of a two tank system and an opto-electrical communication system will be used to present the description of hybrid dynamic systems using DEVSLib. The application of the P-DEVS formalism

to the description of hybrid control systems will be discussed using models of a supermarket refrigeration system and a crane with an embedded discrete controller. Models of a bank teller, a restaurant and an electronic assembly factory will be used to present the construction of process-oriented models using SIMANLib and ARENALib. Finally, models of an orange juice factory, a tank-level controller and a soaking-pit furnace system will be described to show the hybrid process-oriented functionalities included in SIMANLib and ARENALib.

1.3 Document Structure

The description of the developments presented in this dissertation is organized in the following chapters and appendices:

Chapter 2. The evolution and current state of the different modeling and simulation techniques that can be applied to hybrid system modeling is presented in this chapter. The evolution of continuous-time and discrete-event modeling methodologies is described, detailing the characteristics of the Modelica language, the P-DEVS formalism and the Arena simulation environment. The evaluation of the described methodologies has been used as an starting point for the development of this Ph.D. dissertation.

Chapter 3. The requirements needed to describe P-DEVS models in Modelica are identified and discussed in this chapter. The identification of the conceptual differences between Modelica and P-DEVS supposes the first step in the development of this dissertation. The rest of the work presented in this dissertation refers to the approaches taken and developed in order to facilitate the description of P-DEVS models in Modelica.

Chapter 4. The design and implementation of a message passing mechanism in Modelica is described in this chapter. This has been the most important challenge encountered during the development of this dissertation, and represents the

cornerstone of the performed works. The approaches studied and implemented during the development of this mechanism are discussed. The selected approach, its use and the ports defined to establish communication between models are detailed.

Chapter 5. The DEVSLib library, that supports the P-DEVS formalism in Modelica, is described in this chapter. The general architecture of the library and its components are described. The developed message passing mechanism is used to describe the communication between models in DEVSLib. The presentation of the library is performed from the point of view of the developer, describing the implementation details.

Chapter 6. The construction of discrete-event models using the DEVSLib library is described in this chapter. Two case studies are discussed to describe the use of the DEVSLib library. A model of an automatic teller machine is described as an example of a pure discrete-event system modeled using P-DEVS. Also, a discrete-event model of a Lotka-Volterra predator-prey system, which is defined using differential equations, is discussed. This model has been constructed using the QSS integration methods developed using DEVSLib, to demonstrate that the library can be used to model multiple types of DEVS-based systems.

Chapter 7. The description of hybrid systems using the DEVSLib library is presented in this chapter. The behavior of the interface models included in the library, in order to combine P-DEVS models with other Modelica libraries is detailed. The use of these interfaces to describe hybrid systems is presented by means of two case studies. The first case study represents a system with two tanks controlled by a discrete-event controller. This case study is included to describe the interactivity between the continuous-time part (i.e., the tanks and valves) with the discrete-event behavior (i.e., the controller), which is algorithmically described. The second case study represents an opto-electronic communication interface, where the electronic part is modeled using continuous-time Modelica models and the optical part is modeled

using DEVSLib. This case study is included to show the versatility of using DEVSLib and Modelica to describe multi-domain hybrid systems.

Chapter 8. The application of the modeling functionalities included in DEVSLib to the description of hybrid control systems is presented in this chapter. These functionalities are described by means of two case studies: a crane system with an embedded discrete controller and a supermarket refrigeration system. Both models are used to present the feasibility of describing hybrid control systems using DEVSLib combined with other Modelica models. An evaluation of different approaches (using Modelica, atomic DEVSLib models or coupled DEVSLib models) to describe the controllers is also discussed.

Chapter 9. The functionalities required to describe process-oriented models in Modelica are analyzed in this chapter. Process-oriented models communicate using a mechanism equivalent to P-DEVS models, and so the described message passing mechanism is used to facilitate their description. However, additional funcitionalities are required in order to model systems following the process-oriented approach. These functionalities include the management of the entities and their flow through the system, and the management of variable-size data structures to store user-defined information and statistical indicators.

Chapter 10. A new Modelica library, named SIMANLib, that includes low-level functionalties to describe process-oriented models is presented in this chapter. The architecture of the library, its design, components and implementation are detailed. SIMANLib includes components equivalent to the Create, Dispose, Queue, Seize, Delay, Release, Branch, Count, Tally and Assign SIMAN blocks. The description of the SIMANLib components has been performed using the P-DEVS formalism, and their development has been performed using the DEVSLib library. These components are found in the majority of models developed to describe logistic systems, and are also used to describe the internal behavior of the ARENALib components. A case study of a restaurant modeled using SIMANLib is described.

Chapter 11. Another new Modelica library, named ARENALib, that includes high-level functionalities to describe process-oriented models is presented in this chapter. ARENALib functionalities are at a higher-level when compared with SIMANLib functionalities, that describe simpler actions and processes. The architecture of the library, its design, components, implementation and use are detailed. ARENALib includes components equivalent to the Create, Dispose, Process, Record, Decide and Assign modules of the Arena's BasicProcess panel. A case study of a electronic assembly factory is described.

Chapter 12. The functionalities included in SIMANLib and ARENALib for the development of hybrid systems following the process-oriented approach are described in this chapter. The use of these functionalities to describe systems is shown by means of three case studies: an orange juice canning factory, a tank-level control system and a soaking-pit furnace system. These case studies include different characteristics described using the hybrid process-oriented modeling functionalities included in SIMANLib and ARENALib.

Chapter 13. The implementation of the RandomLib library is presented in this chapter. This new library can be used, in combination with the other developed libraries, to describe discrete-event stochastic models. It includes a random number generator and multiple random variates generation functions.

Chapter 14. The conclusions of this dissertation, as well as some ideas for future research work, are presented in this chapter.

Appendix A. This appendix contains a description of a semaphore model developed using Modelica. This model was designed and implemented during the development of the message passing mechanism as a synchronization method for P-DEVS models communication. However, due to the poor performance obtained using this model it was discarded in the final implementation of the communication mechanism. Because of this its description is not included in Chapter 4.

1.4 Publications

The following contributions have been published during the development of this Ph.D. dissertation:

1. **V. Sanz**, A. Urquia and S. Dormido. ARENALib: A Modelica Library for Discrete-Event System Simulation. In *Proceedings of the* 5^{th} *International Modelica Conference*, Vienna, Austria, 2006, pp 539–548.

2. **V. Sanz**, A. Urquia and S. Dormido. DEVS Specification and Implementation of SIMAN Blocks Using the Modelica Language. In *Proceedings of the Winter Simulation Conference 2007*, Washington, D.C., USA, 2007, p 2374.

3. **V. Sanz**, A. Urquia and S. Dormido. Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling. In *Proceedings of the* 2^{nd} *International Workshop on Equation-Based Object-Oriented Languages and Tools*, Paphos, Cyprus, 2008, pp 83–94.

4. **V. Sanz**, A. Urquia and S. Dormido. Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling. *Simulation News Europe*, 18(2), 2008, pp 42–53.

5. **V. Sanz**, S. Jafer, G. Wainer, G. Nicolescu, A. Urquia and S. Dormido. Hybrid Modeling of Opto-Electrical Interfaces Using DEVS and Modelica. In *Proceedings of the DEVS Integrative M&S Symposium, Spring Simulation Multiconference*, San Diego, CA, USA, 2009.

6. **V. Sanz**, F.E. Cellier, A. Urquia and S. Dormido. Modeling of the ARGESIM "Crane and Embedded Controller" System using the DEVSLib Modelica Library. In *Proceedings of the* 3^{rd} *IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09)*, Zaragoza, Spain, 2009.

7. F.E. Cellier and **V. Sanz**. Mixed Quantitative and Qualitative Simulation in Modelica. In *Proceedings of the* 7^{th} *International Modelica Conference*, Como, Italy, 2009, pp 86–95.

8. **V. Sanz**, A. Urquia and S. Dormido. Parallel DEVS and Process-Oriented Modeling in Modelica. In *Proceedings of the* 7^{th} *International Modelica Conference*, Como, Italy, 2009, pp 96–107.

9. **V. Sanz**, A. Urquia and S. Dormido. Integrating Parallel DEVS and Equation-Based Object-Oriented Modeling. In *Proceedings of the DEVS Integrative M&S Symposium, Spring Simulation Multiconference*, Orlando, FL, USA, 2010.

1.5 Research Projects

The works required for the development of this dissertation have been performed in the framework of the following research projects:

1. "Herramientas interactivas para el modelado, visualización, simulación y control de sistemas dinámicos", *CICYT, DPI2004-01804*, January 2004 – December 2006, Principal researcher: Prof. Dr. Sebastián Dormido Bencomo.

2. "Control de sistemas complejos en la logística y producción de bienes y servicios. Acrónimo: COSICOLOGI-CM", *IV PRICIT 2005–2008, Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid, Ref. S-0505/DPI/0391*, January 2005 – December 2008, Principal researcher: Prof. Dr. Sebastián Dormido Bencomo.

3. "Modelado, simulación y control basado en eventos", *CICYT, DPI2007-61068*, October 2007 – September 2012, Principal researcher: Prof. Dr. Sebastián Dormido Bencomo.

2

Hybrid System Modeling and Simulation

2.1 Introduction

Continuous-time and discrete-event modeling are two of the main paradigms used to describe dynamic systems. The use of one or the other paradigm is dictated by the characteristics of the system and the requirements of the study to be performed.

Many systems include a combination of interacting continuous-time and discrete-event dynamics. This type of systems are denominated *hybrid dynamic systems*.

The techniques used to describe hybrid systems follow these approaches:

- The continuous-time based approach, that focuses on developing continuous-time methodologies and techniques that include functionalities to manage events.
- The discrete-event based approach, that is based on the inclusion of continuous-time simulation functionalities (i.e., numerical integration of algebraic and differential equations) to discrete-event simulation tools.

Formalisms, modeling languages or simulation tools are usually designed for modeling and simulation of either continuous-time or discrete-event models. They are later extended including some functionalities to describe hybrid systems.

A description of the evolution of both continuous-time and discrete-event modeling and simulation approaches is discussed in this chapter. The purpose of this description is to identify the state-of-the-art methodologies for describing continuous-time and discrete-event models, discuss their functionalities for describing hybrid systems and study the possibility of combining both methodologies in order to improve the description of hybrid dynamic systems. This analysis has been considered as the starting point for the development of this dissertation.

2.2 Continuous-time Modeling

Continuous-time models are defined by the continuous variation of their state variables and the time. The values that can be assigned to the state variables can change an infinite number of times in a finite time interval (i.e., are of type real). Continuous-time models can be described using a combination of differential and algebraic equations, depending on the characteristics of the system and the physical laws that describe its behavior.

2.2.1 Evolution of Continuous-time Modeling

During the first half of the 20th century, the simulation of continuous-time models was performed using physical devices to represent the equations used to describe the behavior of the system. For instance, the mechanical differential analyzer developed by Bush [1931] used angles to represent variables, ball and disc integrators, and gear boxes for function generation.

Lately, it was demonstrated that these simulations could be performed using electronic circuits [Ragazzini et al., 1964]. This improvement facilitated the experimental set up of the problem and the measure of the variables of the system, now represented using electrical voltages.

However, simulation studies using this approach were tedious [Åström et al., 1998]. The equations used to describe the system had to be transformed and represented using basic operations (like integration, addition and multiplication).

The representation of magnitudes and resolutions was limited and scaling of variables was usually required. Multiple interconnections between model components were required to represent functions relating several variables.

Basically, the work of the modeler was to represent the system by means of equations, and also to find a suitable representation of those equations in order to simulate them. Multiple mathematical manipulations had to be manually performed to the equations in order to obtain this representation (e.g., the removal of algebraic loops), thus leading to error prone modeling. This approach was denominated analog simulation [Jackson, 1960].

The analog simulation evolved with the use of digital computers to perform simulation studies. The systems of ordinary differential equations could be programmed in a digital computer, transforming the differential equations into difference equations. The simulation was performed using a numerical integration algorithm. These numerical algorithms play a basic role in simulation and have been widely studied [Atkinson, 1989; Butcher, 2003; Cellier and Kofman, 2006]. Multiple numerical algorithms have been developed to simulate the behavior of ordinary differential equations (ODE's), like the Runge-Kutta methods [Butcher, 2003], and differential-algebraic equations (DAE's), like the DASSL integration algorithm [Petzold, 1983; Brenan et al., 1989].

In order to facilitate the description of models, the CSSL standard [Augustin et al., 1967] appeared as a unification of the concepts and language structures of the simulation tools available up to date. A description of the CSSL standard and its functionalities can be found in Rimvall and Cellier [1986]. Multiple implementations of modeling and simulation environments followed the CSSL standard. ACSL [Mitchell and Gauthier, 1976] became the de-facto standard for continuous-time modeling and simulation. ACSL was also improved with the inclusion of constructs for combined continuous/discrete modeling.

However, even with the use of technological advances like the digital computers, the development of models was still tedious. The use of physical devices was substituted with the use of programming languages and numerical integrators, but the task of modeling was still equivalent to that in the analog simulation.

The modeler had to describe the system using equations and represent those equations, using a programming or simulation language, into a form suitable for simulation.

2.2.2 Graphical Block-Diagram Modeling

Graphical representations of systems were commonly used to describe models in analog simulations. However, due to the lack of graphical capabilities of the early digital computers, simulators were reverted to textual descriptions of the models. The development of models using graphical descriptions were recovered, and extended, with the availability of graphical displays for personal workstations and computers, as well as the development of graphical user interfaces (GUI's).

The block-diagram modeling methodology allows to describe models in a hierarchical and modular way. Each model is composed as a combination of blocks, input and output ports, and interconnections between ports and blocks. Coupled models can be composed as a combination of models, with interconnected ports. Port connections are performed drawing lines between ports. Model libraries contain commonly used blocks to facilitate the development of new models, by simply drag and drop the required blocks.

Several environments support the graphical block-diagram modeling methodology. The MATRIX$_X$ environment included the SystemBuild tool [Shah et al., 1985]. Matlab, currently considered as a de-facto standard for computing and algorithm programming in engineering, included the SIMULINK environment [Grace, 1991]. VisSim is a PC-based environment developed in 1990 [Darnell and Kolk, 1990]. ACSL also included Graphics Modeller in 1993.

The block-diagram modeling paradigm inherits many of its characteristics from the analog simulation. Blocks represent the basic operations used in analog simulation to describe equations (e.g., addition, multiplication, integration, etc.). The use of a graphical tool in a digital computer facilitates the description of the model, providing advantages regarding the modular and hierarchical description of models.

The requirement of defining explicit state models (ODE) in analog simulation is inherited by the block-diagram modeling, and supposes a limitation. The construction of models still requires the modeler to manually perform mathematical manipulations to the equations. A paradigm shift is required to solve this limitation.

2.2.3 The Physical Modeling Paradigm

Models in the physical modeling paradigm are defined in a modular way. A system is decomposed into subsystems, and each subsystem is described using an interface, balances of mass, energy and momentum, and material equations. The interface is used to describe the relations between subsystems. A model is considered as a constraint between model variables [Åström et al., 1998].

The physical modeling paradigm supposes an evolution from the block-diagram modeling. The description of the model is closer to the definition of a real physical system, and is naturally performed using differential, algebraic and discrete equations [Cellier et al., 1996]. The equations of the model may also change due to the occurrence of discrete events, leading to hybrid models. The mathematical manipulations of the equations, required to execute the simulation in a digital computer, are automatically performed by the modeling environments using symbolic formula manipulation algorithms (like the Tarjan [1972] and Pantelides [1988] algorithms).

2.2.4 The Object-Oriented Modeling Methodology

Equation-based object-oriented (EOO) modeling methodology facilitates the physical modeling paradigm. One of the first contributions regarding the EOO modeling was performed by Elmqvist [1978]. The object-oriented programming techniques were applied to modeling, in order to facilitate the description of systems and reducing the time and cost of model development [Cellier, 1996]. The idea is to define basic models of components and use them to construct bigger and more complex models. Similarly to how an engineer designs a new system, us-

ing already existing components. The EOO methodology has the characteristics described below [Cellier, 1996].

- *Encapsulation of knowledge.* The model has to contain all the information that represents the object, and a well-defined interface to communicate with its environment.
- *Topological interconnection capability.* It should be allowed to connect models following the topological structure of the real system.
- *Hierarchical modeling.* Models constructed with basic equations or as a combination of other models can not be distinguished when observed from the outside, and can be arranged in a hierarchical fashion using the model interface. Models can have the same interface and the same behavior, while being described using different methods (i.e., equations or interconnected components).
- *Object instantiation.* Models can be described as generic classes. Objects can be instantiated from those classes by a mechanism of model invocation.
- *Class inheritance.* Common information shared by several models should be included into general classes. That information should be used by other models using an inheritance mechanism.
- *Generalized networking capability.* Model interconnections can be made directly or using nodes. The behavior of these nodes is defined by the across or through variables that compose it. The values of across variables in a node are equaled. The values of through variables are summed up and the sum equaled to zero.

Each model is composed of *internal description* and *interface*. The internal description can be defined behaviorly (i.e., using equations or algorithms), or describing its internal structure (i.e., as a combination of interconnected components). The interface describes the interaction of the component with other components and its environment.

Equations are acausal, maintaining their mathematical meaning (opposite to the *assignment* meaning usually given in programming languages to the '=' sign). Connections between models are non-directional.

2.2.5 Object-Oriented Modeling Environments

Multiple modeling languages have been developed to support the EOO methodology. The modeling environments supporting EOO languages automatically perform the symbolic manipulations required to translate the acausal, object-oriented description of the model into efficient executable code [Cellier and Kofman, 2006].

EOO modeling languages facilitate the description of the continuous-time part of hybrid models using differential and algebraic equations. In addition, these languages provide constructs to describe discontinuities in the continuous-time behavior, equations with variable structure, and time and state events. The modeling environments needs, for simulating hybrid models (i.e., a set of synchronous differential, algebraic and discrete equations), are the following [Urquia, 2000]:

1. A simulation algorithm appropriate for hybrid systems (for instance, the Omola simulation algorithm is described in [Andersson, 1994]).

2. An adequate treatment of the discrete events [Elmqvist et al., 1993]: the detection, the accurate determination of the trigger time [Elmqvist et al., 1993; Cellier, 1979; Cellier et al., 1993; Elmqvist et al., 1994] and the re-start problem solution.

3. Algorithms to carry out the symbolic manipulation of the linear systems of simultaneous equations and to tear the nonlinear ones [Elmqvist and Otter, 1994].

Many languages and tools support the EOO methodology. Dymola appeared in the 1990's, as a result of the development of the ideas proposed by Elmqvist [1978], Cellier [1979] and the DSBlock interface developed by Otter and Elmqvist [1995]. Other languages that support this methodology are gPROMS [Barton

and Pantelides, 1994], EcosimPro [EA International, 2010], χ (Chi) [van Beek and Rooda, 2000], Verilog-AMS [Frey and O'Riordan, 2000] and VHDL-AMS [IEEE, 1997]. Modelica constitutes an international effort to standardize the description of models following the object-oriented methodology, joining ideas from existing languages and tools [Mattsson et al., 1998].

2.3 The Modelica Language

Modelica is a free modeling language, distributed under its own license, mainly designed to describe mathematical models of physical systems [Modelica Association, 2009]. Modelica is developed and maintained by the Modelica Association, which is an international association composed by multiple organizations and individual members [Modelica, 2010]. The language includes several characteristics from previous languages, like ALLAN [Jeandel et al., 1997], Dymola [Elmqvist, 1978], NMF [Sahlin et al., 1996], ObjectMath [Fritzson et al., 1995], Omola [Andersson, 1989], SIDOPS+ [Breuneuse and Broenink, 1997] and Smile [Kloas et al., 1995]. Multiple free and commercial tools support the Modelica language, such as CATIA [Dassault Systemes, 2009], Dymola [Dynasim AB, 2006], LMS Imagine.Lab AMESim [LMS International, 2009], MapleSim [Maplesoft, 2009], MathModelica [MathCore Engineering AB, 2009], SimulationX [ITI GmbH, 2009], OpenModelica [Fritzson et al., 2002] and Scicos [Campbell et al., 2006].

2.3.1 Characteristics of Modelica

Some of the main functionalities to describe models offered by Modelica are [Modelica Association, 2009]:

- *Description of models using acausal equations.* The causality is automatically assigned by the modeling environment by performing symbolic manipulations to the equations.

- *Combined use of equations and algorithms to define models.* The algorithms are executed imperatively, facilitating the description of behaviors with a fixed causality. The single assignment rule is not applied inside an algorithm section, but it is applied between different sections.

- *Reusable algorithm descriptions, as functions.* That allow to describe algorithmic operations as functions with parameters, and reuse them by simply calling the defined function using the appropriate parameters.

- Models can be either directly coded in a single Modelica class, composed by interconnected instantiations (objects) of different classes, or a combination of both methods. The modeler can select to describe the behavior of a model, using equations, or to describe its internal structure, including and interconnecting previously developed components.

- *Information encapsulation*, that allows to hide information contained in a class that may not be relevant for outer classes or users. This functionality helps to structure the information contained in a model, and to avoid erroneous assignments or misuse of the internal components of a class.

- *Multiple class inheritance and definition of partial classes*, which include general properties of a class but can not be instantiated (i.e., all Modelica models inherit the characteristics of a superclass named *class*). Classes may inherit information or characteristics from one or multiple classes, using the *extends* clause. This facilitates the description of common characteristics that are shared by several models or classes.

- *Class parameterization of the defined objects.* Using the *replaceable* and *redeclare* constructs it is possible to modify the class of an object, even when already defined in a model. It simplifies the experimentation with the model. The modeler is allowed to modify the class of a defined object instead of having to re-describe the model and its components (e.g., in a model of a car, different types of motors – electrical, combustion, etc. – can

be tested without having to describe multiple models of the same car each one with a different motor, only the class of the motor is changed).

– Solving the initialization problem for a model is sometimes problematic. Modelica offers capacities for the initialization of the model, like the *start* and *fixed* attributes, *initial equation* and *initial algorithm* sections, and the *initial()* condition. These capacities allow to define different descriptions within the same model. One description is only used during the initialization and the other is used to describe the dynamic behavior.

– Provides language constructs to describe the trigger conditions of time and state events, and also the actions associated to the events [Elmqvist et al., 1993; Mattsson et al., 1999; Otter et al., 1999]. These actions can be: (1) update the value of discrete-time variables; (2) reinitialize continuous-time state variables, using *when* clauses; and (2) change the mathematical description of equations and assignments, using the *if* statement.

– *Textually based treatment of event conditions* (using the *noEvent* construct). Real elementary relations within expressions are taken literally instead of generating crossing functions (i.e., no state or time event is triggered). This characteristic can be used to avoid errors during the treatment of expressions inside if statements, where the value of one of the branches is invalid (e.g., the square root of a negative value).

– *Model annotations*, that may contain additional information of the model (i.e., the graphical representation, icon representation, environment-dependent information, version, documentation, etc.). This functionality helps the modeler to describe the behavior, characteristics and use of the developed model.

– *Components and connections vectorization.* That facilitates the description of multiple equal components or connectors, by declaring them as arrays.

– *External function interface with C and Fortran.* Which facilitates the inclusion of C and Fortran code into Modelica, extending the functionalities of Modelica with those of these general programming languages.

– *Supports automatic and user-defined selection of state variables* (using the *stateSelect* attribute). Dymola automatically performs the selection of the state variables of the model. However, in order to obtain a better selection the modeler can use this functionality to indicate the desired state variables. The dynamic selection of variables during the simulation is also supported.

2.3.2 Modelica Classes

Everything in Modelica is described using a class. There are different specialized classes to facilitate the description of models. These classes present restrictions in the amount and type of components they may contain, from the general Modelica *class*. The characteristics and restrictions of these specialized classes are summarized in Table 2.1.

A Modelica model is one of the mentioned specialized classes, but it has no restrictions from the general class. A model in Modelica may include a variation of the following components:

– *parameters/constants*: that represent the variables whose value remain constant during the simulation.

– *variables*: that represent the variables whose value may vary during the simulation.

– *algorithm sections*: to describe algorithmic behavior (i.e., imperatively described and sequentially executed).

– *equation sections*: including descriptions of the relations between the variables of the model (algebraic and differential variables).

– *initial algorithms/equations*: like the previous sections, but only used to initialize the state of the model.

A model in Modelica has to fulfill the single-assignment rule. This means that the number of unknown variables and equations in the model has to be equal, and that the number of equations in each branch of a conditional equation must also be equal. Otherwise, the model is incorrect.

Modelica provides the *connector* class, to describe the model interface, and the *connect* sentence, to describe the interactions (or connections) between models. Variables in the connectors can be either *across* or *through*. Variables in Modelica connectors are described by default as across, and the *flow* modifier is provided to describe through variables. The values of across variables between two connected connectors are equaled, and the values of through variables are summed up and the sum is equaled to zero. For instance, the voltage across a node in an electric circuit represents an across variable, while the current represents a through variable.

Table 2.1: Modelica 3.1 specialized classes and their characteristics [Modelica Association, 2009].

Record	used to define structured and complex data types. It can only include public components (equation, algorithm, initial and protected sections are not allowed). They have implicitly available construction functions.
Type	used to define new data types, based on the basic types (enumeration, array, Real, Boolean and Integer).
Model	used to describe general models (identical to the general class, with no restrictions).
Block	used to describe block-diagram models. Each connector must have well defined input and output ports (using the *input* and *output* modifiers).
Function	used to describe algorithmic functions, with parameters and output values.
Connector	used to define the interface ports of the models. They or any of its components can not contain equations.
Package	used to hierarchically structure the developed models, and create model libraries. They may only contain model declarations and constants.
Operator	used to define overloaded operations over data types described as records. May only contain function declarations.
Operator function	easier way to describe an operator with only one function.

2.3.3 Modelica Libraries

The possibility of reusing components from different libraries strengthen the Modelica modeling capabilities. Modelica supports multiple modeling formalisms by means of libraries of components [Modelica Libraries, 2010]. For instance, StateGraphs [Otter et al., 2005], Petri Nets [Mosterman et al., 1998], DEVS [Beltrame, 2006], System Dynamics [Cellier, 2008] and Bond Graphs [Cellier and Nebot, 2005].

Also, the description of models from multiple application domains is facilitated due to the currently available Modelica libraries. Libraries for domains such as thermodynamics [Cellier and Greifeneder, 2008; Casella and Leva, 2003], electrical [Cellier et al., 2007], mechanical [Otter et al., 2003], fuel cells [Rubio et al., 2005], vehicle dynamics [Andreasson, 2003] and virtual laboratories [Martin-Villalba et al., 2008] are available.

The main Modelica library is the Modelica Standard Library (MSL), which is developed and supported by the Modelica Association [MSL, 2010]. An incomplete list of some freely available Modelica libraries is shown in Table 2.2.

Table 2.2: Some available free Modelica libraries [Modelica Libraries, 2010].

Library Name	Short Description
WasteWater	Free library for modelling and simulation of waste water treatment plants [Reichl, 2003].
ObjectStab	Free library for power systems voltage and transient simulation [Larsson, 2000].
ATplus	Building Simulation and Building Control (fuzzy control library included) [Felgner et al., 2002].
MotorcycleDynamics	This Modelica library has been developed for the dynamic simulation of a motorcycle, and tailored to test and validation of active control systems for motorcycle dynamics [Donida et al., 2006].
NeuralNetwork	It provides the neural network mathematical model [Codecà and Casella, 2006].
VehicleDynamics	Free library to model the dynamics of vehicle chassis [Andreasson, 2003].
SPICELib	Free library with some of the modeling and analysis capabilities of the electric circuit simulator PSPICE [Urquia et al., 2005].
SystemDynamics	Free library for modeling according to the principles of system dynamics of J. Forrester [Cellier, 2008].
TechThermo	Free library for technical thermodynamics [Steinmann and Zunft, 2002].
FuzzyControl	Free library for fuzzy control.
ThermoPower	Free library to model thermal power plants (based on Modelica.Media) [Casella and Leva, 2003]
BondLib	Free library to model physical systems with bond graphs [Cellier and Nebot, 2005].
ExtendedPetriNets	Free library to model Petri Nets and state transition diagrams (extended version) [Fabricius and Badreddin, 2002*a*].
FuelCellLib	Free library to model fuel cells [Rubio et al., 2005].
QSSFluidFlow	Free library for quasi steady-state fluid pipe flow [Fabricius and Badreddin, 2002*b*].
SPOT	Free library providing components to model power systems both in transient and steady-state mode.
ModelicaDEVS	A free library for discrete-event modeling using the DEVS formalism [Beltrame and Cellier, 2006].
MultiBondLib	Free library to model physical systems with multi-bond graphs [Zimmer, 2006].
ExternalMedia Library	Including external fluid property computation code in Modelica [Casella and Richter, 2008].
Verif	A free library for verifying the ModelicaSpice library (part of BondLib).
Buildings	Free library for modeling building energy and control systems, based on Modelica.Fluid [Wetter, 2009].
VirtualLabBuilder	VirtualLabBuilder Modelica library facilitates the implementation of virtual-labs using only Modelica [Martin-Villalba et al., 2008].

2.3.4 Simulation of Modelica Models

Models in Modelica are described following the EOO modeling methodology. They are later translated by the modeling environment into a hybrid DAE form. The formal description of a hybrid DAE is [Modelica Association, 2009]:

$$c := f_c(relation(v))$$

$$m := f_m(v, c)$$

$$0 = f_x(v, c)$$

with $v := [\dot{x}, x, y, t, m, pre(m), p]$, and where:

- p are the variables without time dependency (i.e., parameters or constants).
- t is the independent variable (time).
- $x(t)$ is the set of variables that appear differentiated.
- $m(t_e)$ are the variables that are unknown and only change their values at event instants t_e. $pre(m)$ are the values of these variables immediately before the event.
- $y(t)$ is the set of algebraic variables.
- $c(t_e)$ are the conditions of all if-expressions, included when-expressions after conversion.
- $relation(v)$ are the relations containing variables v_i (e.g., $v_1 < v_2$).

This description defines a DAE which may include discontinuities, variable structure and/or discrete-events.

Equations in Modelica follow the synchronous data flow principle, meaning that at each time instant the active equations express relations between variables that have to be fulfilled concurrently [Otter et al., 1999]. The order in which the equations are evaluated is automatically determined by data flow analysis of the system of equations, leading to unique computations of the unknown variables.

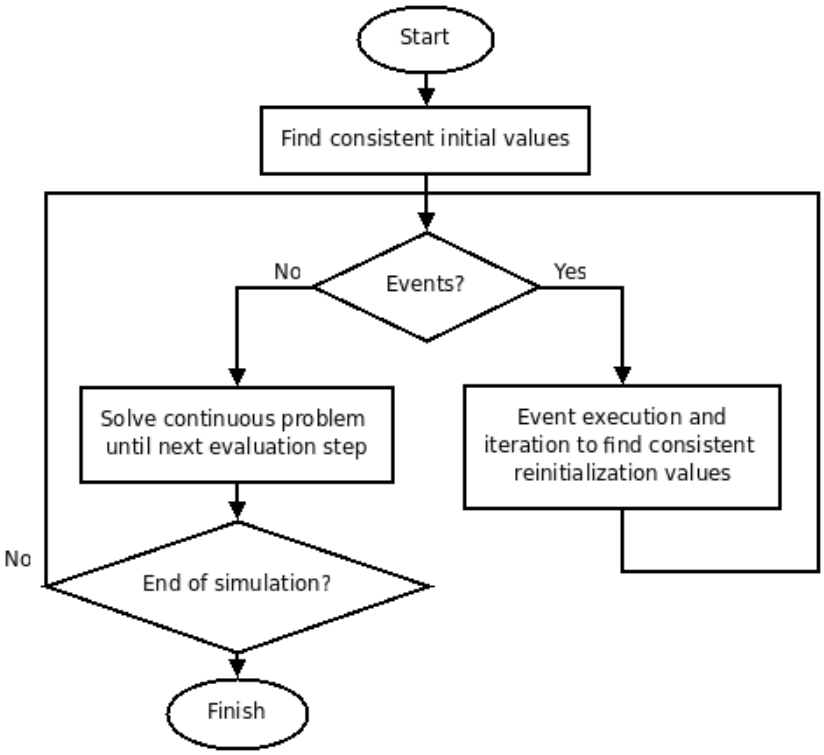


Figure 2.1: Simulation algorithm of hybrid models.

The interpretation of the language specification regarding the treatment of events has been questioned by Nikoukhah [2007], considering that different interpretations may lead into different model and compiler constructions. A proposal for introducing synchronous and asynchronous events in Modelica was performed by Nikoukhah and Furic [2008].

The simulation is performed as follows Modelica Association [2009]: (1) the continuous-time part is solved using a numerical integration algorithm; (2) if any of the event conditions is met during integration, the integration algorithm is halted and the event instant is determined; (3) at the event instant the set of algebraic and discrete equations are solved; and (4) once the event has been treated, the event conditions are checked again. If a new event is triggered, it is immediately executed (i.e., event iteration). Otherwise, the integration is restarted. The diagram shown in Fig. 2.1 summarizes this procedure.

2.4 Discrete-Event System Modeling

Discrete-event models are defined by the occurrence of events [Banks et al., 1996; Cassandras and Lafortune, 1999]. An event can be defined as a phenomenon that occurs instantaneously in a given point in time and affects the represented system (e.g., the arrival of a new customer, the impact of a ball with the ground, the end of a pre-programmed process, etc.). The state of a discrete-event model can only change a finite number of times in an finite time interval, depending on the occurrence of events. The values of the state variables remain constant between two consecutive events.

Events can be of two types: state events, when a certain condition that involves any state variable of the model is met (e.g., the level of a tank reaches the maximum value), and time events, that are scheduled to occur in a certain point in time (e.g., a process that will finish in three minutes). Also, events can occur simultaneously, and perform several changes in the state variables at the same time.

The development of discrete-event modeling and simulation techniques has been broad, in comparison to the continuous-time modeling techniques that are mainly based on the DAE formalism. Multiple formalisms and simulation languages have been developed, depending on the characteristics of the systems to model and the purposes of the simulation studies.

Modeling formalisms help to describe and study the behavior and characteristics of systems, by means of a mathematical description. The application of discrete-event modeling formalisms to the construction of the discrete-event part of hybrid models facilitates the model development, maintenance and reuse [Robinson et al., 2004]. It also helps to ensure the correctness and validity of the developed model.

Some of the most common formalisms for discrete-event system modeling are: Finite State Automata/Machines, StateCharts, Process Algebra, π-Calculus, Petri Nets, Generalized Semi-Markov Processes and DEVS. Finite State Automata, Petri Nets, StateCharts and DEVS, can be remarked due to their high

acceptance in engineering [Hrúz and Zhou, 2007]. Page [1994] discussed the characteristics of some of these modeling formalisms based on a set of requirements. None of the evaluated formalisms fulfilled the whole set of requirements, showing deficiencies in one or more aspects. Each formalism has its own characteristics and functionalities.

Many extensions of the basic formalisms have been developed in order to add new functionalities to them. For example, Time Petri Nets is an extension of the Petri Nets formalism to allow the modeling of dynamic models (i.e., represent the evolution of time), Hybrid Petri Nets [David and Alla, 2001] and Hybrid Automata [Lynch et al., 2003] are extensions for hybrid system modeling using Petri Nets or Automata, respectively.

DEVS (Discrete EVent Systems specification) was developed by Zeigler [1976] as a general formalism for representing systems. The main characteristic of DEVS is the hierarchical and modular description of models. Extensions to the DEVS formalism include Parallel DEVS [Chow, 1996], DEV&DESS for combined continuous-time and discrete-event systems [Zeigler et al., 2000], RT-DEVS for real-time discrete-event systems [Hong et al., 1997], Cell-DEVS for cellular automata [Wainer and Giambiasi, 2001], Fuzzy-DEVS [Kwon et al., 1996] and Dynamic Structuring DEVS [Barros, 1995].

DEVS can be considered as a universal formalism [Zeigler et al., 2000] for DEVS (Discrete Event System Specification), DTSS (Discrete Time System Specification) and DESS (Differential Equation System Specification), due to possible model transformations from other formalisms to DEVS [Vangheluwe, 2000]. The DEVS formalism has been considered as the "differential equations for discrete-event systems" [Zeigler, 1989]. Differential equations can be simulated using DEVS [Kofman et al., 2001].

2.5 Discrete-Event System Simulation

Formal models are independent from any programming language, and thus have to be translated or implemented in order to be simulated. Simulations can be

programmed using general purpose programming languages or specific simulation languages. The development of discrete-event simulation languages is based on the perspective used to represent the world, in order to simulate it. These perspectives where introduced by Lackner [1962] and extended by Kiviat [1969] into the three categories, or "world-views", commonly used in the literature (each discrete-event simulation language or tool focuses in one of these world-views):

- *Event-scheduling* (also called event-driven) focuses on events, which cause changes in the state of the system. The model of the system consists on a description of the causes and the effects of the events in the system. The order in which the events are treated represents the evolution of the state of the system [Fishman, 2001]. The simulation is performed maintaining a list of events ordered by its time of occurrence. The simulation clock is advanced to the next event, and its treatment is executed.

- *Activity-scanning*, that focuses on the activities performed in the system and the conditions that control the begin and end of such activities. The resources are considered as prerequisites for those activities. An example of application of this approach using the Stroboscope environment is presented in Martinez and Ioannou [1995]. A comparison of the activity scanning approach and the process interaction approach is given in Ioannou and Martinez [1999].

- *Process-interaction* (also called process-oriented) focuses on how entities flow through the system. Some processes are applied to the entities, after capturing the required resources. This provides a more natural representation of the processes and components in a system, and their interactions. This is the approach of simulation languages such as GPSS/H, SIMAN, SIMSCRIPT II.5 and SLAM.

The factors involved in the evolution of discrete-event simulation are discussed in Nance and Sargent [2002]. External factors, such as the revolution of computer hardware, the advances of computer software, computer graphics, human-computer interactions and computer networks are presented. Also inter-

nal factors, such as the development of formal modeling methodologies, the study of pseudo-random number generators and the improvement on the verification and validation of the developed models among others, are discussed.

Nance [1993] provided a survey of the evolution of the discrete-event simulation languages structured in the following periods:

- *Search (1955-60)*: with focus on the identification of concepts for model representation and the needs for simulation modeling. The first developed simulation language is credited to Tocher and Owen [1960], and called GSP (General Simulation Program).

- *Advent (1961-65)*: where the foundations of the current simulation programming languages appeared. Some of the main languages that appeared during this period are GPSS, SIMULA I, SIMSCRIPT, CSL, GASP, OPS-3 and DYNAMO.

- *Formative (1966-70)*: where the concepts of simulation where reviewed and clarified, taking advance of the new computing possibilities due to hardware improvements. Some languages suffered several revisions, like GPSS (with GPSS II and III), SIMULA (with SIMULA 67) , SIMSCRIPT (with SIMSCRIPT II), GASP (with GASP II) and OPS-3 (with OPS-4), while other languages appeared, like ECSL.

- *Expansion (1971-78)*: where some languages suffered mayor expansions, like GPSS (with GPSS/NORDEN, NGPSS, GPSS V6000, GPDS, GPSS 1100 and GPSS/H), SIMSCRIPT (with SIMSCRIPT II.5, C-SIMSCRIPT, ECSS and CSP II) and GASP (with GASP IV and GASP PL/I).

- *Consolidation and regeneration (1979-86)*: where the main simulation languages consolidated their positions (like GPSS and SIMSCRIPT II.5) and were made available for multiple platforms, such as personal computers and microprocessors. Also, two new languages appeared as descendents from GASP: SLAM II and SIMAN.

These periods are also discussed in Robinson [2005] and summarized in four instead of five, mentioning the impulse of visual interactive simulations in the evolution of languages.

Robinson [2005] also discusses the evolution of discrete-event simulation from the 1990s to the present. During this period, and due to the increasing performance of personal computers and better human-computer interfaces, the use of visual interactive simulations, simulation optimization, virtual reality and software integration played the main role in the evolution of simulation technologies. Multiple Visual Interactive Modeling Systems (VIMS as denominated by Pidd [2004]) or Simulation Packages (as denominated by Law and Kelton [2000]) were developed during this period, like Arena, AutoMod, ProModel, WITNESS, Simul8 and Extend among many others. The characteristics and functionalities of the Arena simulation environment are discussed later in this chapter.

2.6 The Parallel DEVS Formalism

The Parallel DEVS (P-DEVS) formalism was introduced by Chow [1996] as an extension to the original Classic DEVS formalism. P-DEVS removes the sequential management of events, allowing simultaneous occurrences of events. It also facilitates the user the control over confluent events, or simultaneous internal and external events, by defining the confluent transition function (δcon).

Several simulation environments support the Parallel DEVS formalism, including DEVS-C++ [Zeigler et al., 1996], adevs [Nutaro, 1999], DEVSJAVA [Zeigler and Sarjoughian, 2003] and CD++ [Liu and Wainer, 2007]. These environments are mainly based on general programming languages, like C++ or JAVA. They provide functions and data structures designed to facilitate the description of P-DEVS models, but general computer programming skills and knowledge are required in order to develop models using them. Each environment uses a particular format to describe models, so model exchange and reutilization between environments is difficult to perform. The use of a common language or format

to describe P-DEVS models could facilitate their development, maintenance and reuse between different tools.

Models in P-DEVS, as well as in the Classic DEVS formalism, can be described behaviorally (named *atomic*) or structurally (named *coupled*). This section contains a description of the P-DEVS specification and behavior of atomic and coupled models. Also, multiple existing DEVS-based methodologies and formalisms to describe hybrid systems are discussed.

2.6.1 Atomic P-DEVS Models

According to the P-DEVS formalism, an atomic model is the smallest component that can be used to describe the behavior of a system. It is defined by a tuple of eight elements [Chow, 1996; Zeigler et al., 2000]:

$$M = (X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$$

where:

$X_M = \{(p, v) \mid p \in IPorts, v \in X_p\}$	Set of *input ports and values.*
S	Set of *sequential states.*
$Y_M = \{(p, v) \mid p \in OPorts, v \in Y_p\}$	Set of *output ports and values.*
$\delta_{int} : S \longrightarrow S$	*Internal transition* function.
$\delta_{ext} : Q \times X_M^b \longrightarrow S$	*External transition* function, where $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the *total state* set and e is the *time elapsed* since the last transition.
$\delta_{con} : Q \times X_M^b \longrightarrow S$	*Confluent transition* function.
$\lambda : S \longrightarrow Y_M^b$	*Output* function.
$ta : S \longrightarrow \Re_{0,\infty}^+$	*Time advance* function.

An atomic model remains in the state $s \in S$, for a time $t_s = ta(s)$. After t_s is elapsed, an *internal event* is triggered and the state is changed to $s_{new} = \delta_{int}(s)$.

Before that, an output can be generated using the output function and the state previous to the event ($output = \lambda(s)$).

After the execution of the transition, a new internal event is scheduled at time $t_{snew} = ta(s_{new}) + time$. Then, $t_{last} = time$, where $time$ is the current simulation time.

Multiple inputs can be received simultaneously through one or several ports.

- If any input is received at time t_{ext} and $t_{ext} < t_s$ (so the inputs are received before the next internal event), an *external event* is triggered. During the external event, the state is changed to $s_{new2} = \delta_{ext}(s, e, bag)$, where s is the current state, e is the elapsed time since the last transition ($t_{ext} - t_{last}$) and $bag \subseteq X_M$ is the set of received input messages.

- If the external input is received at time t_{ext} and $t_{ext} = t_s$, the external and the internal events are triggered simultaneously. This situation triggers a *confluent event* (that substitutes the external and internal events), and the state is changed to $s_{new3} = \delta_{con}(s, e, bag)$, being s the current state, e the elapsed time, and $bag \subseteq X_M$ the set of received inputs (similarly to the δ_{ext} function). Also, similarly to the internal events, an output can be generated as $output = \lambda(s)$ before executing the confluent transition function.

New internal events are also scheduled after the external and confluent transitions using $ta()$. Note that the time advance function can return a zero value, generating an immediate internal event.

2.6.2 Coupled P-DEVS Models

The P-DEVS formalism supports the hierarchical and modular description of the model. Every model has an interface to communicate with other models.

A coupled P-DEVS model is a model composed of several interconnected atomic or coupled models, that communicate externally using the input and output ports of the coupled model interface. It is described by the following tuple [Zeigler et al., 2000]:

$$M = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC)$$

where:

$X = \{(p, v) \| p \in IPorts, v \in X_p\}$	Set of *input ports and values*.
$Y = \{(p, v) \| p \in OPorts, v \in Y_p\}$	Set of *output ports and values*.
D	Set of the *component names*.
M_d	*DEVS model*, for each $d \in D$.
EIC	*External Input Coupling*: connections between the inputs of the coupled model and its internal components.
EOC	*External Output Coupling*: connections between the internal components and the outputs of the coupled model.
IC	*Internal Coupling*: connections between the internal components.

The connection of P-DEVS models implies the establishment of a information transmission mechanism between the connected models. P-DEVS models follow a message passing communication mechanism. A model generates messages as outputs, using its output function, which are received by other models as external inputs. Messages can be received simultaneously through one or multiple ports. Connections between models can be in the form of 1-to-1, 1-to-many and many-to-1. Each message can transport an arbitrarily complex amount of information, depending on the particular application.

2.6.3 DEVS-based Approaches for Hybrid System Modeling

The DEVS formalism was initially designed to describe discrete-event systems. However, multiple methods have been developed to describe hybrid systems using DEVS-based techniques:

– The generalization of the Classic DEVS formalism into the GDEVS formalism [Giambiasi and Carmona, 2006], that allows to describe models whose state is described using polinomies (instead of constants). This facilitates the description of continuous-time behavior using discrete-event mechanisms.

– The state quantization is a method to obtain a discrete-event approximation of a continuous-time system [Kofman et al., 2001]. The simulation is performed at discrete steps based on the variation of the state, instead of the time. It represents a reduction in computational cost and the possibility for distributed implementation. Zeigler and Lee [1998] initially proposed a method to describe continuous-time systems using state quantization. It was later extended into the concept of Quantized State Systems (QSS), including an hysteresis in order to guarantee legitimate DEVS models [Kofman and Junco, 2001; Kofman, 2004].

– Mixed discrete-event and continuous-time description of the system. Multiple modeling environments and languages include functionalities to describe continuous-time systems and simulate them using numerical integration methods. These tools allow the combination of discrete-event and continuous-time dynamics. Some of these environments are the JAMES II [Himmelspach and Uhrmacher, 2009], D-SOL [Jacobs et al., 2002], the Virtual Laboratory Environment [Quesnel et al., 2008] and CD++ [Wainer, 2002].

  The DEV&DESS formalism describes a subclass of dynamic systems that includes the subclasses specified by DEVS and DESS (Differential Equation System Specification) [Prähofer, 1991; Zeigler et al., 2000]. The system is described using a combination of discrete-event and continuous inputs, states and outputs, that interact to represent the desired system behavior. As described in Zeigler [2006], the DEV&DESS formalism can be embedded into DEVS.

- The Heterogeneous Flow System Specification (HFSS) is an extension of DEVS that builds in an input sampling mechanism as well as variable structure capability [Barros, 2002*b*,*a*, 2003]. However, there has been no discussion of the conditions under which models in the formalism are well-defined or whether closure under coupling holds for the formalism [Zeigler, 2006].

- Another approach has been to transform continuous-time models, described using Modelica, into DEVS models in order to simulate them [D'Abreu and Wainer, 2005]. This method uses Bond Graphs as an intermediate formalism to perform the transformation.

However, none of these methods support the EOO methodology for continuous-time system modeling. The advantages of the continuous-time object-oriented modeling, described above in this chapter, should be considered for describing hybrid systems.

The Modelica language could be a vehicle for combining the use of the DEVS formalism with other modeling formalisms and techniques. The feasibility of describing atomic DEVS models in Modelica was demonstrated in [Fritzson, 2003]. Also, a Modelica library, called ModelicaDEVS [Beltrame and Cellier, 2006; Beltrame, 2006], was developed for modeling continuous-time systems using the DEVS formalism and the QSS integration algorithms [Kofman, 2004; Cellier and Kofman, 2006].

2.7 The Arena Simulation Environment

Arena is a commercial software environment marketed by Rockwell Automation Inc. designed for discrete-event system modeling and simulation [Kelton et al., 2007]. Arena follows the process-oriented approach to describe systems. It is one of the most widely used environments for describing discrete-event and logistic systems.

Since the objective of this dissertation is the description of P-DEVS and process-oriented models in Modelica, Arena has been selected as an example in which to base the performed work. This will help to validate the developed models by comparison with equivalent models constructed using Arena. Comparisons between hybrid models will also be performed, since Arena provides some functionalities for describing hybrid systems.

Arena models are composed of *flowchart diagram* and *static data*. The flowchart diagram describes the structure of the system, the connection between components and the flow of entities through them. The static data represents the particular characteristics of the elements in the system (i.e., the structure of the queues, the capacity of the resources, etc.). Thus, components in Arena are divided into flowchart modules and data modules. Models are constructed using a graphical user interface, including the required modules into a blank "draft" model (i.e., using drag and drop), and configuring the connections and parameters of the included modules.

The behavior of flowchart diagram modules is similar to the behavior of P-DEVS models. The transmission of messages between P-DEVS models could represent the flow of entities between modules. Thus, the P-DEVS formalism will be used to introduce process-oriented models in Modelica.

2.7.1 Arena Panels

Components in Arena are arranged into panels, similarly to the libraries in Modelica. Each panel includes elements to describe different types of processes at multiple levels of abstraction. The main panel is named *BasicProcess*, and contains basic model components. However, these components can be used to represent many of the processes and behaviors usually found in systems. Some of the other available panels are the AdvanceProcess, AdvanceTransfer, AgentUtil, FlowProcess and Packaging.

The flowchart modules of the BasicProcess panel are:

– *Create*, that represents the starting point for entities in the system.

- *Dispose*, opposed to the create module, represents the end point for the entities.
- *Process*, represents any process to be performed to an entity during a defined period of time. The use of resources to process the entity is not mandatory, and can be configured using the parameters of the module.
- *Decide*, represents a division in the flow of entities. The division can be probabilistic or following a certain condition. Multiple division rules can be applied.
- *Batch*, is used to group entities temporarily or permanently, depending on certain conditions.
- *Separate*, is used to separate previously batched entities or to copy entities into multiple replications.
- *Assign*, is used to assign new values to the global variables of the system or the user-defined attributes of the entities.
- *Record*, is used to collect statistics during the simulation. These statistical indicators are reported at the end of the simulation.

The data modules of the BasicProcess panel are:

- *Entity*, represents a type of entities that are created in the system. Different parameters can be assigned to different types of entities (e.g., customers, pieces, cars, etc.).
- *Queue*, describes the characteristics of a queue associated to a Process.
- *Resource*, describes the characteristics of the resources associated to a Process.
- *Variable*, is used to define global variables in the system.
- *Schedule*, is used to describe the patterns of time associated with the availability of resources, the creation of entities, or processing delays.

– *Set*, is used to aggregate multiple elements of the system into sets (i.e., sets of resources, counters, tallies, etc.)

At the lowest level of abstraction, Arena includes two panels: the Blocks and the Elements. These two panels correspond to the components of the SIMAN modeling language [Pegden et al., 1995]. Each Arena component is internally described using a combination of these SIMAN components.

2.7.2 SIMAN Language

Models described using SIMAN are also composed of flowchart diagram and static data. The flowchart diagram is described using Blocks, and the static data is described using Elements. The blocks represent simple actions performed during the flow of entities through the system. For instance, seizing or releasing a resource, being delayed in a process, update an statistical indicator or wait in a queue. The elements, like the data modules in Arena, represent the particular characteristics of some components in the system (i.e., resources, queues, variables, etc.). Some of the elements are also used to describe the experiment to be performed with the system, like the duration, number of runs, the initialization of the random number generator, etc.

2.7.3 Random Number Generation in Arena

Process-oriented models are sometimes stochastic [Law, 2007]. The inter-arrival times for entity creation and processing delays are examples of random variables commonly used in models.

Arena includes a Combined Multiple Recursive Generator (CMRG) to generate random uniform numbers and random variates [L'Ecuyer, 1999]. This random number generator (RNG) gives the possibility of creating multiple random streams, and sub-streams, that can be considered as independent RNGs [L'Ecuyer et al., 2002]. Each random variable can be assigned with a different stream, thus facilitating the execution of independent replications or the application of variance reduction techniques [Law and Kelton, 2000].

A detailed description of this generator can be found in L'Ecuyer [2001]. The backbone generator, whose period is later divided into streams and sub-streams, is described with two components of order three. At the step n, the state of the generator is described by the pair of vectors $s_{1,n} = (x_{1,n}, x_{1,n+1}, x_{1,n+2})$ and $s_{2,n} = (x_{2,n}, x_{2,n+1}, x_{2,n+2})$ which evolve according to:

$$x_{1,n} = (1403580 \times x_{1,n-2} - 810728 \times x_{1,n-3}) \; mod \; m_1$$

$$x_{2,n} = (527612 \times x_{2,n-1} - 1370589 \times x_{2,n-3}) \; mod \; m_2$$

where $m_1 = 2^{32} - 209 = 4294967087$ and $m_2 = 2^{32} - 22853 = 4294944443$, and its output u_n is defined by:

$$z_n = (x_{1,n} - x_{2,n}) \; mod \; 4294967087$$

$$u_n = \begin{cases} z_n / 4294967088 & if \; z_n > 0 \\ 4294967087 / 4294967088 & if \; z_n = 0 \end{cases}$$

Having this generator, with a period ρ, a transition function T can be defined that $T(s_n) = s_{n+1}$ and $T^{\rho}(s) = s$, being s_n the state of the generator at step n. To partition the period of the generator, two numbers v and w, being $z = v + w$, are selected. The period is divided into adjacent streams of length $Z = 2^z$, and each stream is divided into $V = 2^v$ sub-streams of length $W = 2^w$. Selecting $v = 51$ and $w = 76$, the generator period is close to 2^{191}, and can be divided into disjoint streams of length 2^{127}. At the same time, each stream can also be divided into 2^{51} adjacent sub-streams, each of length 2^{76}.

Being s_0 the initial seed, I_g is the initial state of the stream g, having $I_1 = s_0$, $I_2 = T^Z(s_0)$ and so $I_g = T^{(g-1)Z}(s_0)$. The first sub-stream of g starts in I_g, the second in $T^W(I_g)$, the third in $T^{2W}(I_g)$, and so on. C_g denotes the state of the generator at a given moment of the execution. B_g denotes the initial state of the sub-stream that contains C_g. And N_g denotes the initial state of the next sub-stream to B_g. Any sub-stream can be selected to generate random numbers using the transition function T over any of these states.

2.7.4 Random Variates Generation in Arena

Using the CMRG generator as source of uniform random numbers, Arena provides multiple functions to generate random variates following multiple probability distributions. Some of the included probability distributions are shown in Tables 2.3 and 2.4.

Table 2.3: Some discrete probability distributions supported by Arena.

Empirical Discrete $(\{cp_1, ..., cp_n\}, \{v_1, ..., v_n\})$
Bernoulli (p)
Discrete Uniform (i, j)
Binomial (t, b)
Geometric (b)
Negative Binomial (t, b)
Poisson (λ)

Table 2.4: Some continuous probability distributions supported by Arena.

Empirical Continuous $(\{cp_1, ..., cp_n\}, \{v_1, ..., v_n\})$
Uniform (a, b)
Exponential (β)
Erlang (β, m)
Gamma (α, β)
Weibull (α, β)
Normal (μ, σ^2)
LogNormal (μ, σ^2)
Beta (α_1, α_2)
Johnson $(\alpha_1, \alpha_2, a, b)$ bounded if $\alpha_2 > 0$
Johnson $(\alpha_1, \alpha_2, \gamma, \beta)$ unbounded otherwise
Triangular $(min, mode, max)$

2.8 Conclusions

Multiple languages and environments currently support the description of hybrid systems. Some approaches are focused on modeling continuous-time systems and support the management of time and state events. Other approaches are focused on modeling discrete-event systems and include some functionalities to

describe the continuous-time part of the system (i.e., numerical integration algorithms). Formalisms, modeling languages or simulation tools are usually designed for modeling and simulation of either continuous-time or discrete-event models. They are later extended including some functionalities to describe hybrid systems. The combination of continuous-time and discrete-event modeling methodologies could facilitate the description of hybrid models.

Modelica is one of the most advanced languages for continuous-time and hybrid system modeling. The equation-based object-oriented methodology, supported by Modelica, provides several advantages in comparison with the block-diagram modeling approach. The acausal description of models using differential, algebraic and discrete equations is one of these advantages.

The use of mathematical formalisms to describe discrete-event models facilitate its development, maintenance, reuse, validation and correctness. P-DEVS is a modeling formalism that supports the description of discrete-event, discrete-time and continuous-time models. Also, the process-oriented approach allows to perform a natural description of logistic systems, in terms of entities, resources and processes. It is a widely used approach to describe systems in academia and the industry.

Modelica currently supports multiple discrete-event modeling formalisms, like Petri Nets, Classic DEVS, StateCharts and StateGraphs. However, the P-DEVS formalism and the process-oriented approach are currently not supported. The requirements needed to support these two discrete-event modeling approaches in Modelica will be discussed. The integration of the P-DEVS formalism with Modelica could facilitate the description of the discrete-event part of a hybrid system in Modelica (using the P-DEVS formal specification, instead of just language constructs), and the combination of P-DEVS models with other already available Modelica components. This development will also help to introduce the DEVS formalism into the Modelica community, without requiring the use of different modeling tools. The Modelica language could be a vehicle for combining the use of the DEVS formalism with other modeling formalisms and techniques.

3

Integrating the P-DEVS Formalism in EOO Languages

3.1 Introduction

The identification of the requirements needed to describe P-DEVS models using an equation-based object-oriented (EOO) language is discussed in this chapter. These requirements, when particularly applied to the Modelica language, can be seen as the definitions of the challenges to be solved with the development of this dissertation. The proposed solutions constitute the rest of the works presented.

3.2 Identification of Requirements

In this section, the requirements needed to describe P-DEVS models using an EOO modeling approach are discussed. These requirements meet the necessity to describe atomic and coupled P-DEVS models, and the possibility to combine discrete-event and continuous-time models.

3.2.1 Discrete-Event Model Behavior

P-DEVS models, as discrete-event models, have a fixed causality. The actions associated with the events are described algorithmically using functions.

EOO models are described using differential, algebraic and discrete equations. Discrete-time and event management constructs are required to describe the be-

havior of a P-DEVS model in EOO languages. The discrete part of the model can be described in different ways, depending on the functionalities provided by the language itself (i.e., algorithm sections [Elmqvist et al., 1998], concurrent programming language statements [van Beek and Rooda, 2000], operating procedures [Barton and Pantelides, 1994] or event-driven processes [IEEE, 1997; Frey and O'Riordan, 2000]).

In general, EOO languages provide functionalities to manage discrete events. These functionalities have to be combined to reproduce the semantics of P-DEVS models (i.e., event detection, management and execution of transition functions), in order to facilitate the description of P-DEVS models in EOO languages.

3.2.2 Model Communication Mechanism

Each P-DEVS model, atomic or coupled, has an interface to communicate with other models. These interfaces allow the composition of modular and hierarchical models, in order to construct more complex models. EOO models also contain model interfaces that allow the connection of multiple components in a similar fashion, to construct more complex models. However, the concepts underneath both model interfaces and their connections are different.

Model communication in P-DEVS usually involves the exchange of information. A P-DEVS model can send information to another model connected to one of its output ports, using the output function. The information transmitted from one model to another is the *message*. These messages can transport an arbitrarily amount of complex structured information, from a single number to the description of a customer (as an entity in a system). Thus, P-DEVS models communicate using a message passing mechanism. Connections between models can be in the form of 1-to-1, 1-to-many and many-to-1. Messages can be received simultaneously through one or multiple ports.

On the other hand, the connections between models in EOO languages are based on the energy-balance principle. Variables in the connectors describe either across or through values. Across variables in a node (i.e., a connection point) have the same value, while the through values are summed up and the sum equaled

to zero. For instance, the voltage in a node of an electric circuit represents an across variable (e.g., $A.u = B.u$, if A and B are connectors composed of an across variable u, and connected using the sentence $connect(A, B)$). The current in a node of an electric circuit represents a through variable (e.g., $A.i + B.i = 0$, where A and B are composed of a through variable i).

The amount of information in the connection is fixed, due to the amount of variables in the connector, and can not be modified during the simulation. The structure of this information is also fixed, described by the variables of the connector. The information transmitted using EOO connections (i.e., the values of the variables in the connectors) can only be assigned once at each time instant, not allowing to transmit multiple values simultaneously.

A message passing mechanism has to be defined to be used with EOO languages in order to facilitate the description of P-DEVS models. Ideally, this message passing mechanism should be transparent to the user in order to facilitate the integration of both formalisms without increasing the complexity of model development.

3.2.3 Interfacing P-DEVS and Other Modeling Formalisms

The idea is to combine models described using P-DEVS with models defined using other formalisms for continuous-time modeling (i.e., the physical modeling paradigm), using EOO languages. This combination facilitates the description of multi-formalism hybrid systems.

Two approaches for communicating P-DEVS models with other formalisms are proposed:

- *Translated Interface Connections*: Connecting the output of a P-DEVS model to the input of a continuous-time model, or viceversa. Due to the mentioned differences in the model communication mechanism, it is required to define interface models that translate messages into discrete-time signals, and both continuous-time and discrete-time signals into messages. These interface models allow to couple discrete-event and continuous-time

components together in the hierarchy of models that compose a hybrid system.

The model of a motor of a pendulum clock [Kriger, 2002] can be observed as an example of this type of interaction. The oscillation of the pendulum is described as a continuous-time model. The rest of the clock is modeled as a discrete-event system. Interfaces are required to communicate to the clock the oscillation pace of the pendulum, in form of tics (each tic represented by a message). On the other hand, the oscillation of the pendulum can be stopped and started using some buttons in the clock that generate messages, which need to be translated into discrete-time signals in order to be managed by the continuous-time model.

– *Direct Interface Connections*: Allowing to describe the behavior of a discrete-event model which is influenced by the state of a continuous-time model. P-DEVS models could receive continuous-time or discrete-time signals as inputs to its transition functions. In order to maintain the modularity in the model construction, these inputs must be connected using the model interfaces. These connections are similar to the interactions described in the DEV&DESS formalism between the discrete-event and the continuous-time parts of a hybrid model.

  As an example of these interactions we can consider a crane system controlled by a discrete controller (such example, modeled using DEVSLib, is presented in Sanz, Cellier, Urquia and Dormido [2009]). The controller periodically (executing internal transitions) calculates the control-signals using the current position of the crane, which is represented by a continuous-time signal. The value of the continuous-time variable is considered as an input for the internal transition function of the controller. No sampling is needed to obtain the value of the signal. The environment reads the current value of the signal whenever the internal transition function has to be executed. This signal could be quantized and stored as a state variable of the controller, but it will decrease unnecessarily the simulation performance.

3.3 Requirements Applied to Modelica

This section presents the described requirements when applied to the particular case of the Modelica language. Each of these challenges have been treated during the development of this dissertation, and so references are included to the chapters where the solutions are discussed.

3.3.1 Atomic P-DEVS Models

Modelica provides language constructs to describe the trigger conditions of time and state events, and also the actions associated to the events [Mattsson et al., 1999]: (1) update the value of discrete-time variables and reinitialize continuous-time state variables, using *when* clauses; and (2) change the mathematical description of equations and assignments, using the *if* statement.

These functionalities have been used to describe models following multiple formalisms, like State Charts, Petri Nets, State Graphs and Classic DEVS. As will be demonstrated in this dissertation, the same functionalities can be used to describe the behavior of P-DEVS models. To this end, the detection of the occurrences of internal, external and confluent events has to be defined. Also, the execution of the actions associated with each type of event has to be managed (i.e., the execution of transition functions). The description of atomic and coupled P-DEVS models in Modelica is discussed in Chapter 5. The description of process-oriented models using the developed support for the P-DEVS formalism in Modelica is performed in Chapters 9, 10 and 11.

3.3.2 Modular P-DEVS Models

Modelica functionalities to communicate models follow those of the EOO languages, and are based on establishing relations between ports. Ports in Modelica are named *connectors*, and the relationships between ports are performed using *connect* sentences. Ports are composed of one or several variables. The connection between two ports establishes a relationship between their variables. Modelica

variables inside connectors are defined as across by default, or as through using the *flow* modifier.

However, these language constructs are not enough to describe the required P-DEVS message communication mechanism because:

- They do not allow the simultaneous transmission of messages from one port to another, due to the single-assignment rule (a variable can only be assigned from one equation).
- They do not allow to connect multiple output ports to the same model and transmit simultaneous messages, also due to the single-assignment rule.
- The amount of information transmitted by the connector is fixed by the number of variables in it.
- The structure of the information transmitted with the connection is also fixed due to the variables defined in the connector.

The concepts underneath P-DEVS and Modelica communication mechanisms are different. In order to allow the description of P-DEVS models in Modelica, a message passing mechanism has to be implemented. The proposed and implemented message passing mechanism in Modelica is described in Chapter 4.

3.3.3 Interface Between P-DEVS Models and Models Described Using Other Formalisms in Modelica

In order to combine P-DEVS and Modelica models from other libraries, the two types of model communication have to be fulfilled:

- Multiple interface models have to be constructed to translate the messages, as described in the message passing mechanism described in Chapter 4, into discrete-time signals. Also, the continuous-time and discrete-time signals from the Modelica models have to be translated into messages. A description of the developed interface models is performed in Chapter 7. Also, a description of the functionalities included to construct hybrid process-oriented models is included in Chapter 12.

– The direct connections from Modelica to P-DEVS models can be supported by allowing continuous-time inputs for the transition functions. The value of the continuous-time signal connected to one of these inputs is used as an input for the transition function. A definition of this extension to the formalism is included in Chapter 5, together with the description of the atomic P-DEVS model behavior in Modelica.

3.4 Conclusions

The differences between P-DEVS and EOO models have been identified. These differences can be considered as requirements in order to support P-DEVS models in EOO languages. The requirements to describe P-DEVS and process-oriented models in Modelica are:

– The description of discrete-event model behavior.

– The description of model communications following a message passing mechanism.

– The description of model interfaces to combine discrete-event models with models from other Modelica libraries.

The study of these requirements and the development of solutions will be the base of the works described in the rest of this dissertation.

4

Message Passing Mechanism in Modelica

4.1 Introduction

The most important challenge found during the development of the DEVSLib library has been the communication between P-DEVS models. P-DEVS models communicate following a message passing mechanism. Modelica, using its current functionalities, does not facilitate the description of a message passing mechanism. The description of this communication mechanism in Modelica is the cornerstone for supporting P-DEVS models.

In this chapter, the work performed to facilitate the description of the P-DEVS model communication mechanism in Modelica is discussed. The differences between both communication approaches are described, identifying the required functionalities for message passing. The specification and design of the message passing mechanism is generalized to the EOO languages. The elements, messages and mailboxes, defined to perform the communication using message passing are discussed, together with the operations required to manage them. An example of communication between models using the defined mechanism is included. After that, different approaches are analyzed to manage the communication of messages between Modelica models. The selected approach, based on dynamic memory management, is used to implement the mechanism as an external library coded in C. A default message type is described together with the functions to manage

it. Finally, the use of this mechanism to describe P-DEVS model communication is discussed. The implemented mechanism will be used in the development of the DEVSLib library.

4.2 Definition of the Problem

As described in Chapter 3, the concepts underneath P-DEVS and Modelica model communication mechanisms are different. The communication between P-DEVS models follows a message passing mechanism. Models can share information, transmitted as impulses of structured data through connections (i.e., the *messages*). The communication between Modelica models follows the energy-balance principle. This communication establishes relations between variables (across and through) inside connected ports (i.e., *connectors*).

The current Modelica communication mechanism (using *connectors* and *connect* sentences) does not facilitate the description of a message commmunication mechanism. The constructs included in the language:

- Do not allow the simultaneous transmission of messages from one port to another, due to the single-assignment rule (a variable can only be assigned from one equation).
- Do not allow to connect multiple output ports to the same model and transmit simultaneous messages, also due to the single-assignment rule.
- The amount of information transmitted by the connector is fixed by the number of variables defined inside it.
- The structure of the information transmitted with the connection is also fixed due to the variables defined in the connector.

Model communication in P-DEVS involve additional things than in Modelica, because of the transmission of information. These differences can be extended to any EOO language. In order to facilitate the description of P-DEVS models in EOO languages, a message passing mechanism has to be described and implemented.

4.3 Required Functionalities of the Message Passing Mechanism

In order to reproduce the information exchange between P-DEVS models, the message passing mechanism should satisfy the following requirements:

1. Connections between models should be of any form (i.e 1-to-1, 1-to-many and many-to-1).
2. Messages should be transmitted/received instantaneously (i.e., without delay).
3. Multiple messages could be transmitted simultaneously through the same or multiple ports of a model.
4. Messages should be able to transport any kind of information. The information transported by the message could be different in each transmission. The type and structure of that information should be defined by the modeler.

Ideally, the mechanism should be transparent to the user in order to facilitate its use without increasing the complexity of model development.

4.4 Specification and Design of a Message Passing Mechanism for EOO Languages

A general communication model can be described using three concepts: the sender, the receiver and the communication channel. The proposed message communication mechanism replicates this structure, defining elements to represent each of these concepts and the operations performed by each element in order to accomplish with the communication. The sender is naturally represented by the model that initiates the communication. The receiver is represented by a *mailbox* in the model that receives the communication. The communication

channel is represented by the *message* transmitted between them. The mailbox and the message are the data structures proposed to describe the message passing mechanism. The behavior of these two structures is detailed, as well as the operations to perform model communication usign them (i.e., message sending, tranmission, detection, and treatment).

4.4.1 Messages and Mailboxes

The model communication mechanism using messages involves two elements:

- The *message* itself. The message represents the information either traveling inside a model or from one model to another.
- The *mailbox*. The mailbox receives the incoming messages and stores them until they are read. The mailbox also represents the concept of a bag of messages in the P-DEVS formalism.

The specification of the characteristics of the mailboxes and the messages, and their behavior, are the following:

- The *content* of the message, this is the information transported, can be of different types. A message could transport a single number, an array, a record or any other object, defined as an instance of a class. It is the modeler who defines the contents of the messages and the way to manage them. For instance, in a typical communication between P-DEVS models the messages transport single numbers. However, in the implementation of the QSS systems at least three values are required (the value, the first and the second derivatives, in case of the third order integrator), which are transported using an array. In the communication between Arena modules the information transported by each message represents an entity, with pre-defined atributes and also a variable number of user-defined atributes. Each entity can transport a different number of attributes, depending on the flowchart diagram of the system.

- Messages can be of different types. Different types of messages can be used to establish priorities in their management or to describe different types of information in the system (e.g., parts of type A and B) Also, the type of a message is independent from its content. Messages of the same type can trasnport different contents, and vice-versa. A mailbox can store any message independently of its type.

- The mailbox warns the model when new incoming messages are received. This behavior prevents from unnecessarily checking the mailbox for new messages.

- Once received, the message can be read from the mailbox.

- The transmission of messages between models is performed instantaneously. Any message sent from one model will immediately be received by another model.

- Messages can be received simultaneously, either in the same or different mailboxes.

- Received messages have to be stored temporarily in the mailbox, until they are read.

- Message communication has to be performed in two stages: sending and reception. The former involves the transmission of any message in the system at a given point in time, so all the messages sent are stored in the mailbox at the end. After that, all the messages are available for reception in each mailbox and can be read and managed as required. If a model sends several messages to the same mailbox, all the sent messages have to be stored in the mailbox before the first message can be read by the receiver.

The design of these two elements, in order to include them in an EOO language, can be performed as follows.

The message can be described using two components: the type and the content.

- The *type* of a message can be represented using an integer value. Different values can be assigned in order to separate the messages of the system into different sets. The type can also be used to organize the messages stored in the mailboxes or as a priority to read them from the mailbox, similarly to the behavior of UNIX IPC messages mechanism.

- The *content* represents the information transported by the message. The content of a message is defined by the modeler. Depending on the characterstics of the language, the content can be defined as a general abstract class, whose characteristics can be inherited to describe particular classes for different types of contents. It can also be described as a reference (i.e., a pointer) to the user-defined data structures used to describe each particular type of content. In any case, the modeler has to provide enough functionalities to manage the defined types of contents.

The mailbox represents a temporary storage for messages. When a message is sent to a mailbox, it is stored in the mailbox until the receiver reads it. The number of messages stored in a mailbox is not limited, so this structure has to be able to dynamically change its dimension. A mailbox has to be described using linked lists (of messages) stored using dynamic memory.

4.4.2 Communication Using Messages and Mailboxes

The transmission of messages between models can be performed directly or between connected mailboxes:

- In a *direct transmission*, the sender composes the message (i.e., defining its content) and sends it to a mailbox in the receiver using a pre-defined function (i.e., $sendMessage()$). In this case, the reference to the mailbox in the receiver has to be known in advance by the sender (e.g., using a pre-defined mailbox name). This requirement does not satisfy the modularity principle of the object-oriented methodology, and so its use is not recommended. However, this method also provides a simple mechanism to communicate

information about concurrent or immediate events inside a model itself (i.e., being the sender and the receiver at the same time).

Mailboxes can also be shared between models. Sharing a mailbox implies that several models can access to the message storage that it represents. Each model sharing the mailbox can access the stored messages, reading or extracting them from the mailbox. Read messages are kept in the mailbox until they are extracted, and removed, from it. Messages that have been read by a model are still available for other models until they are extracted.

– *Connected mailboxes* are a special case of mailboxes which are defined inside connectors. Two mailboxes, inside connectors, connected using a connect sentence represent a bidirectional message communication pipe. They will act as input/output mailboxes instead of only receiving messages, like the mailboxes in the direct transmission. A message sent through one end of the pipe will be transported to the opposite end, and vice-versa. If more than two models are connected to the same pipe, a copy of the message will be transported to each receiver connected to the pipe. This provides a message broadcast functionality that also emulates the message transmission in P-DEVS, however in P-DEVS the communication is not bidirectional. The functionalities of the connect sentence in Modelica have to be extended in order to support this mailbox behavior.

  An example of this behavior is shown in Fig. 4.1. Model A defines a new message m, and sets its type to 1 (using the function *settype*()) and its content to c (using the function *setcontent*()). When composed, it sends the message to the mailbox *mbox* (which is defined inside a connector) using the function *putmsg*(). The message is then copied to the mailboxes in models B and C. Each of these models is waiting for the reception of a new message, using the function *checkmsg*() inside a when statement (which will be active after the reception of a new message). When received, models B and C extract the message from the mailbox (using the *getmsg*() function) and get the type of the message (*gettype*() function). Model B is

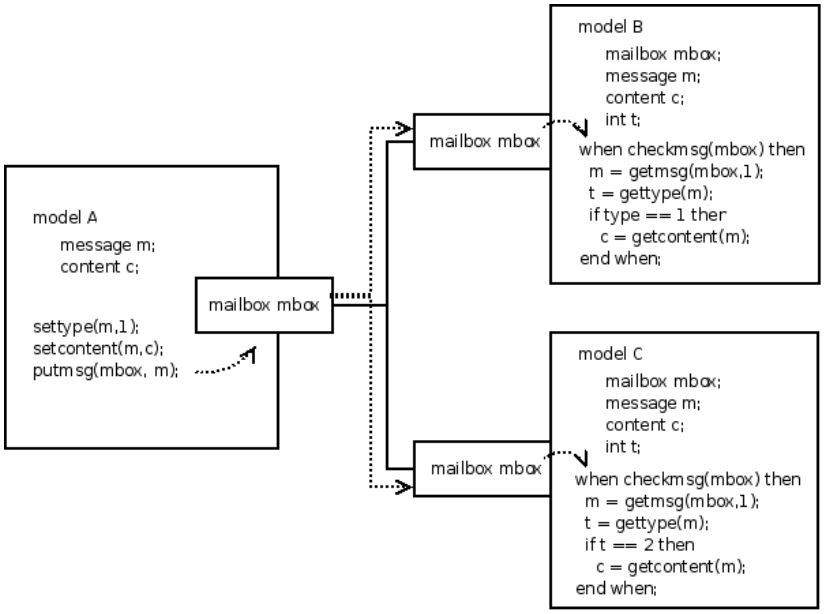


Figure 4.1: Model communication with messages using connectors.

waiting for messages with type 1, and model C is waiting for messages with type 2. Thus, model C discards the message and model B reads its content (using function *getcontent*()).

The detection of message reception is implicit in the action of sending it, since they are transferred instantaneously. Every time a model sends a message to a mailbox, the simulator knows that the message will be received by another model and that event will have to be treated properly. At that time, the simulator can schedule a new time event that will represent the reception of the message (even if this situation occurs immediately and no simulation time elapses). In this way, events for message reception and treatment can be scheduled properly in advance.

The mailbox warns when a new message has arrived. The mailbox activates a listener function that can be used as a condition to detect any incoming message (e.g., used with the *when* or *if* statements in Modelica). This does not mean that the condition has to be effectively checked at each simulation step, because it is notified by the send message operation. Once a new message arrives to a mailbox, the arrived message or messages have to be read and treated. The management of the content of each message has to be defined by the user.

Following with the design of the message passing mechanism, several operations to manage the behavior of messages and mailboxes are proposed. The functions and their description are shown in Tables 4.1 and 4.2. The operations are described as function prototypes, where the parameters indicate the class (or type) of the objects received as inputs.

Table 4.1: Operations with mailboxes.

Operation	Description
`newmailbox(mailbox)`	Initializes the `mailbox`.
`checkmsg(mailbox)`	Warns about the arrival of a new message. It changes its value from false to true and immediately back to false at each message arrival event.
`newmsg()`	Detects the arrival of a message to any of the mailboxes declared in the model. This helps to manage the simultaneous arrival of messages in different mailboxes.
`nummsg(mailbox)`	Returns the number of messages stored in the `mailbox`.
`readmsg(mailbox,select)`	Reads a message from the `mailbox`. The `select` parameter represents a user-defined function used to select the desired message to be read from the mailbox (e.g., the first, last, a given position of the list, or using a particular condition).
`getmsg(mailbox,select)`	Extracts a message from the `mailbox`, deleting it. The `select` parameter is used in the same way as in the `readmsg` function.
`putmsg(mailbox,message)`	Sends the `message` to the `mailbox`.

Table 4.2: Operations with messages.

Operation	Description
`newmsg(content,type).`	Creates a new message with the defined `type` and `content`.
`gettype(message).`	Returns the type of the `message`.
`settype(message,newtype).`	Updates the type of the `message` to the value of `newtype`.
`getcontent(message).`	Reads the content of the `message`.
`setcontent(message,newcontent).`	Inserts the `newcontent` into the `message`.

4.4.3 Example of Model Communication Using Messages

In order to show the behavior of the model communication mechanism using messages and mailboxes, the following example is described. It corresponds to a single-queue system, described using similar elements to the ones found in the SIMAN language.

The structure of the model is shown in Fig. 4.2. It is composed of a Create, Queue, Seize, Delay, Release and Dispose models. The communication between these models is performed using messages. The input and output ports used to define the interface of the models contain mailboxes. As described above, the connection between two ports establishes a communication channel. Messages will be sent from one side and received at the other.

The Create model periodically creates new messages and sends them through its output port. These messages are received by the Queue model and are stored there waiting to be procesed. After receiving a new message, the Queue model creates an auxiliary message ($maux$) and sends it to the Seize model after setting its value to 0 and the type to 1 (the value of the type will be used to differenciate input messages in the Seize model). The Seize model can receive two types of messages:

- Type 1, that representes a new message stored in the queue. If this type of message is received, the Seize model increases the counter for the number of messages waiting in the queue and checks the state of the resource. If the resource is idle: (1) it seizes the resouce (using the function $seize(resource)$); (2) extracts a message from the input mailbox of the Queue model that contains the messages waiting (the Seize model has to obtain the reference to this mailbox either as a parameter or using a pre-defined value); (3) decreases the counter for the number of messages in the queue; and (4) finally sends the extracted message to the next model.

- Type 2, that represents messages arrived from the Release model meaning that the resource has been released. If this type of message is received, the Seize model checks the counter of messages to see if there is any mes-

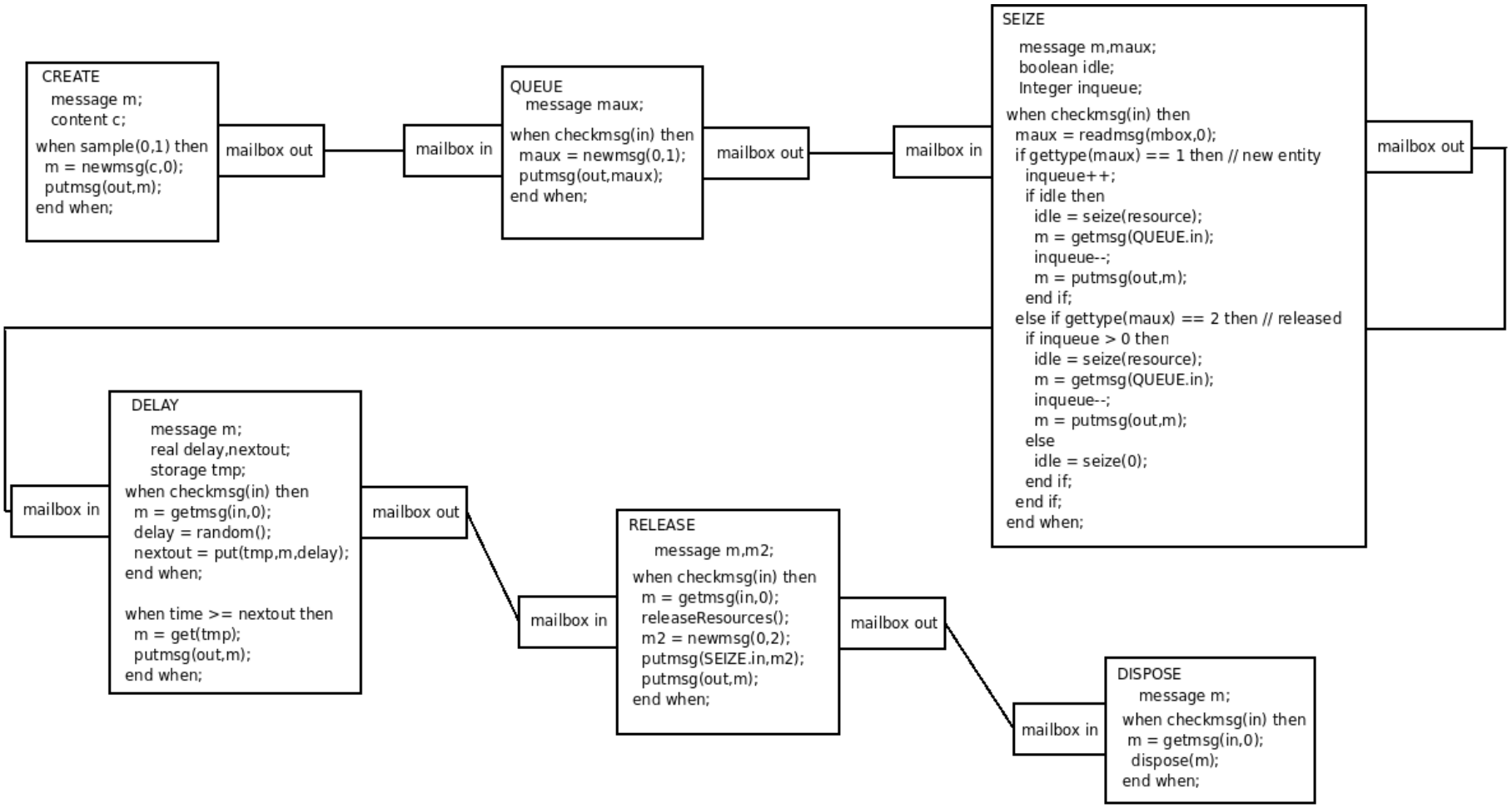


Figure 4.2: Example of a SIMAN single-queue system modeled using messages.

sage waiting in the queue. If any message is waiting, the recently released resource is assigned to a new message, which is extracted form the queue and sent to the next model. If no messages are waiting, the Seize models updates the state of the resource (to idle) and waits for new messages.

The messages sent from the Seize model arrive to the Delay, that represents the time the message is being processed. The received messages are assigned with a random delay and stored in a temporary storage using the function *put*(). This function returns the time (*nextout*) of the first delay to finish, between the messages stored in the storage. When the simulation time reaches the value of *nextout*, the Delay model extracts the message from the storage and sends it to the Release model. When the Release model receives a message, it relaeases the resource (using the function *releaseResources*()) and redirects the message to the next connected model. It also creates a new message ($m2$) with type 2 that will be sent to the Seize model to notify the release of the resource. Finally, the messages arrive to the Dispose model that removes them from the system.

4.5 Analysis of Alternative Implementations of Message Passing Communication in Modelica

Due to the current functionalities of the Modelica language (i.e., the use of the synchronous data flow principle, the single-assignment rule, the imposibility to define a variable number of objects in a model or to define data structures of a varible size), only a partial implementation of the described mechanism has been performed. The message passing mechanism has been implemented in an external library written in C and named "events.c". It contains multiple data structures and functions to reproduce as close as possible the described behavior of messages and mailboxes. These external data structures and functions are connected with Modelica using the external function interface included in Dymola [Olsson, 2005].

The approach used to perform the message transmission is completely transparent to the final user. At the user level, the communication is just defined by

connecting the output ports of some models to the input ports of other models. The transmission and reception of messages is performed using Modelica functions.

Several approaches were studied and developed in order to implement a suitable message passing mechanism in Modelica. The approaches studied and implemented while developing the message passing mechanism are described next. A direct implementation using modelica connectors, the use of a temporary storage described using text files and the use of dynamic memory to store the transmitted messages were evaluated. Finally, the dynamic memory approach was used to implement the communication mechanism.

4.5.1 Direct Transmission

A direct implementation of a message passing mechanism using Modelica connectors was studied. It consists in specifying inside the connector all the variables that define a type of message. The values assigned to the variables of one connector represent the content of the message.

These values are related, because of the connect sentence, to the values of the variables in the connector of the next model. In this way, a message is directly transmitted from one model to another. Different types of messages require different connectors, one for each type, with different variables.

The direct transmission is the simplest way of communicating models, but presents a problem: the simultaneous reception of several messages. There are three possible situations for this problem:

- 1-to-1 connection: one model sends several messages to another model at the same time.
- Many-to-1 connection: several models simultaneously send messages to another model.
- A combination of the previous cases: several models simultaneously send one, or more, messages to another model.

The following solutions have been applied to this problem:

1. Synchronizing the message transmission between models using semaphores. The synchronization allows the sender and receiver to manage the flow of messages between both models, using a send/ACK mechanism like in the TCP/IP communication. Thus, the sender model will send a message to the receiver and wait for an ACK. On the other hand, the receiver model will read the messages when it is ready to process them, and only send the ACK back if it is still ready to continue processing more messages. A model of the semaphore synchronization mechanism, based on a previous work by Lundvall and Fritzson [2003], has been implemented and is freely available for download at Euc [2009]. A disadvantage of this solution is the performance degradation due to the event iteration that takes place during the synchronization phase of the message transmission. The characteristics, structure and use of the semaphore model are detailed in Appendix A, since this approach is not used in the final implementation.

2. Including in the connector a through (i.e., *flow*) variable that represents the number of messages sent from a model. So, the model receiving the messages will know the number of messages received, even with many senders because the number of messages sent from each sender will be summed up.

However, the content of multiple messages can not be transmitted simultaneously using the direct transmission approach. The variables of the connector that describe the message can not be assigned with different values, that represent the different contents, at the same time.

4.5.2 Text File Storage

The other analyzed approaches for implementing the message passing mechanism are based on using an intermediate storage of the transmitted messages. This storage behaves as a communication buffer between two or more models.

The first approach was to use a text file to store the messages, so the sender writes the message to the file and notifies it to the receiver, that subsequently

reads the written message. This approach allows simultaneous reception of messages, because several messages can be written to the file, but its performance and versatility are poor.

The storage is implemented in a text file. Each line of text stores the information related to each of the transmitted messages. The connector contains a reference to the text file (e.g., the file-name) and the flow variable indicating the number of messages received (as described in the direct approach). The reference to the file is shared between the models connected to that connector, allowing them to access the file simultaneously.

Each model able to receive messages (i.e., with an input port) creates a storage text file and sets the reference to that file in the connector. Functions to read/write messages from/to the file were developed. A model writes one or several messages to the file using the write function. Another function can be used by the receiver to check the number of messages in the file. When a new reception is detected, either by checking the file or detecting a change in the value of the flow variable in the connector, the receiver reads the messages and processes them. Thus, this approach allows the simultaneous reception of several messages.

A disadvantage associated with this approach is the poor performance due to the high usage of I/O operations to access the files. Also, the structure of the information stored in the files is not very flexible if any additional information has to be included. If new types of messages need to be used, the file management functions (i.e., read and write) have to be re-implemented to correctly parse the text file to support the new changes.

4.5.3 Dynamic Memory Storage

In order to improve the performance of the text file approach, the intermediate storage was moved from the file-system to the main memory. Using the Modelica external functions interface, a library in C was created to manage the intermediate storage using dynamic memory allocation. A message is represented in Modelica using a *record* class, and in C using its equivalent *struct* data structure. Messages

are stored using dynamic linked lists during their transmission from one model to another.

Instead of a reference to the text file, the connector contains a reference to the memory space that stores the messages, together with the flow variable that indicates the number of messages received. That reference is the memory address pointing to the beginning of the linked list. Similarly to the file text approach, each model able to receive messages initializes the linked list as a queue (i.e., using the *createQueue*() function), and sets the reference to it in the connector. Messages can be transferred to the queue using the write function (i.e., *sendEvent*()), and can be extracted using the read function (i.e., *getEvent*() or *readEvent*()). Another function is used to check the availability of received messages (i.e., *numEvents*()), in order to process them.

This approach also allows the simultaneous reception of several messages. The performance is highly increased in comparison to the text file approach. Also, the structure of the information only depends on the data structures managed by the functions and can be easily adapted. For modifying the message type, it is only necessary to change a data structure and not all the functions used to manage that structure.

4.6 Implemented Message Passing Mechanism in Modelica

The implemented message passing mechanism in Modelica is described in this section. It consist on the description of a default message type, that will be used to transport information between models, and the required functions the manage this message type. These functions implement the management of the communication using the described dynamic memory approach. The re-definition of the default message type, in order to adapt it to other applications, is also described in this section.

4.6.1 Default Message Type

The default type of message is composed of: *Type*, represented by an integer variable, and *Value*, which is represented by a real variable. The message also includes a *Port* variable, that represents the port the message has been received through, but this value is managed by the receiver model and not by the user. This structure is defined in the "events.c" file as a C struct, and in Modelica as a record named *stdEvent*.

This default message can be used to define the content of a message as real or integer numbers, or a combination of both. Also, as several messages can be simultaneously sent through an output port, this type of message can be used to transmit arbitrarily complex information (e.g., arrays of numbers).

4.6.2 Functions to Manage the Default Message Type

The functions used to manage the default type of message and the linked lists (or queues) used to store them during the transmission are:

QCreate Creates a new queue to store messages.

QDestroy Deletes the messages and frees the memory of an already created queue.

QAdd Inserts a new message in the queue. The message is inserted at the end of the queue (FIFO).

QAddFirst Inserts a new message in the queue. The message is inserted at the beginning of the queue (LIFO).

QAddLVF Inserts a new message in the queue. A ordering value is used to select the position in the queue. Each message has an associated value, and the messages are ordered from lower to higher values (LVF, or Low Value First).

QAddHVF Inserts a new message in the queue. Like in the LVF, an ordering value is used to select the position of the messages. They are ordered from higher to lower values (HVF, or High Value First).

QRead	Reads the first message from the queue without deleting it.
QGet	Reads the first message form the queue and deletes it.
QGetPos	Reads the message located in a given position in the queue and deletes it.
QSize	Returns the length of the queue.
QFirstTime	Returns the insertion time of the first message in the queue.
QFirstOrder	Returns the ordering value of the first message in the queue.
QPosOrder	Returns the ordering value of the message in a given position in the queue.

4.6.3 Defining Other Types of Messages

The implemented message passing mechanism allows the user to define the content of each message. There are two ways of re-defining the type of message:

- Changing the C structure in the "events.c" file and the content of the *stdEvent* in Modelica.
- Describing the new message type as a new structure in C, including the functions to manage it (i.e., read, write, etc.), and using the *Value* variable of the standard message type as a reference to the new type, whose object should be stored in dynamic memory. This approach is used in the development of the ARENALib and SIMANLib libraries.

4.7 P-DEVS Model Communication in Modelica

The communication of P-DEVS models in Modelica uses the described implementation of the message passing mechanism, using the default message type. The input and output P-DEVS ports are represented using two Modelica connectors: *inPort* and *outPort*. These two connectors are composed of one across variable, named *queue*, and one through variable, named *event*.

The *event* variable represents a counter of the received messages in an input port. Every time a message is sent through an output port, the *event* value of that port is increased. As *event* is a through variable (or *flow*), all the values of the *event* variables from the output ports connected to an input port are summed up, giving the final number of messages received in that input port. The *event* corresponds to the *flow* variable described with the implemented message passing mechanism to allow the simultaneous reception of multiple messages, even from different senders.

The *queue* variable represents the reference to the dynamic memory space used to temporarily store the received messages until the model executes its external transition. The messages are read, and deleted, from the memory by the external transition function. However, in order to facilitate the management of simultaneous messages, messages can be read arbitrarily – i.e., non-sequentially, using an index – without deleting them from memory.

An example of the communication between P-DEVS models in Modelica, using the *inPort* and *outPort* connectors, is shown in Fig. 4.3. Models A and B have connectors of type *outPort*. These ports are connected to a single input port, of type *inPort*, defined in model C. Model C initalizates the value of its "queue" with a reference to the dynamic memory storage used to store the messages received. The values of the "queue" variables of models A and B are equaled to the same reference, because they are defined as across variables (A.queue = B.queue = C.queue). Thus, models A and B can send messages to C using this reference and the *QAdd*() function. The "event" variable will be used by A and B to notify new messages sent to C after executing the mentioned function (by increasing its value when a new message is sent). The value of the "event" variable of C is equaled to the sum of the other two "event" variables, due to the included flow modifiers (A.event + B.event = C.event). Model C can observe the reception of new messages by checking the variations in the value of its "event" variable.

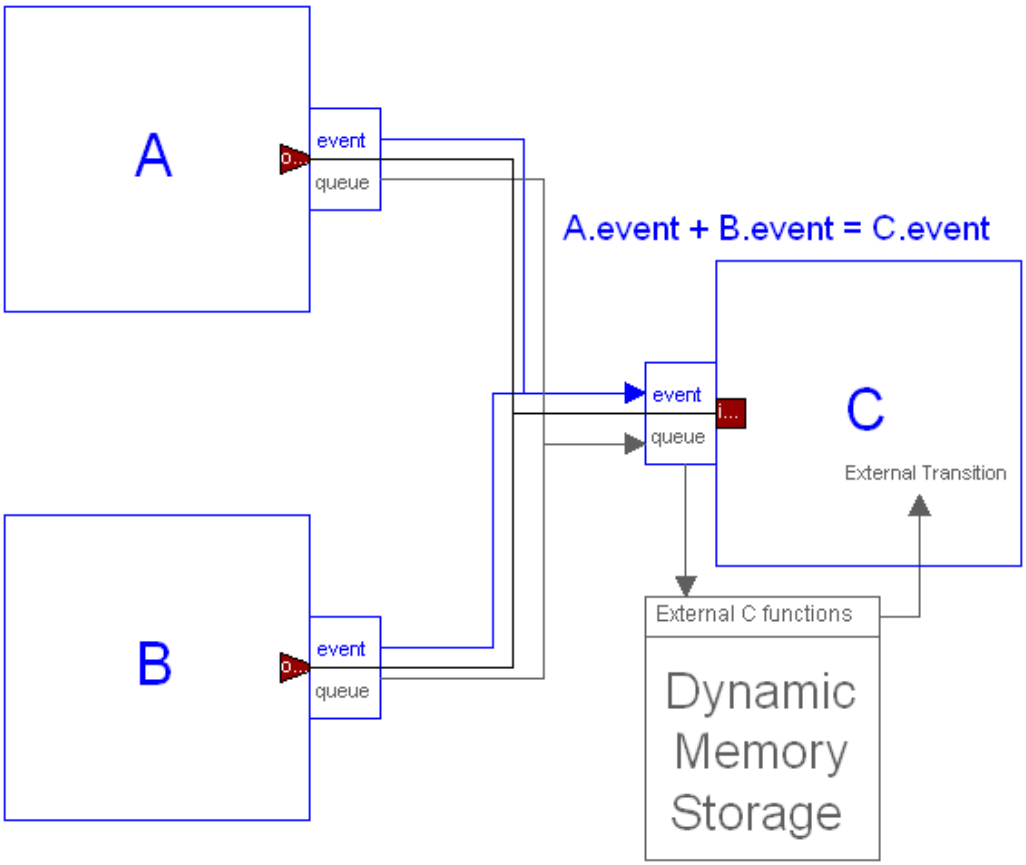


Figure 4.3: Example of P-DEVS models communication scheme in Modelica.

4.7.1 1-to-Many Connections

Using the implemented message passing communication mechanism is not possible to perform 1-to-many connections between models (like the connection shown in Fig. 4.1). This limitation arises because each input port has a queue for storing incoming messages. An output port of a model connected to the input port of another model receives the reference to that queue, used to write the transmitted messages. Each output port can send messages only to one input port, because the queue variable in the connector can not be assigned with several values (corresponding to the references of the queues that will have to receive the message).

A possible solution is the inclusion of an intermediate model to duplicate the received message and simultaneously send copies of it to several receivers. This

model should have several output ports, each one connected to a receiver, that will be used to send the copies of the message. Several output ports can send messages simultaneously to the same input port, because all of them share the reference to the same queue (as shown in Fig. 4.3).

4.8 Conclusions

The communication mechanism between models in P-DEVS are different to the communication mechanism used by most of the EOO languages. Models in P-DEVS communicate using a message passign mechanism, that allows to transfer impulses of information between models. Connections between models in EOO languages are based in the energy-balance principle, using variables defined as across and through. In order to describe P-DEVS models using Modelica, a message passing mechanism has to be described and implemented.

A message passing mechanism has been specified and designed to be included in EOO languages. It is based on the definition of messages and mailboxes as elements to describe the communication, and the operations to manage these elements.

Multiple alternatives to implement the described communication mechanism in Modelica have been evaluated and implemented. The approach selected for the implementation uses dynamic memory to transmit messages between models. Using this approach, a default message type is defined and the operations to perform the communication using messages in Modelica have been implemented. However, this default message type can be easily adapted to other applications. The implementation of the mechanism has been performed as an external library coded in C language, connected to Modelica using its external function interface.

The implemented mechanism has been used to describe the P-DEVS model communication approach in Modelica. It will facilitate the description of P-DEVS and process-oriented models. The implemented mechanism is transparent to the user, and connections between models are performed using Modelica connectors and connect sentences.

5 The DEVSLib Library

5.1 Introduction

DEVSLib is a new free Modelica library, distributed under the Modelica License 2, that supports the P-DEVS formalism. In this chapter the architecture and the implementation of the library are presented.

The implementation is detailed describing how the behavior of atomic and coupled P-DEVS models has been expressed using Modelica. The functionalities provided by Modelica to describe abstract classes, replaceable objects and functions, as well as the functionalities for event management, has been used to perform this implementation. The communication between models in DEVSLib has been described using the mechanism presented in Chapter 4.

The developed library supports a modular and hierarchical description of P-DEVS models, which facilitates the construction of multiple kind of models and their understanding. The development of discrete-event models using the DEVSLib library, and its functionalities to describe hybrid systems will be described in Chapters 6, 7 and 8.

5.2 DEVSLib Architecture

In order to facilitate the understanding and use of DEVSLib, its models can be classified into two groups: the "user's area" and the "developer's area". This division helps to focus on the library components required by the task to perform. A modeler that only needs to construct a new discrete-event model will be focused on the user's area. Modelers that require additional functionalities than the ones included in the library will be focused in the developer's area.

The top level of the hierarchy is shown in Fig. 5.1a. The "user's area" consists of the *User's Guide*, the *atomicDraft* package, the *coupledDraft* model, the *Aux-Models* package and the examples provided within the *Examples* package. The "developer's area" consists of a single package – i.e., the *SRC* package.

5.2.1 User's Area

The "user's area" contains all the models intended to be used directly by the library user. In particular, those needed to develop atomic and coupled P-DEVS models, to interface with continuous-time models and also the models implementing the QSS integration methods. The documentation of these packages is oriented to those who need to use the library, but do not need to understand its internal design and implementation.

The structure of the "user's area" is shown with more detail in Fig. 5.1b. The *atomicDraft* package and the *coupledDraft* model are used to define new atomic and coupled P-DEVS models. The *AuxModels* package contains some useful auxiliary models that are usually needed. It includes the following models:

- Generator and Display are models that can be used as source and sink of messages, respectively.
- DUP and DUP3 are models that duplicate each incoming message and instantaneously send a copy of it through all its output ports (two in the case of DUP, and three in DUP3). The use of these models is detailed in Section 5.5.

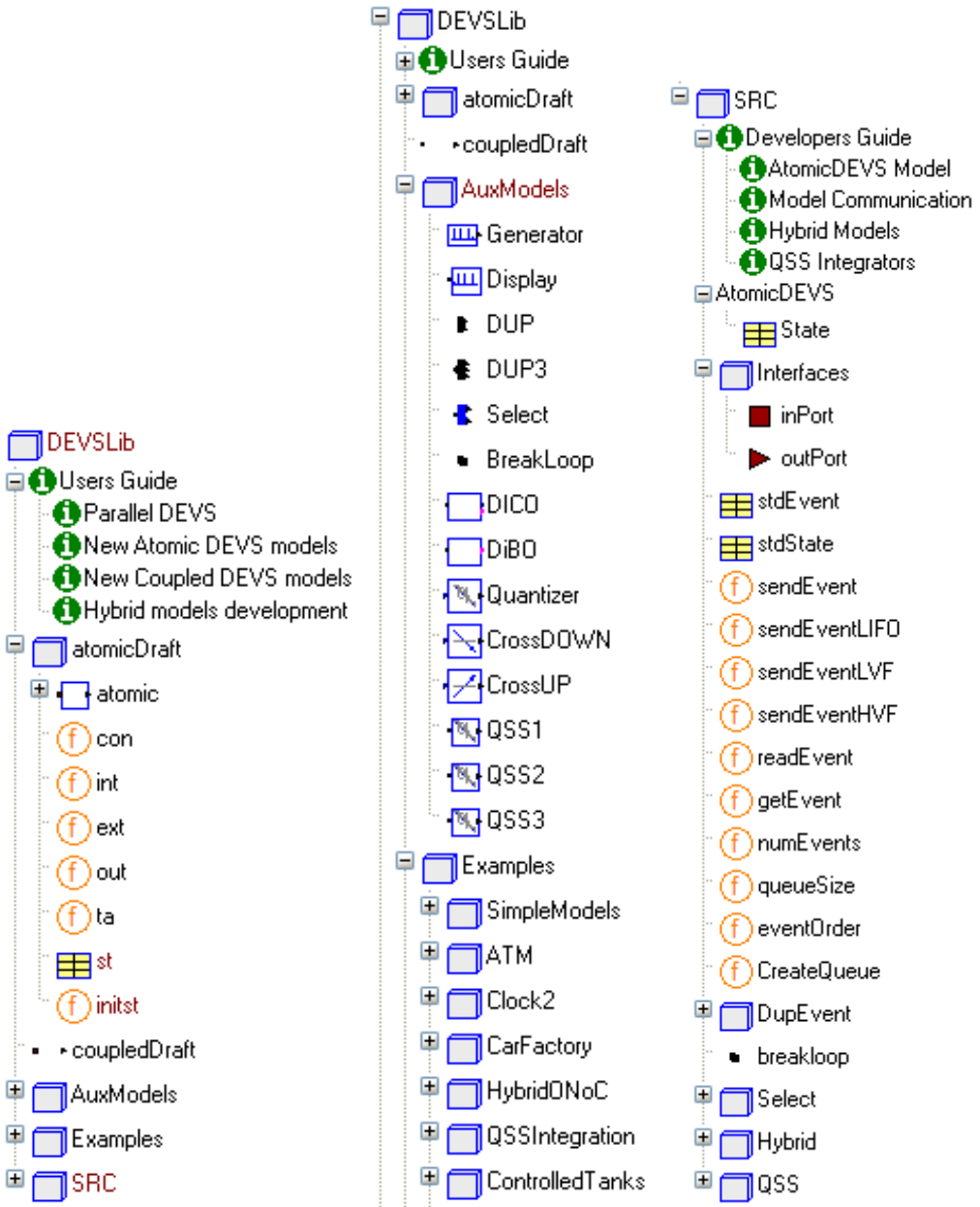


Figure 5.1: DEVSLib library architecture: a) general architecture; b) user's area; and c) developer's area.

- Select is a model that sends each received message through one of its two output ports, depending on a given boolean condition.
- BreakLoop is used to break algebraic loops in coupled models. Its use is detailed in Section 5.5.
- DiCO, DIBO, Quantizer, CrossUP and CrossDOWN are the interface models used to combine DEVSLib models with models from other Modelica libraries. Their use is detailed in Section 7.2.
- QSS1, QSS2 and QSS3 are models that implement the first, second and third order QSS integration methods. Their use is detailed in Section 6.5.

The *Examples* package contains several models that can help the user to learn and understand the use of the library. The included models are the following:

- SimpleModels package contains some simple atomic and coupled DEVSLib models. Implementations of Generator and Display are provided, as well as Processor, Switch, Pipe an other examples described in Zeigler et al. [2000].
- ATM package contains the model of an Automatic Teller Machine. The formal specification of this model can be found in Saadawi [2004].
- Clock2 includes the model of a pendulum clock. It is modeled as a hybrid system, with the pendulum represented by a continuous-time model and the rest of the clock by a P-DEVS model. The specification of the model can be found in Kriger [2002].
- CarFactory includes a model of a car production factory [Sun, 2001].
- HybridONoC contains a hybrid model of an optoelectrical communication system. Detailed information about the model can be found in Sanz, Jafer, Wainer, Nicolescu, Urquia and Dormido [2009].
- QSSIntegration includes a differential equation, a Lotka-Volterra and a flyback-converter models implemented using QSS integration methods. Other required models such as adder, multiplier, gain, square-root, step, constant, and switch, are also included.

– ControlledTanks includes a hybrid model of a two-tank system with discrete controller.

– PetriNetsExamples includes the model of an MM1 queue system. This model will be compared with its implementation included in the Extended PetriNets Modelica library [Fabricius and Badreddin, 2002*a*].

5.2.2 Developer's Area

On the contrary, the "developer's area" contains data structures and partial models that the library user does not need to use directly. The documentation of this area is oriented to the library developers.

The "developer's area" is shown in detail in Fig. 5.1c. The *AtomicDEVS* model contains the Modelica implementation of the general behavior of an atomic P-DEVS model. This model is inherited by the *atomicDraft* package of the "user's area". In the AtomicDEVS model, a data structure (i.e., a record) represents the state of the model and Modelica functions describe the P-DEVS functions (i.e., state-initialization, transition, output and time-advance functions).

In addition, the "developer's area" contains the implementation of input and output ports, functions supporting the message passing mechanism needed to communicate P-DEVS models, the implementation of the event-duplicator model (DUP), the model to break algebraic loops (BreakLoop), the implementation of the Select model, the interfaces to combine DEVSLib with other libraries and the QSS integration methods.

The rest of this chapter is devoted to describe the implementation details of these models, used to describe the P-DEVS behavior in Modelica.

5.3 Atomic P-DEVS Models in DEVSLib

This section describes the implementation of a general atomic P-DEVS model in DEVSLib. DEVSLib includes an abstract model, named *AtomicDEVS*, that implements the basic behavior for the atomic P-DEVS model.

The AtomicDEVS model performs the detection of the internal, external and confluent events, the compilation of the input messages in a bag, the generation of the bag of output messages and the management of the actions described using the transition functions. DEVSLib has been developed using Dymola. The description of the behavior of DEVSLib models has been performed using the mechanisms to manage the simulation time and events included in Dymola. Only the triggering conditions for time events (usually internal events where $t_{nextInt} = t + \sigma$), the management of the messages between models (which are transmitted using the previously described mechanism), and the occurrence of simultaneous events needed to be taken into account for the development of the library.

The AtomicDEVS model also allows continuous-time inputs for the transition functions. The value of the continuous-time signal connected to one of these inputs is used as an input for the transition function, and can affect the behavior of the model. No sampling or any other discretization mechanism is necessary. This behavior is similar to the definition of the transition functions in the DEV&DESS formalism [Zeigler et al., 2000]. Thus, the specification of the models supported by DEVSLib is described with the tuple (notice the extra X_{cont} value, included to describe the mentioned continuous-time inputs for the transition functions, in comparison with the tuple described in Zeigler et al. [2000]) $M = (X_M, X_{cont}, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$, where:

$$X_M = \{(p, v) | p \in IPorts, v \in X_p\}$$

$$X_{cont} = \{x_c | x_c \in \Re\}$$

$$Y_M = \{(p, v) | p \in OPorts, v \in Y_p\}$$

$$\delta_{int} : S \times X_{cont} \longrightarrow S$$

$$\delta_{ext} : Q \times X_M^b \times X_{cont} \longrightarrow S \qquad \text{where}$$

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the *total state* set and

e is the *time elapsed* since the last transition.

$$\delta_{con} : Q \times X_M^b \times X_{cont} \longrightarrow S$$

$$\lambda : S \longrightarrow Y_M^b$$

$$ta : S \longrightarrow \Re_{0,\infty}^+$$

X_M is the set of *input ports and values*, X_{cont} is the set of *continuous-time input values*, S is the set of *sequential states*, Y_M is the set of *output ports and values*, δ_{int} is the *internal transition* function, δ_{ext} is the *external transition* function, δ_{con} is the *confluent transition* function, λ is the *output* function and ta is the *time advance* function.

5.3.1 Components of the AtomicDEVS Model

The AtomicDEVS model has been described as a Modelica *partial* model that contains the basic components of an atomic P-DEVS model. It is composed of:

- The *state*.
- The *interface* to connect with other models.
- Some *functions* that describe its behavior.

The definition of these components in Modelica is shown in Listing 5.1. These components are defined as private (or protected), except the state, to indicate that these components belong to the AtomicDEVS model and cannot be used by other models.

5.3.2 Definition of the State and its Initialization

The state is described using a Modelica record, named *S*. The class of *S* is defined as a replaceable record, named *State*, in order to allow the re-definition of the state. This allows to define the state of the model based on the behavior to represent, including the desired variables in the re-defined *State* record. The *State* has to be always described as a Modelica record.

The initialization of the state is performed using the *initState* function. This function has also been defined as replaceable to allow its re-definition depending on the behavior of the model.

```
  partial model AtomicDEVS "Partial atomic DEVS model"
    replaceable record State = stdState;
    State S "Current State";
    parameter Integer numIn = 1 "Num of input ports";
    parameter Integer numOut = 1 "Num of output ports";
  protected
    // connection to input ports
    Real iEvent[numIn];
    Integer iQueue[numIn];
     // connection to output ports
    Real oEvent[numOut];
    Integer oQueue[numOut];
    replaceable function Fout "Output Function"
      input State s;
      input Integer queue[nports];
      input Integer nports;
    end Fout;
    replaceable function Fcon "Confluent Transtition Function"
      input State s;
      input Real e;
      input Integer bag;
      output State sout;
    algorithm
      sout := s;
    end Fcon;
    replaceable function Fint "Internal Transition Function"
      input State s;
      output State sout;
    algorithm
      sout := s;
    end Fint;
    replaceable function Fext "External Transition Function"
      input State s;
      input Real e;
      input Integer bag;
      output State sout;
    algorithm
      sout := s;
    end Fext;
    replaceable function Fta "Time advance Function"
      input State s;
      output Real sigma;
    algorithm
      sigma := Modelica.Constants.inf;
    end Fta;
    replaceable function initState "Initial State Function"
      output State out;
    end initState;
  ... // rest of the code removed
```

Listing 5.1: AtomicDEVS model components.

5.3.3 Interface of the AtomicDEVS Model

The interface of the model is composed of the input and output ports, and four array variables that relate these ports with the rest of the AtomicDEVS model. The ports correspond to the connectors described in Section 4.7. The arrays, named *iEvent*, *iQueue*, *oEvent* and *oQueue*, are required in order to allow a variable number of ports in the interface, without modifying the internal implementation for each different model. The number of input and output ports has to be specified using the parameters *numIn* and *numOut*. The values of these parameters are used to define the size of the mentioned arrays. The values of the "event" and "queue" variables of each port have to be assigned to a position in each array. The "event" variables of input ports are assigned to the iEvent array. The "queue" variables of input ports are assigned to the iQueue array. The "event" variables of output ports are assigned to the oEvent array. The "queue" variables of output ports are assigned to the oQueue array. An example of these assignments, using two input and one output ports, is shown in Listing 5.2 (notice the different indexes used for the in1 and in2 ports in the assignments to the iEvent and iQueue arrays).

```
    parameter Integer numIn = 2 "number of input ports";
    parameter Integer numOut = 1 "number of output ports";
    inPort in1 "first input port";
    inPort in2 "second input port";
    outPort out1 "first output port";
  equations
    in1.event = iEvent[1];
    in1.queue = iQueue[1];
    in2.event = iEvent[2];
    in2.queue = iQueue[2];
    out1.event = oQueue[1];
    out1.queue = oEvent[1];
```

Listing 5.2: Port to array assignments in the AtomicDEVS model.

5.3.4 Definition of the Transition, Output and Time Advance Functions

The internal transition (*Fint*), external transition (*Fext*), confluent transition (*Fcon*), output (*Fout*) and time advance (*Fta*) functions are also defined as replaceable in order to allow their re-declaration. The modeler can express the desired model behavior by re-defining these functions. The re-definition of these functions has to be performed following the function interfaces shown in Listing 5.1. The inputs of each function have to be maintained in the re-definition. Other inputs can be added to the re-defined functions (e.g., to represent the previously mentioned continuous-time inputs for the transitions).

5.3.5 Event Detection and Execution of Transitions

The event detection and transition execution process performed by the AtomicDEVS model is shown in Fig. 5.2. The AtomicDEVS model triggers an external event when the *event* variable of any of the input ports (checking the assigned *iEvent[i]*) changes its value ($iEvent[i] <> pre(iEvent[i])$). Notice that the num-

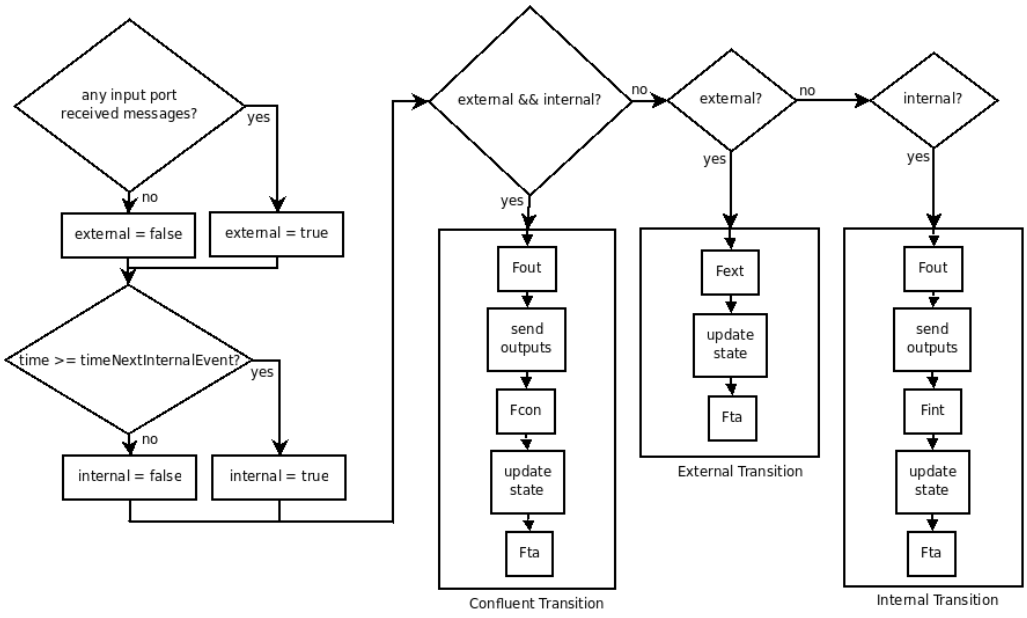


Figure 5.2: Event detection and transition execution diagram of the AtomicDEVS model.

ber of input ports is defined by the modeler, and thus a condition must be set for each port separately. A variable, named *previousInternalTransition*, is used to store the time of the previous transition and calculate the elapsed time. Internal events are triggered when the simulation time reaches the scheduled time for the next internal transition ($time >= pre(nextInternalTransition)$). The value of the variable *nextInternalTransition* is updated after each transition with the value $time + Fta(S)$. Confluent events are triggered with the simultaneous occurrence of both situations. Mutually exclusive boolean conditions decide which transition should be executed at each event.

The boolean variables that define which transition to execute at each event are checked using "when" statements (e.g., *when internalEvent then...*). The three when statements (for the internal, external and confluent transitions) are defined inside an algorithm section, because the same variables are modified inside each when statement. The following actions are performed inside each statement.

During an external transition, the AtomicDEVS model updates the value of the variable that stores the elapsed time, compiles the received messages in the *bag* and executes the *Fext* function (δ_{ext}). The bag of messages is defined as a list of messages, similarly to the queue variable of the input ports. The received messages are extracted from the input ports and put into the bag before executing the external transition function. The external transition function receives as parameters (or inputs) the current state, the elapsed time and the reference to the bag of received messages. The *Fext* function has to manage the received messages and return the future state for the model. After that, the state of the model is updated using the output of *Fext*.

During internal transitions, the AtomicDEVS model executes the output function *Fout* (λ) using the current state. After that, it executes the *Fint* function (δ_{int}) that returns the future state of the model. The state is updated using the output of *Fint* .

During the execution of *Fout*, output messages are sent using the function *sendEvent()*. This function calls the external function *QAdd* to insert the new message into the queue of an output port of the model. The AtomicDEVS model

checks the queues of the output ports (represented by the array *oQueue[i])* to find if any message has been sent through them during the execution of *Fout*. If any message has been sent, the AtomicDEVS notifies the transmission of the message by increasing the value of the event variable of each port (*oEvent[i]*) by the number of messages sent through it. The queues of two connected ports (input and output) are represented by the same data structure in dynamic memory, so each message inserted in the queue of an output port is available (i.e., is received) at the queue of the input port.

In a confluent transition, the AtomicDEVS model generates an output executing the *Fout* function and the current state. After that, it updates the variable that stores the elapsed time, executes the *Fcon* function (δ_{con}), and updates the state of the model with its output.

A new internal transition is scheduled using the *Fta* function (ta) after each transition. The *Fta* function can return any possitive real number, including zero. Immdiate internal transitions and infinite delays can be modeled returning zero and infinite values, respectively.

5.4 Coupled P-DEVS Models in DEVSLib

Coupled DEVSLib models are described following their P-DEVS specification. A coupled model is composed of:

- *Interface*, that allows the connection of the coupled model with another models.
- *Internal components*, which are a combination of atomic or coupled models.
- *Coupling connections* between the interface and the internal components, and between internal components themselves.

The interface of a DEVSLib coupled model is defined using the described input and output ports. The internal components of a DEVSLib coupled model are defined instantiating objects from other already available atomic or coupled DEVSLib models. Since the message passing mechanism used to communicate

DEVSLib models is transparent to the user, the coupling connections between ports and components are defined using Modelica connect sentences between input and output ports.

5.5 Additional Characteristics Included in DEVSLib

The following additional characteristics have been included in DEVSLib to improve the construction of coupled models:

- The first characteristic regards the simultaneous connections between the output port of one model with multiple input ports (i.e., 1-to-many connections). This problem has been described in Chapter 4. DEVSLib includes a model, named *DUP*, to reproduce 1-to-many connections. The DUP model contains one input port, used to receive messages, and two input ports, used to send copies of the received message to multiple receivers. The DUP3 model is similar to the DUP model, but has three output ports. Also, several DUP model can be serially connected if more that three copies of the message are required.

- The second characteristic regards the generation of algebraic loops while connecting model components. An algebraic loop is generated when the output of a model is connected to the input of another model, directly or indirectly connected to the former, creating a loop between both models. Due to Modelica follows the synchronous data flow principle, this situation can not be solved automatically by the simulator (it can not find the correct causality assignment for the models in the loop) and produces an error. Similarly to the previous case, DEVSLib includes a model, named BreakLoop, aimed to avoid this situation. The BreakLoop model defines the causality and breaks the algebraic loop by inserting a *pre()* operator in the detection of its external events.

- The third characteristic is the possibility to connect the output of a model to the input of the same model (i.e., self-connections). This behavior is not

allowed in P-DEVS, but can not be restricted in the Modelica environment. The modeler has to describe the model avoiding this type of connections.

5.6 Conclusions

A new Modelica library, named DEVSLib, has been developed to support the P-DEVS formalism. It includes functionalities to describe atomic and coupled P-DEVS models, and combine them with models developed using other Modelica libraries.

The implementation of the DEVSLib library has been performed as close as possible to the formal specification of models using P-DEVS. A general implementation of the behavior of P-DEVS models has been provided, that can be used to describe multiple types of models. Models are composed of state (including a function for its initialization), interface and transition, output and time advance functions. These elements have been described using the Modelica functionalities for describing object-oriented models, such as the re-declaration of abstract models, replaceable objects and functions, and the functionalities to manage discrete events. The elements of a DEVSLib model can be re-declared in order to describe the desired behavior. The interface is described using input and output ports, and facilitates the description of modular and hierarchical models. The communication between models is performed using the previously described message passing mechanism.

DEVSLib also includes some functionalities to perform 1-to-many connections, not supported by the described message passing mechanism, and the BreakLoop model that can be used to break algebraic loops when constructing coupled models.

6

Construction of Discrete-Event Models Using DEVSLib

6.1 Introduction

The architecture and implementation of the DEVSLib library were presented in the previous chapter. On the other hand, this chapter is devoted to present the use of the library to construct discrete-event models. This is performed from the point of view of a modeler, who wants to construct new discrete-event models using DEVSLib.

The functionalities of the library are presented by means of several case studies. The description of the construction of new atomic models is performed with the implementation of the Processor model described in Zeigler et al. [2000]. This model represents a single-server-with-queue system, that receives jobs and processes them. The construction of new coupled models is also presented using a simple example. Two additional and more complex models are also discussed. The first represents an automatic teller machine that has been described with a pure discrete-event model using the P-DEVS formalism. The second consists in a model of the predator-prey interactions, described by Lotka and Volterra. This is a continuous-time system described using a discrete-event model using the implementation of the QSS integration methods included in DEVSLib. As it will be demonstrated in this and the next chapters, DEVSLib can be used to model discrete-event, continuous-time and hybrid systems.

6.2 Construction of New Atomic Models

The description of atomic models using DEVSLib follows its formal P-DEVS specification. The user defines the variables that represent the state and their initialization. Also, the user has to define the actions performed by the transition functions, in order to update the state of the model after an event, as well as the time advance and output functions.

The construction of a new atomic model starts with the duplication of the atomicDraft model (shown in Fig. 5.1a), which is used as an skeleton for the new model. The atomicDraft model contains:

- A model named *atomic* that extends the AtomicDEVS model and represents the new model.
- A record, named *st*, that represents the new state for the model.
- A function, named *initst*, used to initialize the state.
- Other functions, named *con*, *int*, *ext*, *out* and *ta*, that represent the confluent transition, internal transition, external transition, output and time advance functions, respectively.

The steps required to develop a new model are the following:

1. *Define the interface of the model*, including the required input and output ports, as instances of the DEVSLib "inPort" and "outPort" connectors, into the *atomic* model and setting the value of the *numIn* and *numOut* parameters to the number of included input and output ports. The atomicDraft model includes by default one input and one output ports. The variables (event and queue) of the included ports have to be assigned to the *iEvent*, *iQueue*, *oEvent* and *oQueue* arrays of the AtomicDEVS model, in order to allow the correct reception and transmission of messages.
2. *Redefine the state*, including in the *st* record the required variables to describe the state of the new model (i.e., number of customers in queue,

processing units, etc.). By default, the atomicDraft model includes two variables, *phase* (used to represent the current phase of the model) and *sigma* (used to schedule the next internal event).

3. *Redefine the initialization of the defined state*, including in the *initst* function the initial values for the variables in *st*. The *initst* function receives the initial values for the variables in the *st* record as inputs and returns the initialized *st* record.

4. *Redefine the transition functions*, including in the *ext*, *int* and *con* functions the Modelica code that describes the actions performed during transitions. By default, the confluent transition function (*con*) is defined using the *ext* and *int* functions, executing the internal transition before the external transition. The default behavior of the internal and external transition functions is to return the state previous to the execution of the transition (i.e., the state is not modified).

5. *Redefine the output function*, including in the *out* function the Modelica code that generates output messages (i.e., calling the *sendEvent*() function). The default function does not generate any message.

6. *Redefine the time advance function*, modifying the *ta* function to return the time for the next internal transition, depending on the current state. By default, it returns the value of the sigma variable of the state.

6.2.1 Processor Model Constructed Using DEVSLib

The Processor model, described in Zeigler et al. [2000], is an example of basic P-DEVS atomic model. DEVSLib also includes implementations of Passive, Storage, Generator, BinaryCounter, Processor, Ramp, Switch and Pipe models, described by Zeigler et al. [2000]. Another model, called Display, has been implemented to be able to graphically display the message transmission between models, because Dymola does not display changes in variables occurred during zero-time intervals.

The Processor model receives jobs, each one represented by a real number. If the resource is available, it processes the job and otherwise the job is balked. After a pre-defined processing time is elapsed, the process ends, the resource is released and the model is able to receive new jobs. The P-DEVS specification of this model is the following:

$$M = (X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$$

where:

$X_M = \Re$

$S = \{"passive", "active"\} \times \Re_o^+ \times \Re$

$Y_M = \Re$

$\delta_{int}(phase, \sigma, job) = ("passive", \infty, \emptyset)$

$$\delta_{ext}(phase, \sigma, job, e, x) = \begin{cases} ("active", \triangle, x) & \text{if } phase = "passive" \\ ("active", \sigma - e, job) & \text{if } phase = "active" \end{cases}$$

$\delta_{con} = \delta_{ext}(\delta_{int}, 0, x)$

$\lambda("active", \sigma, job) = job$

$ta(phase, \sigma, job) = \sigma$

The steps to build this particular model using DEVSLib are the following:

1. Create the package for the new model duplicating the atomicDraft model. This package will be called “processor”. The “atomic” model inside the new package has also to be renamed (e.g., to “processor”).

2. Declare the *in* and *out* ports, and relate them with the iEvent, iQueue, oEvent and oQueue arrays of the AtomicDEVS model, as shown in Listing 6.1.

```
model processor extends AtomicDEVS(numIn=1,numOut=1,
  redeclare record State = st);
  redeclare function Fcon = con;
  redeclare function Fint = int;
  redeclare function Fext = ext;
```

```
  redeclare function Fta = ta;
  redeclare function initState = initst(dt=processTime);
  parameter Real processTime = 1;
  Interfaces.inPort in1;   // input port
  Interfaces.outPort out1; // output port
equation
  // relation between ports and arrays
  iEvent[1] = in1.event;
  iQueue[1] = in1.queue;
  oEvent[1] = out1.event;
  oQueue[1] = out1.queue;
end processor;
```

Listing 6.1: Modelica code of a processor system modeled using DEVSLib.

3. The state of the model is composed of phase, sigma, job, processing time (dt) and a counter for the received jobs. The declaration of the state (using the *st* record) and the state initialization function are shown in Listing 6.2. Notice that the input of the initst function corresponds to the processTime parameter declared in the processor model (see Listing 6.1).

```
record st "State of the model"
  Integer phase;    // 1 = passive, 2 = active
  Real sigma; // internal transitions interval
  Real job;         // current processing job
  Real dt;          // default processing time
  Integer received; // num of jobs received
end st;

function initst "State Initialization Function"
  input Real dt;
  output st out;
algorithm
  out.phase := 1; // passive
  out.sigma := Modelica.Constants.inf; // waiting
  out.job := 0;
  out.dt := dt;
  out.received := 0;
end initst;
```

Listing 6.2: Modelica code of the state and the state initialization function of the processor model.

4. Redeclare the external, internal and confluent transition functions to reproduce the behavior of the model (described above with its formal specification), as shown in Listing 6.3.

```
function con "Confluent Transition Function"
  input st s;
  input Real e;
  input Integer bag;
  output st sout;
algorithm
```

```
    sout := ext(int(s),0,bag); // first internal transition
  end con;

  function int "Internal Transition Function"
    input st s;
    output st sout;
  algorithm
    sout := s;
    sout.phase := 1; // passive
    sout.sigma := Modelica.Constants.inf;
    sout.job := 0;
  end int;

  function ext "External Transition Function"
    input st s;
    input Real e;
    input Integer bag;
    output st sout;
  protected
    Integer numreceived;
    stdEvent x;
  algorithm
    sout := s;
    numreceived := numEvents(bag);
    if s.phase == 1 then
      for i in 1:numreceived loop
        x := getEvent(bag);
        if i == 1 then
          sout.job := x.Value;
          Modelica.Utilities.Streams.print("* Msg to process");
        else
          Modelica.Utilities.Streams.print("* Msg balked");
        end if;
        sout.received := sout.received +1;
      end for;
      sout.phase := 2; // active
      sout.sigma := s.dt; // processing_time
    else
      sout.sigma := s.sigma - e;
    end if;
  end ext;
```

Listing 6.3: Modelica code of the transition functions redeclared in the processor model.

5. Also, redeclare the output and time advance functions, as shown in Listing 6.4.

```
  function out "Output Function"
    input st s;
    input Integer queue[nports];
    input Integer nports;
  protected
    stdEvent y;
  algorithm
    if s.phase == 2 then
      y.Type := 1;
      y.Value := s.job;
      sendEvent(queue[1],y);
    end if;
  end out;
```

```
function ta "Time Advance Function"
  input st s;
  output Real sigma;
algorithm
  sigma := s.sigma;
end ta;
```

Listing 6.4: Modelica code of the output and time advance functions redeclared in the processor model.

6.3 Construction of Coupled P-DEVS Models

The *coupledDraft* model included in DEVSLib provides a simple way to start the development of new coupled DEVSLib models. It can be duplicated and the copy adapted to the behavior of a new coupled model. New input and output ports can be included, by inserting new instances the DEVSLib "inPort" and "outPort" connectors. The components of the model can be included in the same fashion, instantiating the required components that have been previously developed. The coupling connections are defined by including Modelica "connect" sentences between input and output ports, either between the interface and the internal components, or among the internal components themselves. Dymola offers functionalities to perform these procedures, using drag and drop, and graphically defined connections between ports.

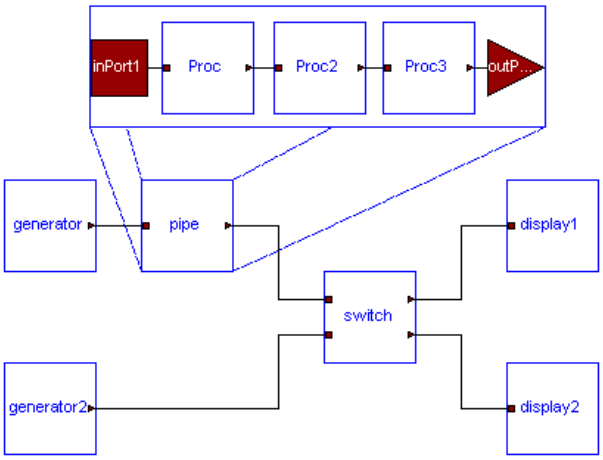


Figure 6.1: Simple coupled P-DEVS model constructed using DEVSLib.

An example of a coupled model constructed using DEVSLib is shown in Fig. 6.1. The components of this model are the Generator, Pipe and Switch models described in Zeigler et al. [2000]. The Generator is used as a source of messages. It periodically generates new messages and sends them through its output port. The Pipe represents a coupled model with three processors. It has been constructed connecting three Processor models (described in the previous section) in series. The Switch model sends the messages received in the first input port through the second output port, and vice-versa. The Display models have been included to observe the outputs of the Switch model.

6.4 Modeling an Automatic Teller Machine

The description of a discrete-event model using DEVSLib is presented in this section. The model represents an ATM system (Automatic Teller Machine) that is composed of a card reader, an operation authorization subsystem and the cash dispenser. The behavior of the system is described in the state diagram shown in Fig. 6.2. The DEVS specification of the system can be found in Saadawi [2004].

The behavior of the system is as follows. The user inserts a card in the ATM. The system recognizes the new insertion and asks the user to introduce his PIN number. In case of a failed PIN number, the system ask the user to enter the correct PIN. If the user fails three times inserting the PIN, the system ejects the card. When the correct PIN is inserted, the system asks the user to enter the amount of cash to withdraw. If the balance in the account of the user is not correct, the system asks the user for a new amount. When the balance is correct, the system gives the cash to the user and ejects the card. While the system is busy, any new card insertion is neglected.

The ATM system constructed using DEVSLib is shown in Fig. 6.3 (notice the required *DUP* and *BreakLoop* models). The BreakLoop models are required to define the causality in the loops. The card reader and the cash dispenser are simple atomic DEVSLib models. They are modeled following the procedure described in Section 6.2. The operation authorization mechanism is modeled using

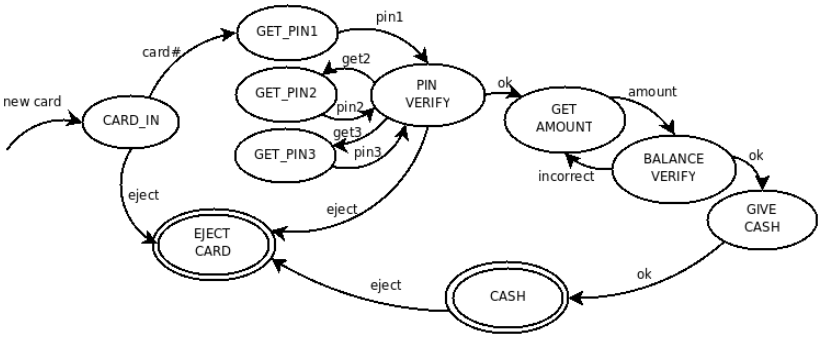


Figure 6.2: State diagram of the ATM system (the system generates outputs at encircled states).

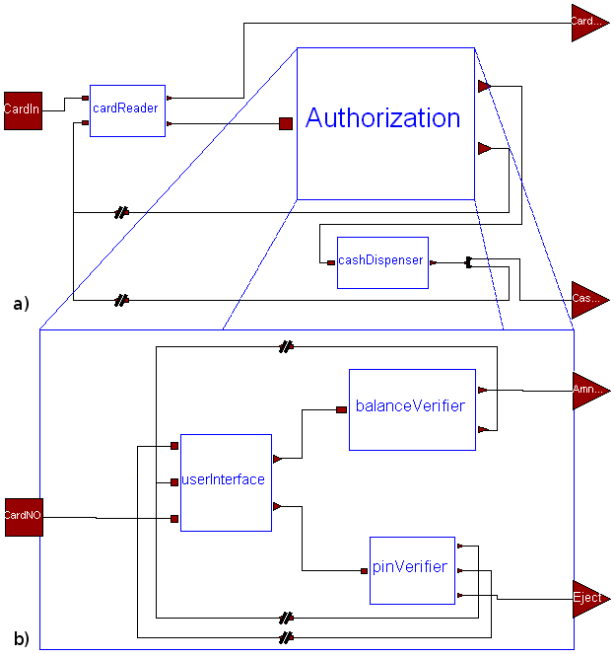


Figure 6.3: ATM system modeled using DEVSLib: a) top-level components and; b) authorization subsystem.

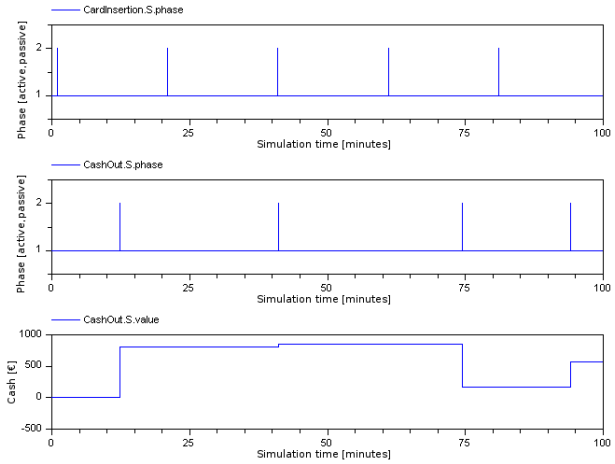


Figure 6.4: Simulation results for the DEVSLib ATM model, obtained using Dymola.

a coupled model, as shown in Fig. 6.3. It is composed by three atomic models: the user interface, the balance verifier and the PIN verifier. The interactions between the system and the user, in order to obtain the PIN number and the amount of cash, have been modeled statistically generating the data from random uniform distributions. The correctness of the PIN number and the balance in the user account have also been modeled using uniform random numbers.

The correspondence between the model shown in Fig. 6.3 and the diagram shown in Fig. 6.2 is as follows. The card reader performs the CARD_IN action. The GET_PIN and the GET_AMOUNT actions are performed by the user interface. The PIN_VERIFY and the BALANCE_VERIFY actions are performed by the pin verifier and balance verifier models, respectively. Finally, the GIVE_CASH action is performed by the cash dispenser. The CASH and EJECT_CARD outputs represent the output messages that arrive to the output ports in Fig. 6.3a.

The simulation results are shown in Fig. 6.4. The card insertions are shown at the top, in the middle the end time of the operations, and below the amount of cash withdrawn by the user in each operation. It can be noticed that since the insertions of the card are modeled at a constant rate, some of the insertions (in this case the third one) are neglected because the system is still busy with the previous insertion.

6.5 Quantized State Systems in DEVSLib

As DEVSLib has been designed for describing general P-DEVS models, the description of continuous-time models using the Quantized State System (QSS) methods is supported by DEVSLib. This section presents the implementation of the QSS1, QSS2 and QSS3 integration methods as atomic DEVSLib models and its application to the simulation of the Lotka-Volterra model.

Quantized State Systems (QSS) are continuous-time systems whose state trajectories are converted into piecewise constant functions using a quantization function with hysteresis [Kofman and Junco, 2001]. These systems can be de-

scribed as discrete-event systems using the DEVS formalism. A change in the input of the system is represented as an event, that generates a new output. Events represent changes in the state trajectories. The hysteresis is used to avoid problems with infinite number of events and define legitimate DEVS models [Zeigler et al., 2000]. QSS has also been extended for simulating hybrid systems, combining QSS models of continuous-time systems with other discrete-event DEVS models [Kofman, 2004].

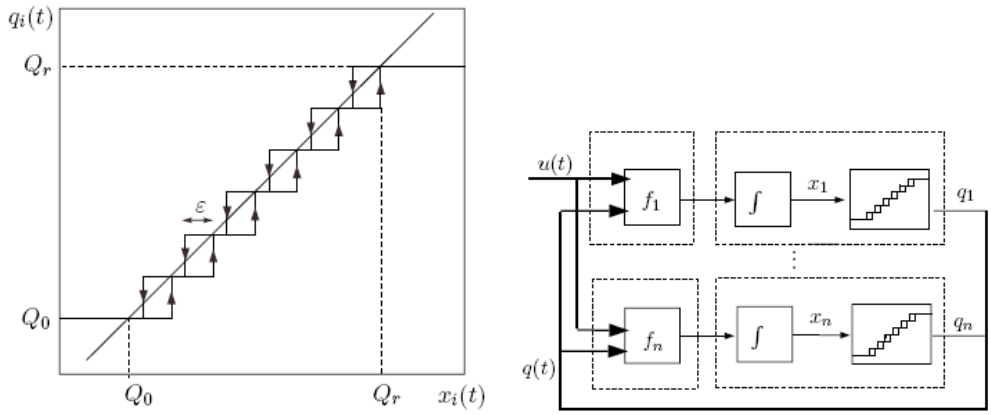
A QSS system can be defined as follows. Having the following system:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ y(t) &= g(x(t), u(t)) \end{aligned} \tag{6.1}$$

Its associated quantized state system is defined as:

$$\begin{aligned} \dot{x}(t) &= f(q(t), u(t)) \\ y(t) &= g(q(t), u(t)) \end{aligned} \tag{6.2}$$

where $q(t)$ and $x(t)$ are related by quantization functions with hysteresis. The behavior of a quantization function with hysteresis is shown in Fig. 6.5a.



(a) (b)

Figure 6.5: a) Quantization function with hysteresis and; b) block diagram of a QSS system [Kofman and Junco, 2001].

6.5.1 QSS Methods in DEVSLib

DEVSLib includes atomic models for the QSS first (QSS1), second (QSS2) and third (QSS3) order integrators. These integrators can be used in combination with other atomic models (i.e., Step, Square, Constant, Add, Gain, Multiplier and Switch, also included in DEVSLib), to describe continuous-time systems in a block-diagram fashion.

The development of the QSS integrators has been performed following the procedure described in Section 6.2. Each integrator has one input and one output ports. The external events represent a change in the first derivative of the state variable. The external transition function updates the value of the state variable using the new derivative and schedules a new internal event. The scheduled internal event represents the point in time when the value of the state will change by the predefined quantum (i.e., the next step of the quantization function shown in Fig. 6.5a). At internal events, the current value of the state variable is sent as an output and a new internal event is scheduled (representing the next quantum-variation of the state variable).

The content of the messages between QSS models in DEVSLib is as follows:

- The first order algorithm only uses the value of the variable to perform the calculations, so one message of the default type (i.e., including Type and Value variables) can transport this information.

- The second order algorithm uses the value of the variable and its first derivative. In this case, two messages are used to transport this information. These messages are transmitted simultaneously and are identified using the Type variable (e.g., "Type == 1" represents the value, and "Type == 2" represents the first derivative).

- The third order algorithm uses the value, the first and the second derivative. Thus, an additional third message is simultaneously transmitted to transport the value of the second derivative. This message is identified

with the "Type == 3". The external transition functions are programmed to recognize the type of each message and use their values as required.

6.5.2 Lotka-Volterra System

The Lotka-Volterra model of the predator-prey interaction [Lotka, 1925; Volterra, 1931] is used to illustrate the continuous-time system modeling with DEVSLib. The equations of the Lotka-Volterra model are the following:

$$\frac{dx}{dt} = x\alpha - xy\beta$$
$$\frac{dy}{dt} = -y\gamma + xy\delta$$

where y is the number of predators, x is the number of preys, and α, β, γ and δ are parameters that represent the interaction between both species (in this case study the value $\alpha = \beta = \gamma = \delta = 0.1$ has been used). The predator and the prey populations are inversely related: the growth of one of the species reduces the growth rate of the other, and vice-versa. The result is an oscillatory behavior in the population of both species.

The model described using DEVSLib QSS algorithms is shown in Fig. 6.6, using the first order integrator (QSS1). The QSS1 model could be substituted with either the QSS2 or QSS3 models to apply other integrator. The multiplier and adder modules are also DEVSLib atomic models. Three DUP models are

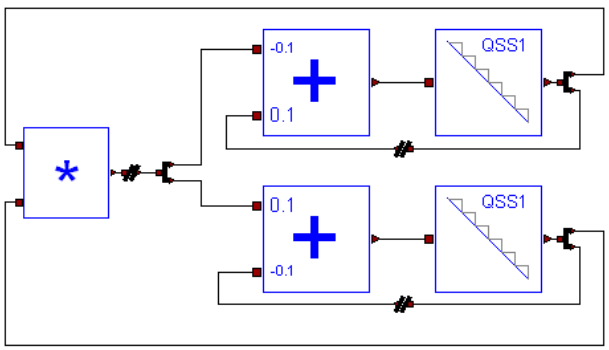


Figure 6.6: Lotka-Volterra model composed using DEVSLib.

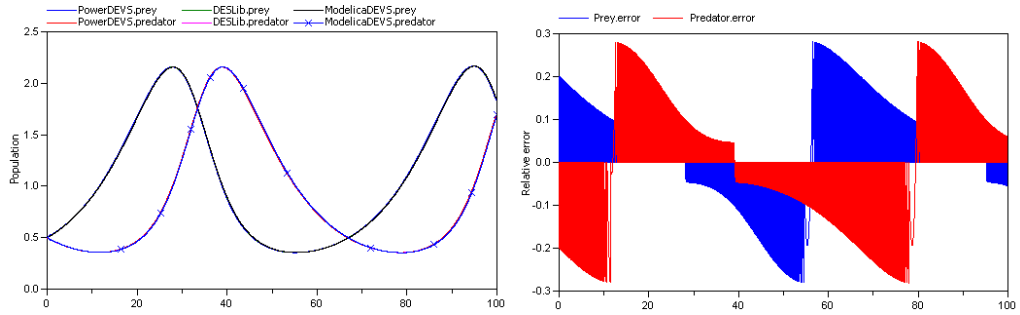
required to divide the flow of messages at the output of the integrators and the multiplier. Also, three BreakLoop models are included to break the algebraic loops between the adder, the multiplier and the integrators.

In order to compare the simulation results and performance, the Lotka-Volterra model has also been developed using the PowerDEVS software tool and the ModelicaDEVS library. The simulation results obtained by using DEVSLib, PowerDEVS and ModelicaDEVS are shown in Fig. 6.7. The model has been simulated using the QSS1 (Fig. 6.7a), QSS2 (Fig. 6.7b) and QSS3 (Fig. 6.7c) methods. The results using QSS1 in the three implementations almost overlap (see the left side of Fig. 6.7a). The results using QSS2 and QSS3 in the PowerDEVS and DEVSLib models are also very similar (see the left side of Figs. 6.7b and 6.7c). The results obtained with the ModelicaDEVS model using QSS2 and QSS3 are different from the other models. The most likely cause of these differences is a programming error in the ModelicaDEVS integrator, which has not been detected in previous evaluations using other models.

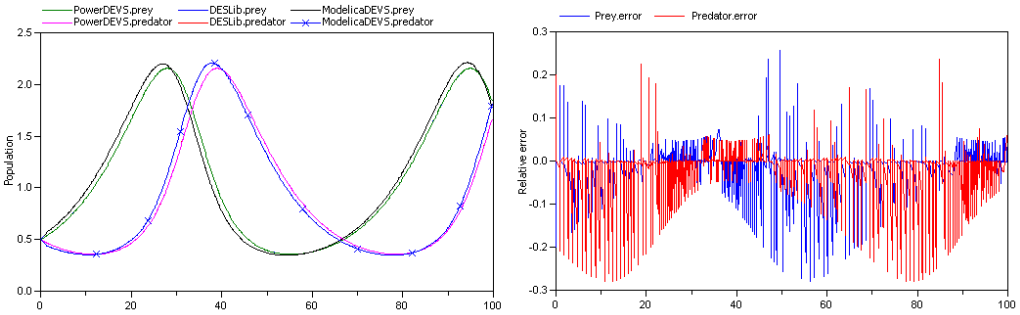
The relative errors, in percentages, between the DEVSLib and the PowerDEVS models are shown at the right side of Figs. 6.7a, 6.7b and 6.7c. The errors show the differences between the outputs of each integrator (i.e., QSS1, QSS2 and QSS3, for predators and preys). These differences remain similar when increasing the order of the integrator. The differences concerning the ModelicaDEVS implementation have not been calculated due to the aforementioned error in the implementation.

The simulation performance of the Lotka-Volterra model, using QSS1, QSS2 and QSS3, has been compared. The obtained results are shown in Table 6.1. The performance indicators are the mean execution time, calculated from six simulation runs, and the number of events. The simulated time is 100 seconds.

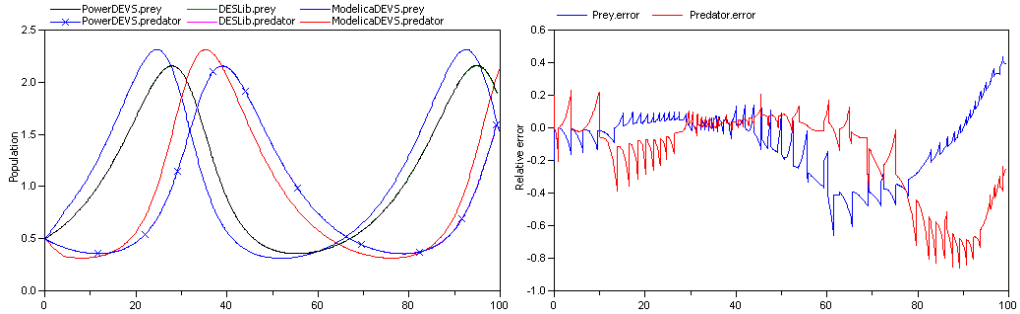
The best performance is obtained using PowerDEVS, as also stated in the comparison performed in Beltrame and Cellier [2006], because it is designed for simulating discrete-event systems following the DEVS simulator described in Zeigler et al. [2000]. Dymola is designed to efficiently simulate continuous-time systems, and includes algorithms to detect and treat discrete-events. This leads to



(a)



(b)



(c)

Figure 6.7: Simulation of the Lotka-Volterra model developed using DEVSLib, PowerDEVS and ModelicaDEVS (relative errors between the PowerDEVS and DEVSLib models at the right). Integration method: a) QSS1; b) QSS2; and c) QSS3.

a robust hybrid system simulation approach. However, these algorithms unnecessarily degrade the performance while simulating pure discrete-event systems.

ModelicaDEVS has been specifically designed for modeling of continuous-time systems using the QSS integration methods. In contrast, DEVSLib has been designed to support the P-DEVS formalism and the QSS methods have been developed by applying the facilities provided by DEVSLib to describe general-purpose atomic P-DEVS models. As observed in the simulation results, the performance of both libraries is similar.

		QSS1	QSS2	QSS3
DEVSLib	Execution Time (s)	2.19	0.078	0.031
	Number of Events	17366	509	153
PowerDEVS	Execution Time (s)	1.19	0.022	0.0047
	Number of Events	5238	172	47
ModelicaDEVS	Execution Time (s)	1.26	0.071	0.047
	Number of Events	15538	490	152

Table 6.1: Comparison of simulation performance based on the Lotka-Volterra model.

6.6 Conclusions

DEVSLib has been designed to describe general discrete-event models following the P-DEVS formalism. It has been demonstrated that the included functionalities can be used to model discrete-event systems, and continuous-time systems using the QSS integration algorithms included in the library.

The construction of new atomic P-DEVS models using DEVSLib is close to their formal specification. New models are implemented describing the elements of the tuple, this is, the state, the interface, and the transition, output, time advance and state initialization functions. The construction of new coupled P-DEVS models using DEVSLib also follows their formal specification. Coupled models are implemented by describing the interface, including the internal components, and describing the coupling relations between them and the interface

ports. Dymola provides drag and drop, and connections drawing functionalities to facilitate this task.

7

Hybrid System Modeling Using DEVSLib

7.1 Introduction

The implementation of the DEVSLib library and its use to construct discrete-event models following the P-DEVS formalism has been presented in the two previous chapters. However, following the main objective of this dissertation, DEVSLib also includes functionalities to describe hybrid systems.

DEVSLib includes interface models that translate the messages into discrete-time signals, and the continuous-time and discrete-time signals into event trajectories (i.e., series of messages). These interfaces can be used to combine P-DEVS models with models from other Modelica libraries. This combination facilitates the description of complex multi-formalism multi-domain hybrid systems.

The interfaces, their implementation and their use are presented in this chapter. Two case studies are described.

- The first case study represents a system with two interconnected tanks controlled using a discrete controller. The tanks and the valves to connect them are described using a continuous-time model. Two approaches to describe the discrete controller using DEVSLib are discussed: using an atomic model or a coupled model. The combination of both models is performed using the interface models included in DEVSLib. The results obtained with the im-

plemented model are compared with the ones obtained using an equivalent model developed using the StateGraphs Modelica library.

– The second case study represents an opto-electrical communication system. The electrical part has been described using the Modelica Standard Library, and the optical part has been described using DEVSLib. The interactions between both parts are also described using the interface models included in DEVSLib. The development of this model has been performed in collaboration with the ARS Lab group from the Carleton University (Canada) [ARS Lab, 2010]. The ARS Lab group developed an equivalent model using their CD++ tool [Wainer, 2002]. The simulation results obtained by both models are compared and discussed.

7.2 Interfaces between DEVSLib and Other Modelica Libraries

This section describes the interfaces included in DEVSLib to facilitate the combination of P-DEVS models with other Modelica libraries. These interfaces are divided in two: the messages-to-signals and the signals-to-messages translations.

These interface models are implemented to manage the standard DEVSLib message type (with Type and Value variables). However, as described in Section 4.6.1, the message type in DEVSLib can be re-defined by the user. In this case, the interface models can be adapted to the new message type. To perform this, the modeler has to:

1. Adapt the interface models to allow the construction of new messages using the new type. The interface models generate messages, and thus they should be able to construct a message with a content valid for the new type. For example, if the new message type includes a name for each message, a procedure to assign names for the new messages needs to be included.

2. Use the appropriate function to send the new messages. Currently, the *sendEvent*() function is used. Depending on the characteristics of the new type of message, this function can be adapted or a new function should be required.

7.2.1 Signals to Messages

The translation of continuous-time and discrete-time signals into messages is performed by the following models:

- *Quantizer*, performs a quantization of the continuous-time (or discrete-time) input signal generating a new message whenever the signal changes its value by a given quantum. The message transports the value of the signal at that time. The condition used to check the variation of the signal(u) is $((u >= preEvent + q + Threshold)\ or\ (u <= preEvent - q - Threshold))$, where $preEvent$ represents the last quantified value of the signal (it is initialized with the initial value of u), q is the value of the quantum, and $Threshold$ is a parameter (set to 10^{-10} by default) used to avoid inconsistencies when the value of the signal reaches (but not crosses) the next quantified value.

- *CrossUP* and *CrossDOWN*, that generate a new message when the value of the continuous-time (or discrete-time) signal crosses a pre-defined threshold in any direction, upwards or downwards. The parameters of both models are: Value, used to check the crossing of the input signal, and EType, used to set the value of the Type variable of the generated message. The condition used to check the crossing in the CrossDOWN model is $((u < Value)\ and\ above)$, where *above* is a boolean variable that indicates if the current value of the input signal is above the $Value$ or not (it is used to detect the direction of the crossing, downwards in this case). The value of the above variable is updated at every simulation step using the condition $((u > Value)\ and\ not\ above)$, that makes above to be set to true. The condition used to check the crossing in the CrossUP is $((u > Value)\ and\ below)$, where *below* is a boolean variable used to detect

the upwards crossing of the *Value*. Also , the value of the below variable is check with the condition ($(u < Value)$ *and not below*), that sets it to true.

– *BCrossUP* and *BCrossDOWN*, that behave similarly to the CrossUP and CrossDOWN models, but in this case their inputs are Boolean. These models generate a new message every time their boolean inputs switch their value. The value of the generated message is 0 for false and 1 for true.

– *CondGen*, that is used to generate messages based on a boolean condition. If the condition becomes true, the model generates a message whose type and value are specified using the model parameters.

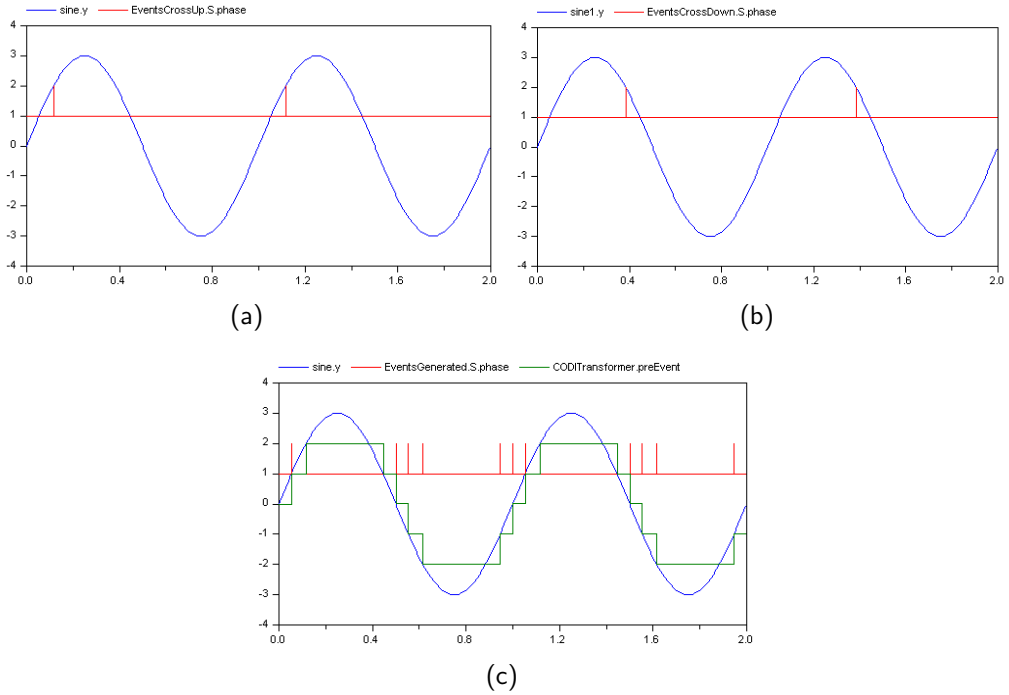


Figure 7.1: Response of DEVSLib signal-to-message interface models: a) CrossUP (value == 2); b) CrossDOWN (value == 2); and c) quantizer (quantum == 1).

The response of the CrossUP, CrossDOWN and Quantizer models when applied to a sinusoid signal is shown in Fig. 7.1. The CrossUP model detects the sinusoid signal crossing the value 2 in upwards direction, and generates a message with that value. The CrossDOWN model detects the sinusoid signal crossing the value 2 in downwards direction, and generates a message with that value. The

Quantizer model generates a message every time the value of the signal changes in a quantity bigger than the defined quantum – i.e., at values 1 and 2, again at 2 and 1, and so on. It has to be noticed that the signal never crosses the values 3 and -3, so messages are not generated at those points.

7.2.2 Messages to Signals

The translation of messages into discrete-time signals is performed by a model named *DICO* (DIscrete-to-COntinuous). The DICO model generates a piecewise-constant continuous signal whose value is equal to the values transported by the received messages. DEVSLib also includes a model, named DIBO (DIscrete-to-BOolean), that generates a boolean signal with value "true" if the value of the arrived message is greater than zero, and "false" otherwise.

7.3 Controlled Tanks System

An example provided in the StateGraph Modelica library [Otter et al., 2005] will be employed to illustrate the use of the DEVSLib interfaces and to compare the performance of these two libraries (i.e., StateGraph and DEVSLib).

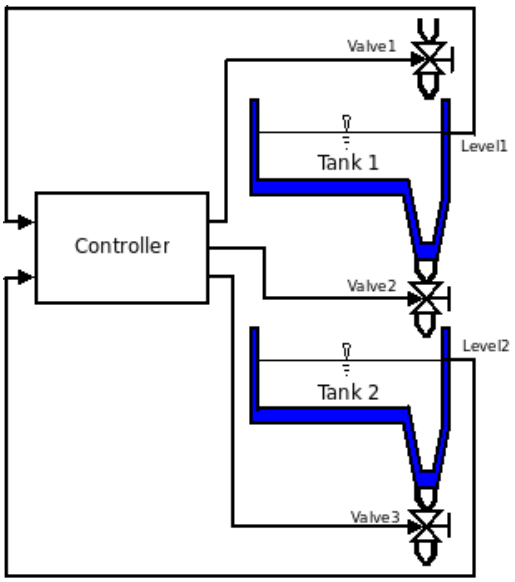


Figure 7.2: Controlled two-tank system.

A diagram of the model is shown in Fig. 7.2. The model consists of two tanks interconnected with valves, which are manipulated by a discrete-event controller. One of the valves is connected to the input flow of the first tank. The output of the first tank is connected to the input of the second tank, with a valve in between to control the flow between both tanks. The third valve is connected to the output of the second tank. The discrete controller receives the level of each tank and controls the positions of the valves (i.e., open/close), in order to fill or empty them.

The normal operation of the system is as follows (summarized in the state diagram shown in Fig. 7.3):

1. Valve 1 is opened and tank 1 is filled (the system changes from IDLE to FILL1 state).

2. When tank 1 reaches its limit, valve 1 is closed (changing from FILL1 to WAIT1).

3. After a waiting time, valve 2 is opened and the fluid flows from tank 1 into tank 2 (changing from WAIT1 to FILL2).

4. When tank 1 reaches its limit, valve 2 is closed (changing from FILL2 to WAIT2).

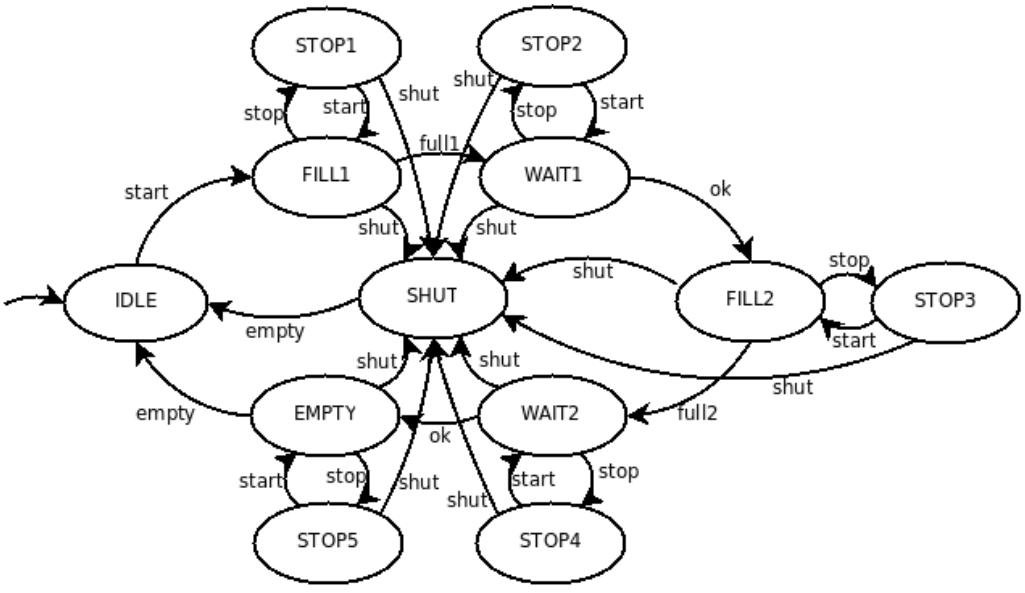


Figure 7.3: State diagram of the controlled two-tank system.

5. After a waiting time, valve 2 is opened and the fluid flows out of tank 2 (changing from WAIT2 to EMPTY).

6. When tank 2 is empty, valve 3 is closed (going back to IDLE again).

Three buttons allow starting, resuming, stopping or aborting the normal operation procedure:

- *Start*, starts the process (leaving the IDLE state). When it is pressed after "stop" or "shut" the process continues (changing the state from STOP to its previous state, or restarting the normal operation procedure, respectively).

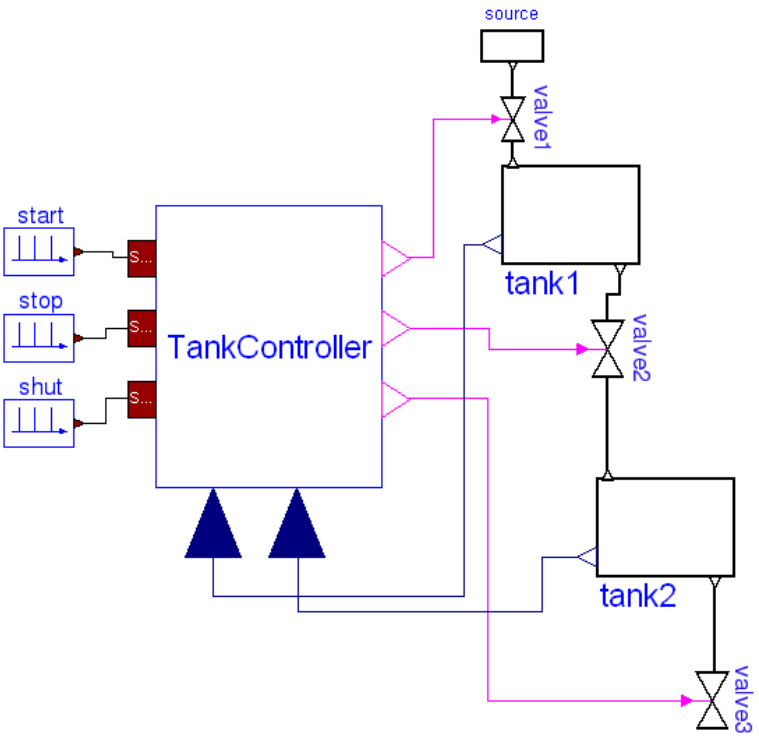
- *Stop*, stops the process by closing all valves (changing to the corresponding STOP state). The controller waits for further input ("start" or "shut").

- *Shut*, is used to shutdown the process, by emptying at once both tanks (changing to the SHUT state, and when empty changing to IDLE). After emptying the system goes to the start configuration and waits.

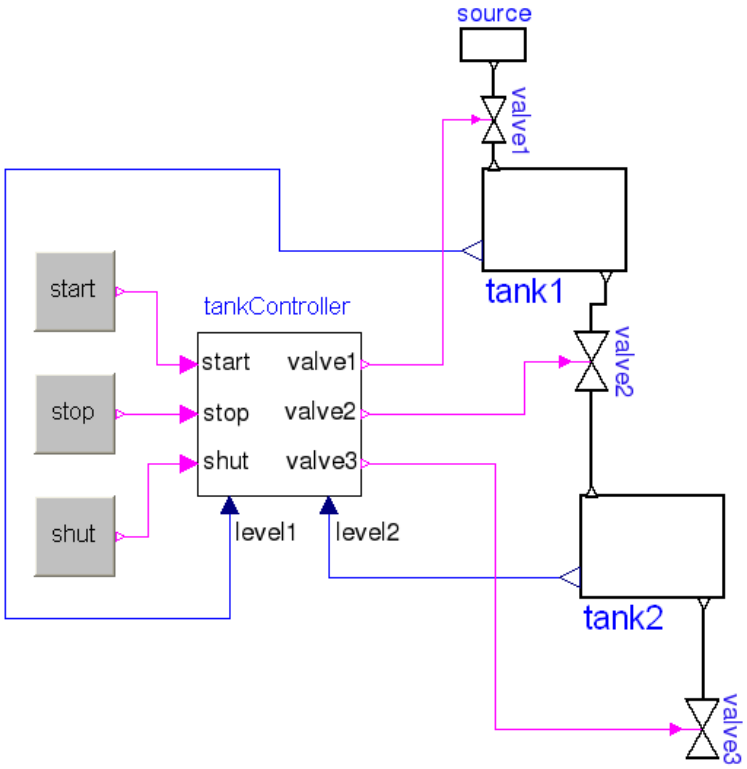
The diagrams of the models developed using DEVSLib and StateGraphs are shown in Fig. 7.4. The continuous-time part (i.e., the tanks and valves) is the same in both models. Its components (source, valves and tanks) were developed using plain Modelica code, and can be later interconnected to describe the structure of the system.

The internal structure of the controller is shown in Fig. 7.5. The StateGraph controller implements the states and the transitions needed to achieve the desired plant operation. The controller implemented with DEVSLib includes the models to translate the continuous-time signals from the tanks, L1 and L2, into trajectories of events. The level of tank 1 is translated with two cross value models, one for detecting the full level (set to 0.98m) and another for the empty level (set to 0.001m). Tank 2 only needs the detection of the empty level. Also, the controller outputs are translated into boolean signals (V1, V2 and V3) that control the state of the valves, using the DIBO models included in DEVSLib.

The controller itself is a P-DEVS coupled model, shown in Fig. 7.6 (all the required *DUP* models have been removed from the figure to facilitate its un-

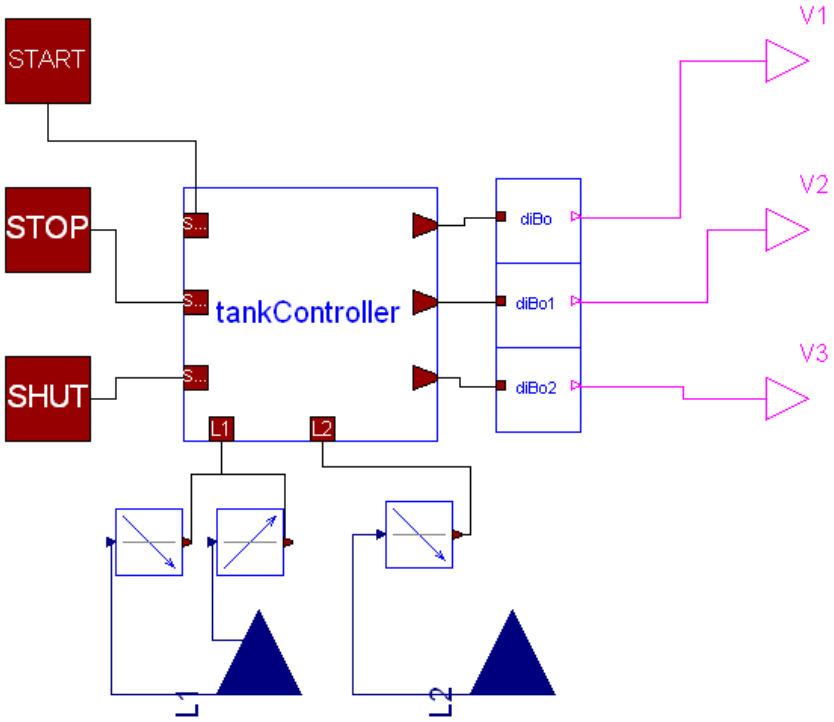


(a)

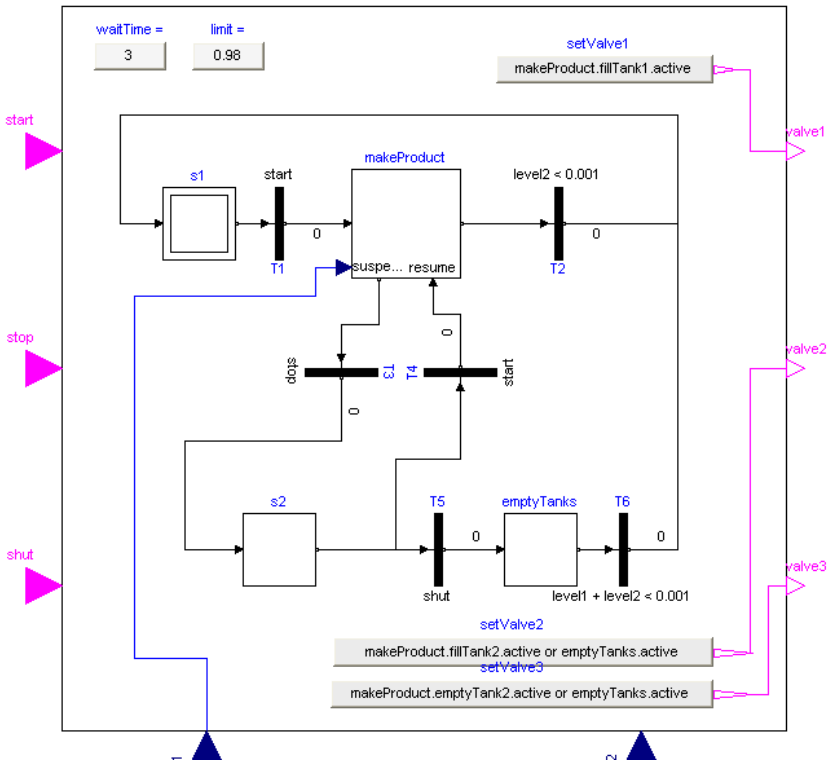


(b)

Figure 7.4: Tank system modeled with: a) DEVSLib and; b) StateGraphs.



(a)



(b)

Figure 7.5: Tank system controller modeled with: a) DEVSLib and; b) StateGraphs.

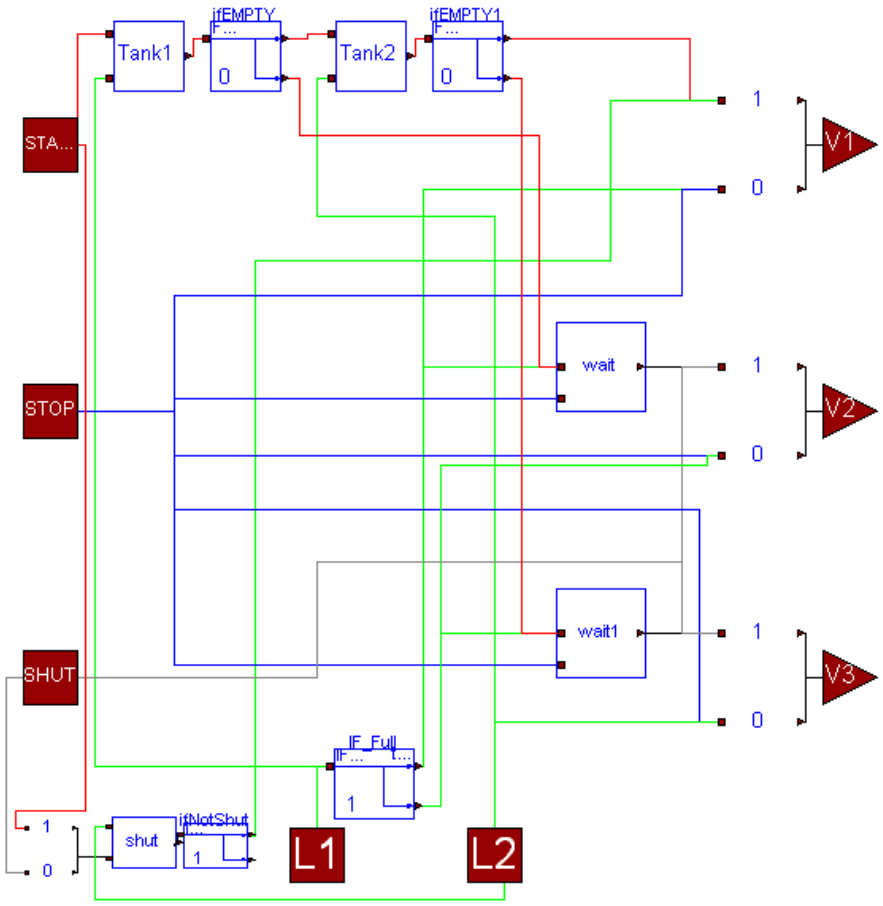


Figure 7.6: Internal structure of the tank controller implemented using DEVSLib.

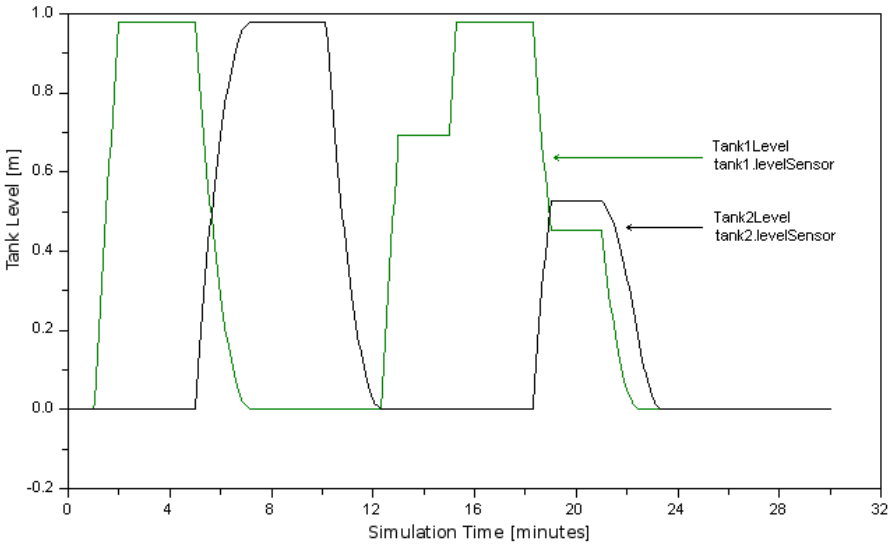


Figure 7.7: Simulation results of the tank filling/emptying system (DEVSLib and StateGraph results overlap).

derstanding), that implements the described logic using small P-DEVS atomic operations included in the library (ifType, storage, setValue, etc.). Also, the DEVSLib controller can be implemented as a P-DEVS atomic model including the control algorithm in the transition functions. The P-DEVS specification of these models is detailed it the documentation of the model included in the library. The simulation results of both models, with the atomic or the coupled controller, are the same (see Fig. 7.7).

The simulation performance of the models composed using DEVSLib and StateGraphs has been evaluated. Two different DEVSLib implementations of the controller have been considered: first implementing the controller as an atomic P-DEVS model and second implementing it as a coupled P-DEVS model. The models have been configured to continue with the normal operation process during the whole simulation time, because the initial configuration stops the normal operation around time 24 minutes. The performance indicators are the mean execution time, calculated from six simulation runs, and the number of events. The simulated time is 1000 minutes. The performance comparison is shown in Table 7.1.

Table 7.1: Performance comparison based on the tank system.

DEVSLib (coupled)	Execution Time [s]	0.313
	Number of Events	170 (time) + 448 (state)
DEVSLib (atomic)	Execution Time [s]	0.078
	Number of Events	168 (time) + 446 (state)
StateGraphs	Execution Time [s]	0.094
	Number of Events	168 (time) + 447 (state)

It can be noticed that the DEVSLib model with the atomic controller and the StateGraph model have similar execution times. The simulation of the coupled DEVSLib controller consumes more time than the simulation of the atomic DEVSLib controller. This difference in performance, even having similar amount of events, is mainly due to the amount of operations performed during each event. The coupled controller activates multiple algorithms while the atomic controller

has only one. However, the coupled DEVSLib controller is easier to understand than the atomic DEVSLib controller.

7.4 Opto-Electrical Communication System

Opto-electrical interfaces, transmitters and receivers, can be used to translate the information contained in form of electrical current into light and viceversa [Agrawal, 1997]. These interfaces constitute the basic components of Optical Networks on Chip (ONoC) [Brière et al., 2004]. A transmitter in such interface receives an electrical current and translates it into optical impulses. The receiver receives the optical impulses and translates them into electrical current again. ONoC interfaces can be used to substitute electrical interconnects between processors in digital integrated circuits [Razavi, 2002].

To facilitate the development of such opto-electrical communication systems, several approaches have been proposed to model and simulate the behavior of the circuits in order to better understand the involved phenomena. Depending on the abstraction level used to describe the system, FEM (Finite Elements Method) and FDTD (Finite-Difference Time-Domain) methods have been used to describe the optical behavior at the physical level [O'Connor, 2004].

On the behavioral and system levels, several authors use VHDL-AMS to describe the components of the system [Mieyeville et al., 2004; Brière et al., 2004; O'Connor, 2004]. Also event-driven approaches have been used to describe components at system level, like SystemC [Brière et al., 2005, 2007; O'Connor et al., 2006] and OMNet++ [Shacham et al., 2007].

A model of basic opto-electrical interfaces constructed using Modelica and DEVSLib is discussed. This model shows the possibility of defining Systems on Chip (SoC), using the P-DEVS formalism, at a higher abstraction level than other existing techniques.

The description of each domain (electrical and optical) using P-DEVS models simplifies the development of multi-domain communication systems. Also, the

description of the electrical part as a continuous-time model can be performed using Modelica, obtaining a more detailed hybrid model.

7.4.1 Communication Between the Opto-Electrical Interfaces

The opto-electrical communication system is composed of a transmitter that sends optical information to the receiver, which recovers it and generates a current. These components are shown in Fig. 7.8.

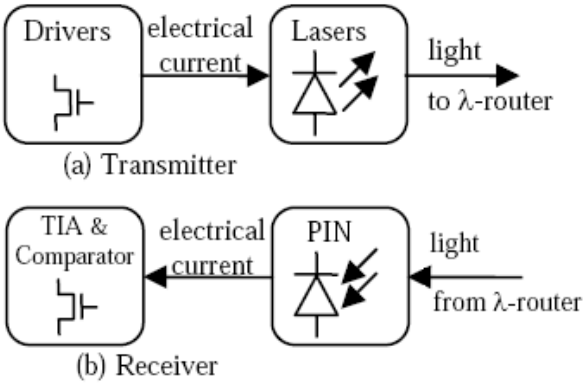


Figure 7.8: Basic opto-electrical interfaces [Biere et al., 2007].

The transmitter is composed of two components: driver and laser. The driver receives external information in form of an electrical current, modulates it, and sends it to the laser. The laser receives the modulated current and generates optical impulses.

The receiver model is also composed of two components: photodiode and transimpedance amplifier (TIA). The photodiode receives the optical impulses and translates them into electrical current. The TIA amplifies the generated current.

The behavior of each of these components can be described using the DEVS formalism as shown in [Biere et al., 2007]. Components are basically described as processors, that receive a message, perform a process to the information contained in the message and, depending on the defined behavior, send an output message containing the information processed. This output will be received as input signal

by the next component. The models developed and discussed in this manuscript are based in the mentioned DEVS formalization.

7.4.2 Modelica/DEVSLib Model

The transmitter and the receiver have been modeled as coupled P-DEVS models constructed using the DEVSLib library. The electrical components have been modeled as continuous-time models, using the electrical components of the Modelica Standard Library. The system modeled with Modelica is shown in Fig. 7.9.

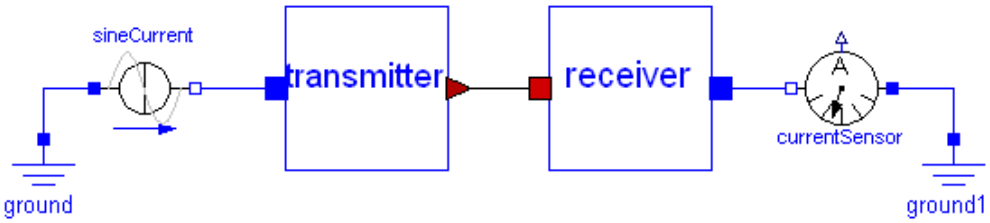


Figure 7.9: Basic opto-electrical communication system modeled using Modelica/DEVSLib.

The transmitter contains the *driver* and the *laser* models, as it is shown in Fig. 7.10. The current received in the transmitter, through the electrical port, is passed to the *driver* model. To simplify the model, the received current is not modified by the *driver*. This situation can be easily changed to model different modulations and polarization of the electrical signal, including them in the *driver* model.

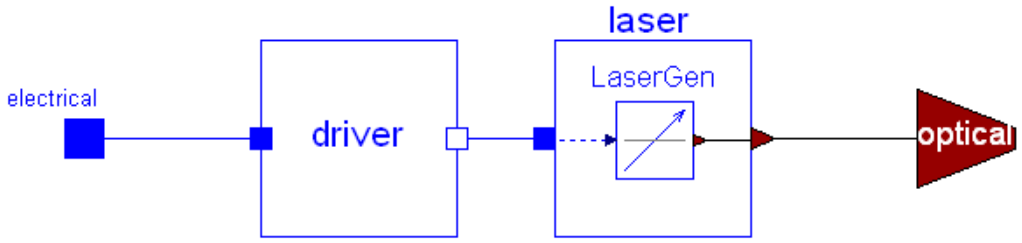


Figure 7.10: Opto-electrical transmitter modeled using DEVSLib.

The *laser* receives the electrical current from the driver. It is composed of the *LaserGen* model, which translates the continuous electrical current into discrete-events, that represent the optical impulses.

The *LaserGen* model has been build using the CrossUP model included in DEVSLib. This model receives a continuous-time input signal (i.e., the electrical current) and a reference value, and generates a discrete-time message every time the signal crosses the reference value in the upwards direction. This behavior represents the generation of an optical impulse every time the electrical current reaches a given value.

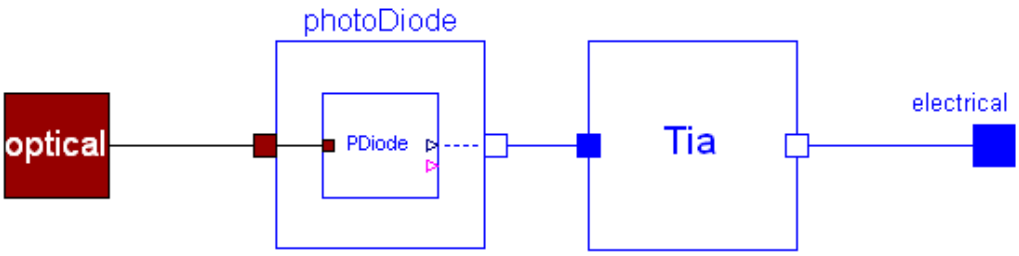


Figure 7.11: Opto-electrical receiver modeled using DEVSLib.

The description of the receiver model is shown in Fig. 7.11. It is composed of the photodiode and the TIA amplifier. The optical impulses are received in the optical input port. These impulses are sent to the *photoDiode* model that translates each impulse into a current variation. The generated current is sent to the *Tia* model, which sends it through the electrical output port.

The photodiode model is composed of a *PDiode* that performs the translation from optical impulses into electrical signals. The Pdiode model has been composed using the DICO model included in DEVSLib. The DICO model translates series of messages into a real piecewise-constant signal, with values equal to the values of the received messages. Each time the *PDiode* model receives an optical impulse it raises the generated current to the value of the received optical impulse. Bigger optical impulses generate higher current variations. This action simulates the excitation of the photodiode. After the optical impulse is received, the generated current decreases, because of the lack of optical excitation in the photodiode.

In order to simplify the system complexity, the *Tia* model does not modify the electrical signal. However, as in the case of the *driver* model, the *Tia* model can be modified using Modelica components, or equations.

7.4.3 Experiment and Results

As mentioned before, a simple example of opto-electrical communication system has been constructed using the previously described transmitter and receiver models in Modelica/DEVSLib. This model is compared with an equivalent model constructed by the ARS Lab group using CD++ [Sanz, Jafer, Wainer, Nicolescu, Urquia and Dormido, 2009].

In CD++, the continuous electrical current is translated into a set of discrete inputs using quantized DEVS models. The input can be read from an external "event" file, which are transmitted to the driver component via its data input port. The processing time of the driver model is set to 2 seconds. This parameter can be changed easily to reflect different scenarios.

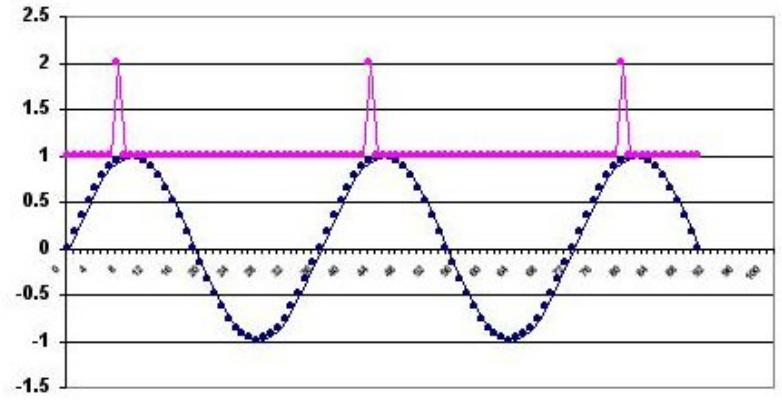
On the other hand, the input of the Modelica/DEVSLib model is generated by a current source, available in the Modelica Standard Library. This current source generates a continuous-time sinusoid current. The sine current activates the optical generation in the transmitter, generating impulses.

In the Modelica/DEVSLib model, the *LaserGen* has been configured as a cross-function with value 0.9A, because the input current corresponds to a sine input with amplitude 1A. The translation of the input current into optical impulses is shown in Fig. 7.12.

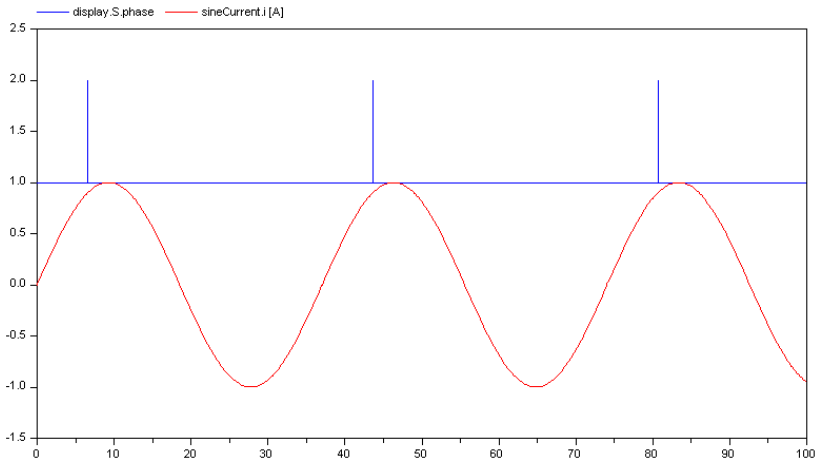
These impulses are received by the receiver and translated into current again. In the Modelica model, the derivative of the generated current is set to -0.5 to reproduce the lack of excitation. To monitor the generated current in the Modelica receiver, a current sensor has been included, also from the Modelica Standard Library. The reception of the optical impulses and its translation into electrical current is shown in Fig. 7.13.

The simulation results of the complete system are shown in Fig. 7.14. It can be noticed that no communication delays have been included in the models, but it will be easy to include them as additional atomic DEVS models.

It can be noticed that the obtained results are very similar in both models.

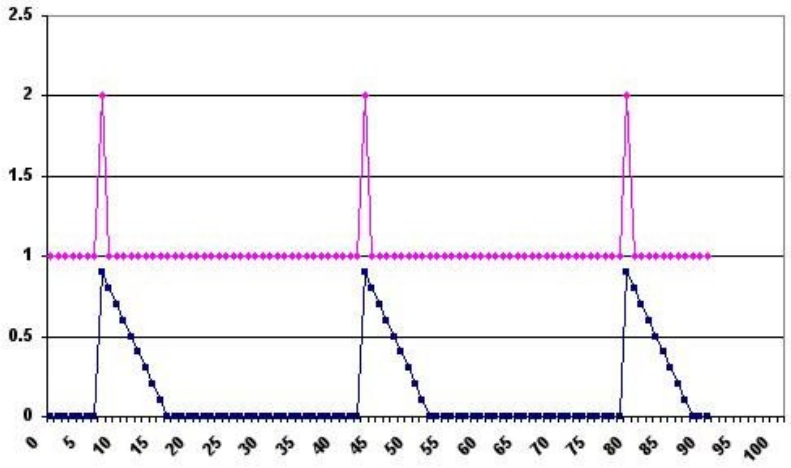


(a)

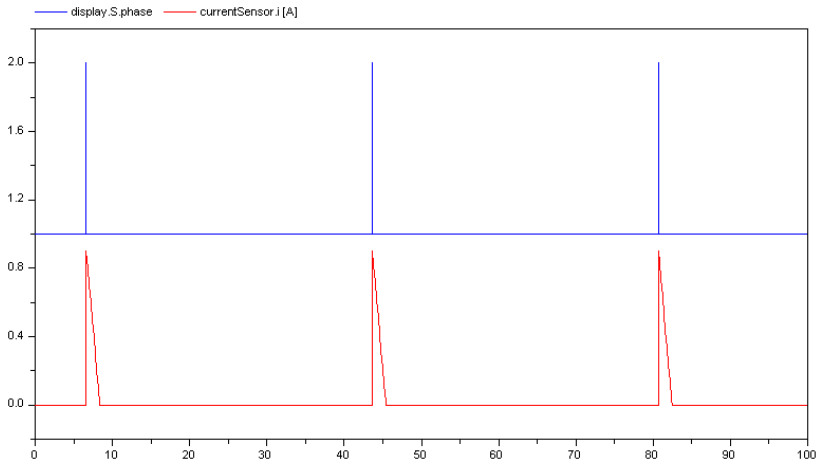


(b)

Figure 7.12: Sinusoid electrical current transformed into optical impulses, modeled with: a) CD++ and; b) Modelica [Sanz, Jafer, Wainer, Nicolescu, Urquia and Dormido, 2009].

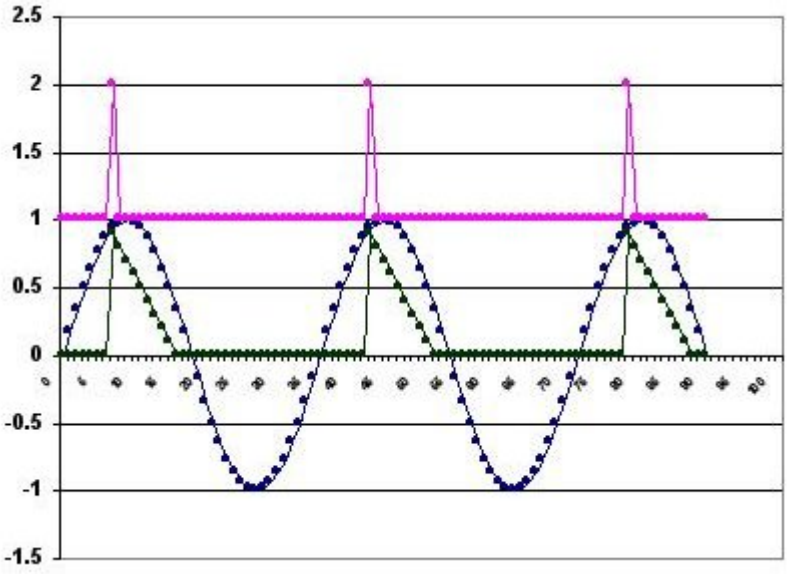


(a)

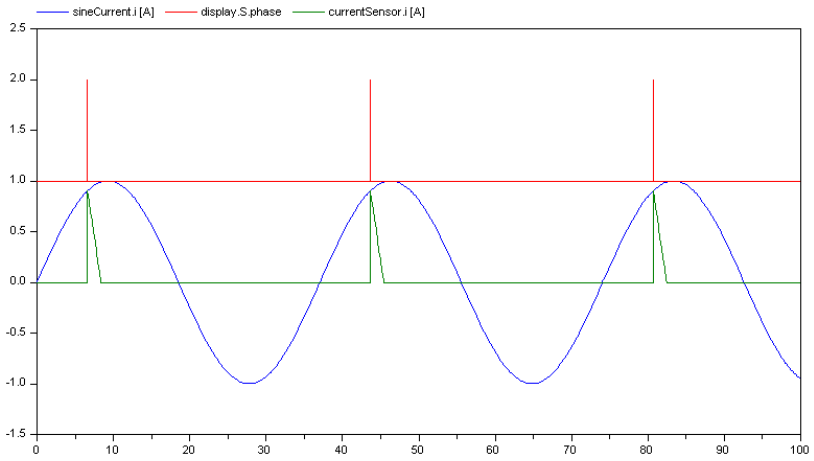


(b)

Figure 7.13: Optical impulses translated into current by the receiver, modeled with: a) CD++ and; b) Modelica [Sanz, Jafer, Wainer, Nicolescu, Urquia and Dormido, 2009].



(a)



(b)

Figure 7.14: Opto-electrical communication system, modeled with: a) CD++ and; b) Modelica [Sanz, Jafer, Wainer, Nicolescu, Urquia and Dormido, 2009].

7.5 Conclusions

The combination of the functionalities of the P-DEVS formalism and the Modelica language facilitate the description of hybrid dynamic models. The DEVSLib library supports the description of P-DEVS models in Modelica, and includes interface models to facilitate the connection between DEVSLib models and other Modelica models.

Since DEVSLib models communicate using a message passing mechanism, the messages have to be translated in order to be used in other Modelica models. Two kind of interfaces have been included in DEVSLib:

- The message-to-signal interfaces, that translate series of messages into discrete-time signals.
- The signal-to-message interfaces, that translate continuous-time and discrete-time signals into series of messages.

These interfaces have been successfully applied to the description of a two-tank system with a discrete controller, and the description of an opto-electrical communication system. Both models are composed of a part described using P-DEVS, and implemented using DEVSLib, and a continuous-time part described using Modelica. These parts communicate using the developed interface models.

8

Modeling of Hybrid Control Systems Using DEVSLib

8.1 Introduction

Hybrid models, which define the interaction of continuous-time and discrete-event dynamics, are used for describing control systems. For instance, control systems where a continuous-time plant is controlled using discrete-time or event-based controllers.

The functionalities provided by the DEVSLib library in order to describe hybrid control systems are described in this chapter. The Parallel DEVS formalism and the Modelica language are combined to describe the controller and the plant. This combined approach facilitates the description of hybrid models, including the interactions between continuous and discrete parts.

DEVSLib has been successfully used to describe hybrid control systems. Two case studies are presented, "Supermarket Refrigeration" system [Sarabia et al., 2009; Larsen et al., 2007] and "Crane and Embedded Controller" system [Schiftner et al., 2006].

8.2 Modeling of Hybrid Control Systems Using DEVSLib

This section discusses the application of DEVSLib components and functionalities to describe hybrid control systems. The description of sensors and actuators, and

of discrete (time or event based) controllers is presented. The description of the plant is considered to be performed using a continuous-time model, and thus its description is not discussed.

8.2.1 Sensors and Actuators

Sensors and actuators are required to communicate the continuous-time model of the plant with the controller. Since time-based sensors are already available in the Modelica Standard Library, DEVSLib does not include them.

Event-based sensors can be modeled using the DEVSLib interfaces described in Section 7.2. The CrossUP and CrossDOWN models can be used as threshold detectors. The quantizer model can be used to observe the outputs of the plant, avoiding time-discretization techniques (i.e., sampling).

These event-based sensors translate the information from the outputs of the plant into messages. These messages represent the inputs of the discrete-event controller.

The outputs of the discrete-event controller are also represented using messages. These messages have to be translated and transferred to the plant. The DEVSLib message-to-signal models (i.e., DICO and DIBO) can be used to translate the generated control signal into a discrete-time real or boolean signal. Also, DEVSLib includes the "setValue" model, which generates a message with a given value every time the model receives an external message. This model can be used to change the value of a message. For example, it can be used to communicate constant control actions (e.g., a maximum, minimum, 0 or 1 values), independently of the value received from the controller.

For instance, consider the temperature control of a heating system. The heater is turned off when the room temperature reaches a certain value. Using DEVSLib, this system can be implemented using a CrossUP model to detect the maximum temperature. The output of the CrossUP is connected to a setValue model, that sends a message with value 0. This message is received by a DIBO model that sets its output to false, due to the 0 value of the message, turning

off the heating system. The description of this simple system using DEVSLib is shown in Fig. 8.1.

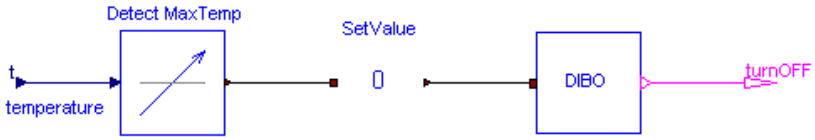


Figure 8.1: Simple temperature control system described using DEVSLib.

8.2.2 Controllers

DEVSLib can be used to describe discrete-time and event-based controllers. Depending on the complexity of the actions performed by the controller, it can be implemented as a single atomic model or as a coupled model. The transition functions of atomic models may contain the algorithms to calculate the control signal. On the other hand, coupled models can be constructed by combining simpler actions (e.g., the temperature control shown in Fig. 8.1).

Discrete-time controllers can be described using an atomic DEVSLib model that only executes periodic internal transitions. Each internal transition represents a sample interval. The time advance function schedules a new internal transition for the next sampling time. The inputs for the controller are represented by continuous-time inputs for the atomic DEVSLib model, as described in Section 5.3. In this way, no additional sampling is required to read the outputs of the plant.

Event-based controllers can be described using either atomic or coupled DEVSLib models, depending on their complexity. Controller inputs are received as messages through the input ports from the DEVSLib sensors. The control signals are also generated as messages and sent through the output ports. These messages need to be translated using the mentioned message-to-signal models (i.e., actuators), in order to connect them with the inputs of the plant. The following case studies present the construction of these kind of controllers using DEVSLib.

8.3 Supermarket Refrigeration System

The DEVSLib functionalities for describing hybrid control systems are applied to the development of a supermarket refrigeration system. This system was proposed in Larsen et al. [2007] as a benchmark for hybrid control applications.

A supermarket refrigeration system is composed of three main components: the display cases, the suction manifold and the compressor rack. The display cases contain the refrigerated goods offered to the customers. These display cases contain an evaporator that is connected to a pressure line. The objective is to control the temperature of the goods, which is approximated by the temperature of the air inside the display. Some external disturbances affect the temperature of the air in the display.

The refrigerant in each display case flows into the suction manifold. A compressor rack provides refrigerant to the displays by compressing the refrigerant in the suction manifold. Each display has an inlet valve to control the flow of refrigerant. Each compressor in the rack can be switch on and off, depending on the pressure in the line. The traditional control approach for these kind of refrigeration systems include two main controllers: the air temperature control included in the display cases, and the pressure control included in the compressor rack.

This section includes a description of the plant and the traditional control approach described in Larsen et al. [2007] and Sarabia et al. [2009]. These components have been modeled using plain Modelica code, the Modelica Standard Library and DEVSLib. A comparison of the simulation results obtained by these three implementations is included.

8.3.1 Display Case

The dynamics of the display case are represented by four state variables: the temperature of the goods T_{goods}, the temperature of the air T_{air}, the temperature of the evaporator wall T_{wall} and the mass of liquified refrigerant in the evaporator

M_{ref}. The inputs of the display are: the pressure in the suction line P_{suc}, the state of the inlet valve (open/close) *valve* and the disturbance $Q_{airload}$.

The following three equations describe the energy-balance between the goods, the air curtain and the evaporator [Larsen et al., 2007].

$$\frac{dT_{goods}}{dt} = -\frac{Q_{goods-air}}{M_{goods} \cdot Cp_{goods}} \tag{8.1}$$

$$\frac{dT_{wall}}{dt} = \frac{Q_{air-wall} - Q_e}{M_{wall} \cdot Cp_{wall}} \tag{8.2}$$

$$\frac{dT_{air}}{dt} = \frac{Q_{goods-air} + Q_{airload} - Q_{air-wall}}{M_{air} \cdot Cp_{air}} \tag{8.3}$$

where $Q_{airload}$ is the external disturbance on the air curtain and

$$Q_{goods-air} = UA_{goods-air} \cdot (T_{goods} - T_{air}) \tag{8.4}$$

$$Q_{air-wall} = UA_{air-wall} \cdot (T_{air} - T_{wall}) \tag{8.5}$$

$$Q_e = UA_{wall-ref}(M_{ref}) \cdot (T_{wall} - T_e) \tag{8.6}$$

UA is the overall heat transfer between media (defined with subscripts). M denotates the mass, Cp the heat capacity of the media, and T_e the evaporation temperature (approximated by Eq. (8.7), in absence of pressure drop in the suction line).

$$T_e = -4.3544 \cdot P_{suc}^2 + 29.2240 \cdot P_{suc} - 51.2005 \tag{8.7}$$

The heat transfer coefficient between the evaporator wall and the refrigerant is a function of the mass of liquified refrigerant (see Eq. (8.6)), which is approximated by the following linear function:

$$UA_{wall-ref}(M_{ref}) = UA_{wall-ref,max}\frac{M_{ref}}{M_{ref,max}} \tag{8.8}$$

The accumulation of refrigerant in the evaporator is described by:

$$\frac{dM_{ref}}{dt} = \begin{cases} \frac{M_{ref,max} - M_{ref}}{\tau_{fill}} & \text{if } valve = 1 \\ \frac{Q_e}{\triangle H_{lg}} & \text{if } valve = 0, M_{ref} > 0 \\ 0 & \text{if } valve = 0, M_{ref} = 0 \end{cases} \tag{8.9}$$

where τ_{fill} is the filling time of the evaporator, $\triangle H_{lg}$ is the specific latent heat of the remaining liquified refrigerant in the evaporator, which is approximated by Eq. (8.10).

$$\triangle H_{lg} = (0.0217 \cdot P_{suc}^2 - 0.1704 \cdot P_{suc} + 2.2988) \cdot 10^5 \tag{8.10}$$

The mass of refrigerant leaving the evaporator into the suction manifold is described by:

$$m = \frac{Q_e}{\triangle H_{lg}} \tag{8.11}$$

The temperature control for the display case is defined as an hysteresis controller that opens and closes the inlet valves to regulate the temperature of the air T_{air}. The parameters for the controller are the thresholds for the maximum and minimum temperatures ($\overline{T}_{air}$ and $\underline{T}_{air}$). The hysteresis is operated at a sample time of 1 second. The state of the valves at the k^{th} sample is defined as:

$$valve(k) = \begin{cases} 1 & \text{if } T_{air} > \overline{T}_{air} \\ 0 & \text{if } T_{air} < \underline{T}_{air} \\ valve(k-1) & \text{if } \underline{T}_{air} < T_{air} < \overline{T}_{air} \end{cases} \tag{8.12}$$

The model of the display case has been developed translating the equations described above into plain Modelica code. Interface ports have been added to the model in order to allow its connection with the other elements of the refrigeration system. The developed model is shown in Fig. 8.2.

The air controller detailed in Fig. 8.2b includes a CrossUP model to detect the maximum temperature for the air. When the air temperature reaches the maximum, the CrossUP generates a message which is translated into another

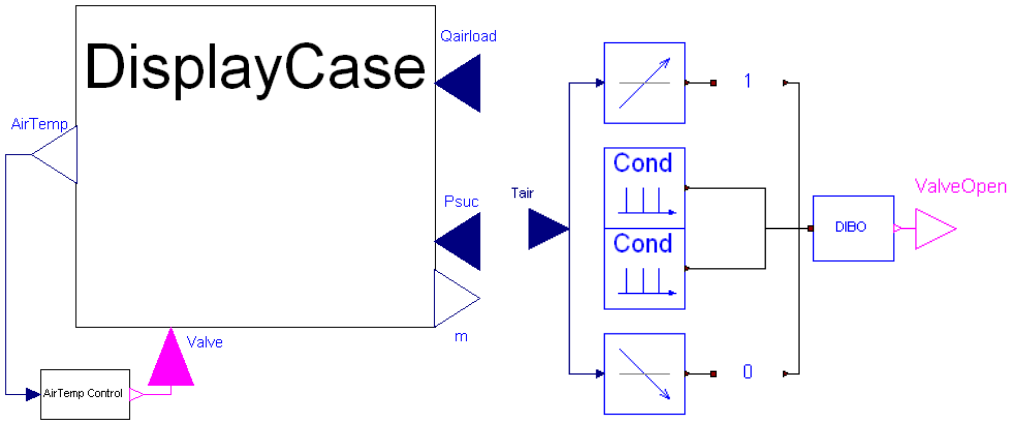


Figure 8.2: a) Display case, including air controller; and b) detail of air controller modeled using DEVSLib.

message with value 1 by the setValue model. This last message is translated by the DIBO model into a "true" value for the valveOpen port. The CrossDOWN model detects the lower limit for the air temperature and actuates similarly to the CrossUP model, but in this case the setValue generates a message with value 0, which will close the valve (i.e., setting to "false" the value of the valveOpen port). The two "Cond" models included in the center of the air controller check the initial conditions for the air temperature, setting the correct value for the valve at the beginning of the simulation.

8.3.2 Suction Manifold

The pressure of the suction line, P_{suc}, is described by:

$$\frac{dP_{suc}}{dt} = \frac{m_{in-suc} + m_{ref,const} - V_{comp} \cdot \rho_{suc}}{V_{suc} \cdot \frac{d\rho_{suc}}{dP_{suc}}} \tag{8.13}$$

where V_{suc} is the total volume of the suction manifold, V_{comp} is the volume flow created by the compressors, m_{in-suc} is the sum of refrigerant mass from the display cases into the suction manifold, $m_{ref,const}$ is a constant mass flow into the suction manifold from unmodeled entities, and ρ_{suc} is the density in the suction

manifold (approximated by Eq. (8.14)).

$$\rho_{suc} = 4.6073 \cdot P_{suc} + 0.3798 \tag{8.14}$$

The model of the suction manifold has been developed similarly to the display case, translating Eq. (8.13) into plain Modelica code. The interface of the model is composed of three inputs (m, V_{comp} and $m_{ref,const}$) and one output (P_{suc}).

8.3.3 Compressor Rack

The volume flow generated by each compressor is:

$$V_{comp,i} = comp_i \cdot \frac{1}{100} \cdot \eta_{vol} \cdot V_{sl} \quad i = 1, ..., q \tag{8.15}$$

where q is the number of compressors in the rack, $comp_i$ is the capacity of the i^{th} compressor, η_{vol} is the volumetric efficiency and V_{sl} is the total displacement volume.

The pressure control for the compressor rack is defined as a PI controller with a dead band (DB) around the reference pressure (see Eq. (8.16)). This controller is typically operated at a sample time of 60 seconds.

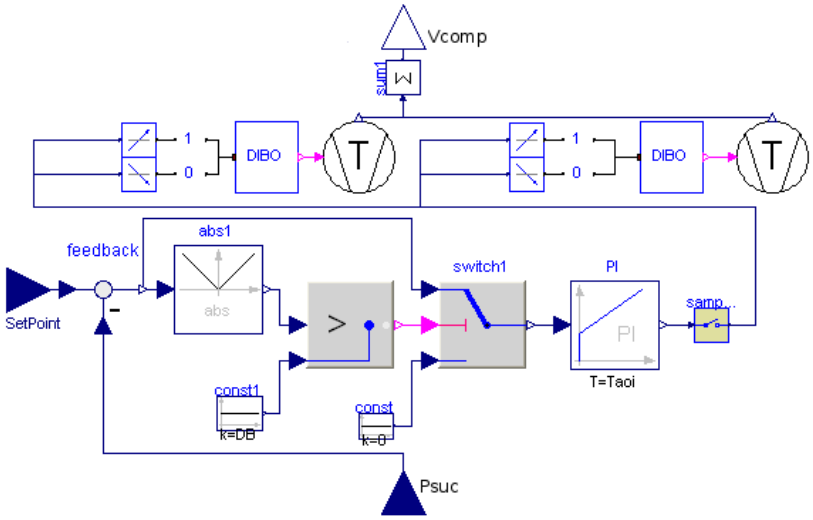
$$u_{PI}(t) = K_p e(t) + \int \frac{e(t)}{k_i} dt \tag{8.16}$$

where

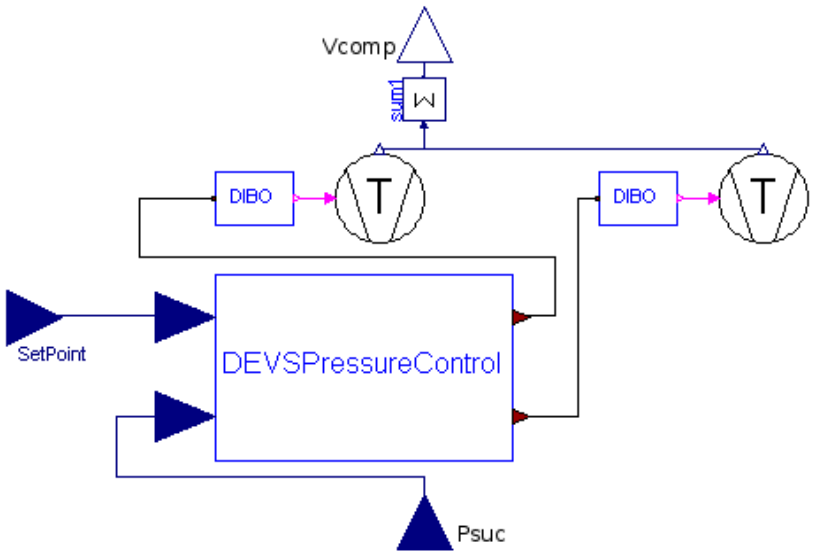
$$e(t) = \begin{cases} P_{suc}^{ref} - P_{suc} \text{ if } |e(t)| > DB \\ 0 \qquad\qquad\quad \text{otherwise} \end{cases} \tag{8.17}$$

For all the compressors in the rack, the nc^{th} compressor is switched on if Eq. (8.18) is satisfied, and switched off in any other case.

$$u_{PI} \geq \sum_{i=1}^{nc-1} C_{comp,i} + \frac{C_{comp,nc}}{2} \tag{8.18}$$



(a)



(b)

Figure 8.3: Pressure control modeled using: a) DEVSLib and the MSL; and b) an atomic DEVSLib model.

The compressor rack has been modeled using three different approaches. The dynamics of the compressors, Eq. (8.15), have been modeled using plain Modelica code, and are common for the three approaches:

- The first approach also uses plain Modelica code to describe the PI control.
- The second approach uses elements from DEVSLib and the Modelica Standard Library to describe the PI control and the activation of the compressors (detailed in Fig. 8.3a). The sampled control signal generated by the PI

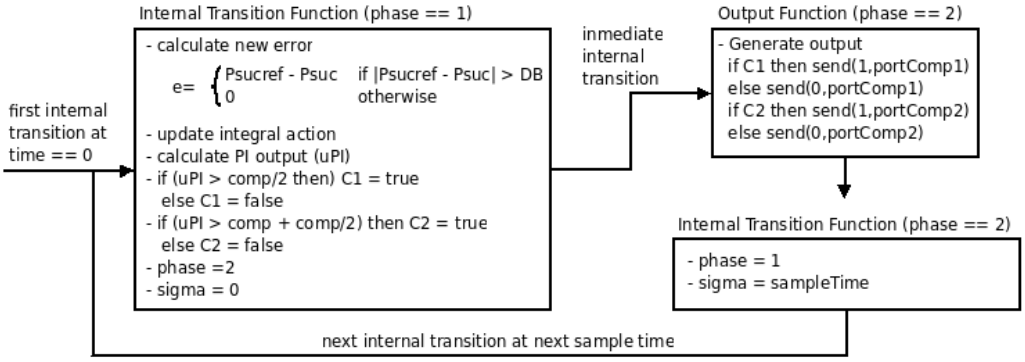


Figure 8.4: Actions performed by the atomic DEVSLib PI controller (note that no output is generated with phase == 1).

controller is evaluated by "CrossUP" and "CrossDOWN" models in order to decide which compressors have to be activated at each sample time.

– The third approach includes an atomic DEVSLib model that represents the PI control, and DIBO models to translate the generated control signal to the compressor models (detailed in Fig. 8.3b). In this case, the state of the PI controller is calculated at each sample time, instead of calculating it continuously and only sampling its output. The actions performed by this atomic DEVSLib PI controller are shown in Fig. 8.4. The controller executes its first sampling at time 0s. At each sample time, it performs these actions:

  1. Executes the output function with phase == 1, and no output is generated (not shown in Fig. 8.4).
  2. Updates the state of the PI controller and decides the next state for the compressors.
  3. Executes again the output function with phase == 2, and sends the new states to the compressors.
  4. Executes again the internal transition function to schedule the next sample (sigma = sampleTime).

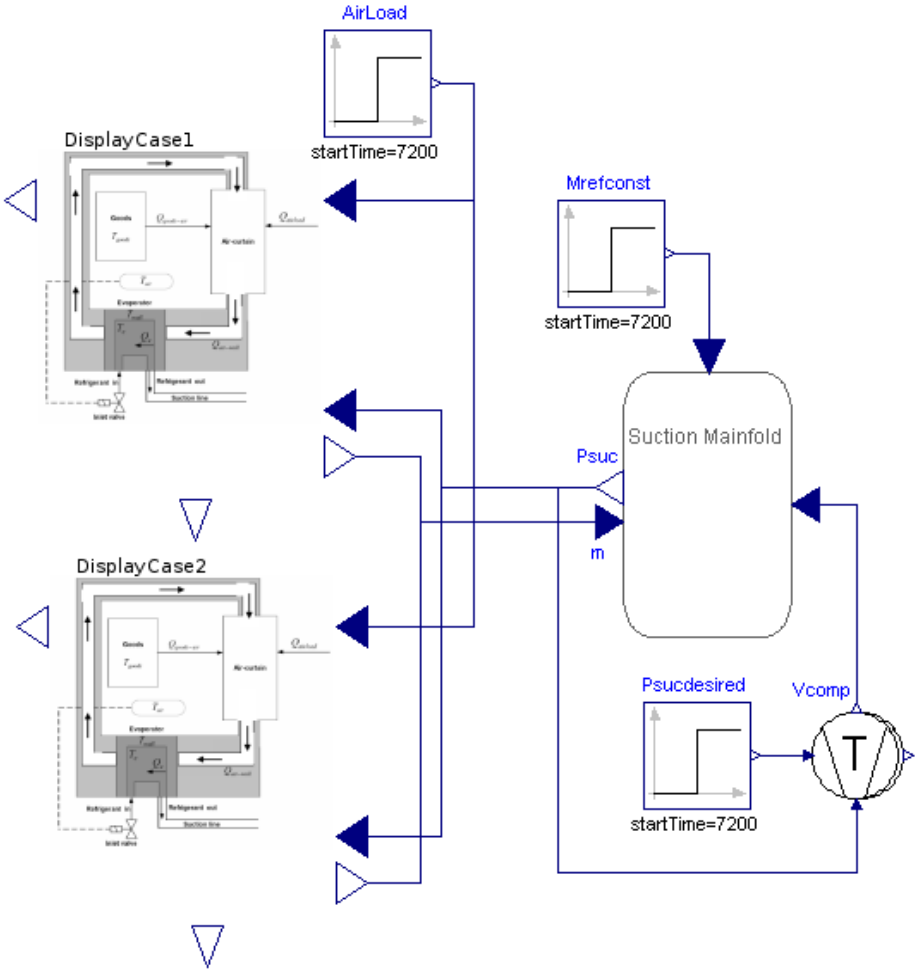


Figure 8.5: Supermarket refrigeration system modeled using DEVSLib and Modelica.

8.3.4 Experiment Setup and Simulation Results

The whole refrigeration system is composed of two display cases, one suction manifold and a compressor rack, that includes two compressors. The developed model is shown in Fig. 8.5.

The system is evaluated during a day/night operation. During the day, the external disturbance in each display case is set to $3000J \cdot s^{-1}$. During the night, each display case is covered with "night-covers" that reduce the external disturbance to $1800J \cdot s^{-1}$, and the constant mass flow in the suction manifold from 0.2 to $0.0kg \cdot s^{-1}$.

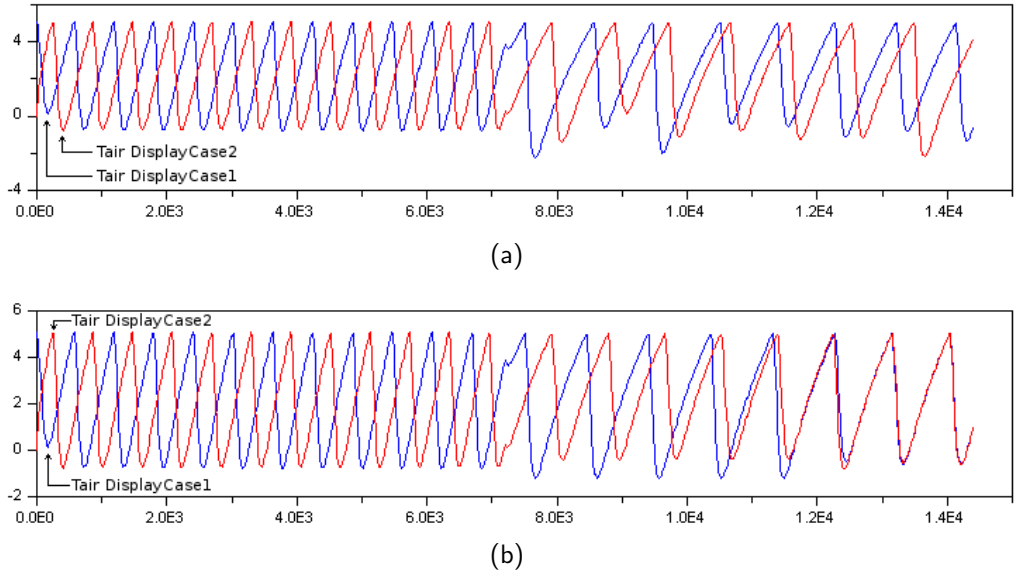


Figure 8.6: Evolution of air temperatures in both displays using: a) first and second control approaches; b) atomic DEVSLib control approach.

Table 8.1: Parameters for the supermarket refrigeration system.

Display Cases		
M_{goods}	200	Kg
Cp_{goods}	1000	$J \cdot Kg^{-1} \cdot K^{-1}$
$UA_{goods-air}$	300	$J \cdot s^{-1} \cdot K^{-1}$
M_{wall}	260	Kg
Cp_{wall}	385	$J \cdot Kg^{-1} \cdot K^{-1}$
$UA_{air-wall}$	500	$J \cdot s^{-1} \cdot K^{-1}$
M_{air}	50	Kg
Cp_{air}	1000	$J \cdot Kg^{-1} \cdot K^{-1}$
$UA_{wall-ref,max}$	4000	$J \cdot s^{-1} \cdot K^{-1}$
$M_{ref,max}$	1	Kg
τ_{fill}	40	s
Suction Manifold		
V_{suc}	5	m^3
Compressor Rack		
V_{sl}	0.08	$m^3 \cdot s^{-1}$
η_{vol}	0.81	$-$

The limit, maximum and minimum, temperatures for the air in the display cases are $5°C$ and $2°C$ respectively. The reference pressure for the compressor rack is set to $1.4bar$ during the day, and $1.6bar$ during the night. The capacity of

Table 8.2: Initial conditions for state variables.

	Disp. Case 1	Disp. Case 2
T_{wall}	0°C	0°C
T_{air}	5.1°C	0°C
T_{goods}	2°C	2°C
M_{ref}	0°C	0°C

each compressor in the rack is set to 50. The rest of the parameters and initial conditions for the system are shown in Tables 8.1 and 8.2.

The system is simulated during $14400s$, defining the switching between day and night at time $7200s$. Simulation results are shown in Fig. 8.6, including the evolution of the air temperatures. The results obtained from the models including the first and second pressure control approaches are equal and overlap (see Fig. 8.6a). The results obtained from the third approach (with the atomic DEVSLib pressure controller) are slightly different from previous ones (see Fig. 8.6b). These differences are explained because in the first and second approaches, the PI control operates in continuous-time and only its output is sampled, while in the atomic DEVSLib PI controller all the calculations are performed at the same time, and remain constant between samples. The results obtained with the third approach are similar to the results obtained by Sarabia et al. [2009], with a model of the supermarket refrigeration system constructed using EcosimPro.

8.4 Crane and Embedded Controller System

This section discusses the implementation of the ARGESIM comparison "Crane and Embedded Controller". ARGESIM is a non profit working group providing the infrastructure and administration for dissemination of information on M&S in Europe [ARGESIM, 2009].

The system consists in a crane controlled by a discrete controller. The crane is composed of a car that moves along a horizontal rail and a load connected to the car by a cable. This system was proposed by ARGESIM as a comparison for different tools that support hybrid modeling.

Various authors implemented this system using different tools or languages. Some implementations were based on the original definition of the system [Scheikl et al., 2002]. These implementations were developed using Matlab [Scheikl, 2001; Schachinge, 2002; Wöckl and Breitenecker, 2003; Weidinger and Breitenecker, 2003], Anylogic [Garifullin, 2003] and VHDL-AMS [Wang and Kazmierski, 2005]. Another implementation, based on the revised definition of the system [Schiftner et al., 2006], uses Modelica/Dymola [Schiftner, 2006].

The original and revised definitions of the system differ in the design of the controller. In the original definition, the controller receives as inputs the angle of the load and the position of the car. In the revised definition, only the latter variable is received. Also, the equations of the controller and its parameters are improved in the revised definition.

The implementation described in this section follows the revised definition of the system. The crane system has been described as a continuous-time model using the Modelica language. The controller has been modeled in part using the DEVSLib library and in part using the Modelica Standard Library (MSL). Both parts are interconnected using the DEVSLib interface models [Sanz, Cellier, Urquia and Dormido, 2009].

8.4.1 Crane System Model

The crane system is composed of the car, the cable, and the load (see Fig 8.7). The discrete controller controls the position of the car to reach the desired position, specified by the user. A detailed description of the system is given in Schiftner et al. [2006].

The car moves along the track in accordance with a force f_c, provided by a motor. The force of the motor is calculated using the following first-order ODE (Ordinary Differential Equation) [Schiftner et al., 2006]:

$$\dot{f}_c = -4(f_c - f_c^{desired}) \tag{8.19}$$

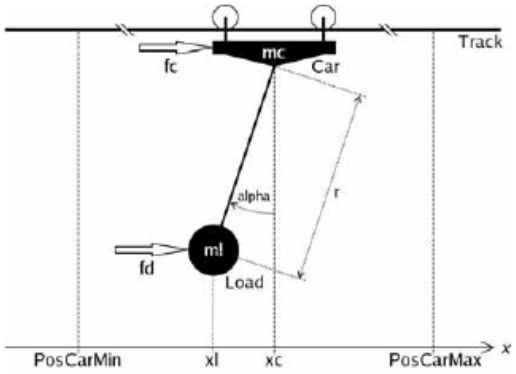


Figure 8.7: Scheme of the crane system [Schiftner et al., 2006]

Where $f_c^{desired}$ is the signal generated by the discrete controller. The car can also be stopped using a brake, whose activation conditions will be detailed below. The movement of the car is restricted to the values $PosCarMax$ and $PosCarMin$. The load hangs from the car by a cable. Similarly to the car, the load is influenced by a force f_d, that represents some disturbances.

Three sensors are used to observe the state of the system:

1. The position of the car (named $PosCar$).
2. The maximum position limit (named $SwPosCarMax$, is activated when $PosCar > PosCarMax$).
3. The minimum position limit (named $SwPosCarMin$, is activated when $PosCar < PosCarMin$).

If either the $SwPosCarMax$ or the $SwPosCarMin$ sensor is active, the system enters $EmergencyMode$, which causes an emergency stop.

Table 8.3: Model variables

Symbol	Description	Unit
α	angle of the cable	rad
f_c	motor force	N
f_d	load disturbances	N
x_c	position of the car	m
x_l	position of the load	m

Table 8.4: Model parameters

Symbol	Description	Value
d_c	friction coefficient of the car	0.5 kg/s
d_c^{brake}	friction coefficient of the car with activated brake	10^5 kg/s
d_l	friction coefficient of the load	0.01 kg/s
g	gravity	9.81 m/s^2
m_c	mass of the car	10 kg
m_l	mass of the load	100 kg
$PosCarMax$	maximum position of the car	5 m
$PosCarMin$	minimum position of the car	-5 m
r	length of the cable	5 m

The equations [Schiftner et al., 2006] that describe the dynamics of the crane system are the following (variables and parameters are detailed in Tables 8.3 and 8.4):

$$\begin{aligned}\ddot{x_c}\left[m_c + m_l \sin^2(\alpha)\right] &= -d_c\dot{x_c} + f_c + f_d \sin^2(\alpha) \\ &+ m_l \sin(\alpha)\left[r\dot{\alpha}^2 + g\cos(\alpha)\right] - d_l\dot{x_c}\sin^2(\alpha)\end{aligned} \tag{8.20a}$$

$$\begin{aligned}r^2\ddot{\alpha}\left[m_c + m_l \sin^2(\alpha)\right] &= \left[f_d\frac{m_c}{m_l} - f_c + d_c\dot{x_c}\right] r\cos(\alpha) \\ &- \left[g(m_c + m_l) + m_l r\dot{\alpha}^2\cos(\alpha)\right] r\sin(\alpha) \\ &- d_l\left[\frac{m_c}{m_l}(\dot{x_c} r\cos(\alpha) + r^2\dot{\alpha}) + r^2\dot{\alpha}\sin^2(\alpha)\right]\end{aligned} \tag{8.20b}$$

$$x_l = x_c + r\sin(\alpha) \tag{8.20c}$$

Equations (8.20a), (8.20b), and (8.20c) can be linearized [Föllinger, 1985] to obtain the following linear model, in order to simplify the model and allow a comparison with the non-liner model:

$$\ddot{x_c} = \frac{f_c}{m_c} + g\frac{m_l}{m_c}\alpha - \frac{d_c}{m_c}\dot{x_c} \tag{8.21a}$$

$$r\ddot{\alpha} = -g(1 + \frac{m_l}{m_c})\alpha + (\frac{d_c}{m_c} - \frac{d_l}{m_l})\dot{x_c} - r\frac{d_l}{m_l}\dot{\alpha} - \frac{f_c}{m_c} + \frac{f_d}{m_l} \tag{8.21b}$$

$$x_l = x_c + r\alpha \tag{8.21c}$$

The model equations have been directly programmed in Modelica using equations (8.21a),(8.21b), and (8.21c) for the linear model, and (8.20a),(8.20b), and (8.20c) for the non-linear model. Both cases include (8.19), that models the motor. Dymola can internally handle the implicit equations of the non-linear model.

8.4.2 Discrete Controller Model

The complete system is shown in Fig. 8.8a. It corresponds to the discrete controller connected to the non-linear model of the crane system. The generators for the desired car positions and the load disturbances are also shown. The non-linear model can be substituted by the linear model. The controller, generators, and connections are compatible in both cases.

The controller is implemented as a cycle-based controller [Schiftner et al., 2006]. It is composed of three parts: the state-space observer, the regulator, and the diagnosis module. The state-space observer calculates five "fictitious" states (q), and the regulator generates a control signal based on the observed states. Additionally, the diagnosis module manages the conditions for the $EmergencyMode$ and the activation of the brake. The structure of the implemented controller is shown in Fig. 8.8b

Position Controller

The state-space observer and the regulator have been implemented with DEVSLib, as an atomic P-DEVS model. This model corresponds to the "PositionController" in Fig. 8.8b, and its P-DEVS specification is the following:

$$M = (X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$$

where:

$X_M = \emptyset$

$S = \{\Re^5 \times \Re\}$

$Y_M = \Re$

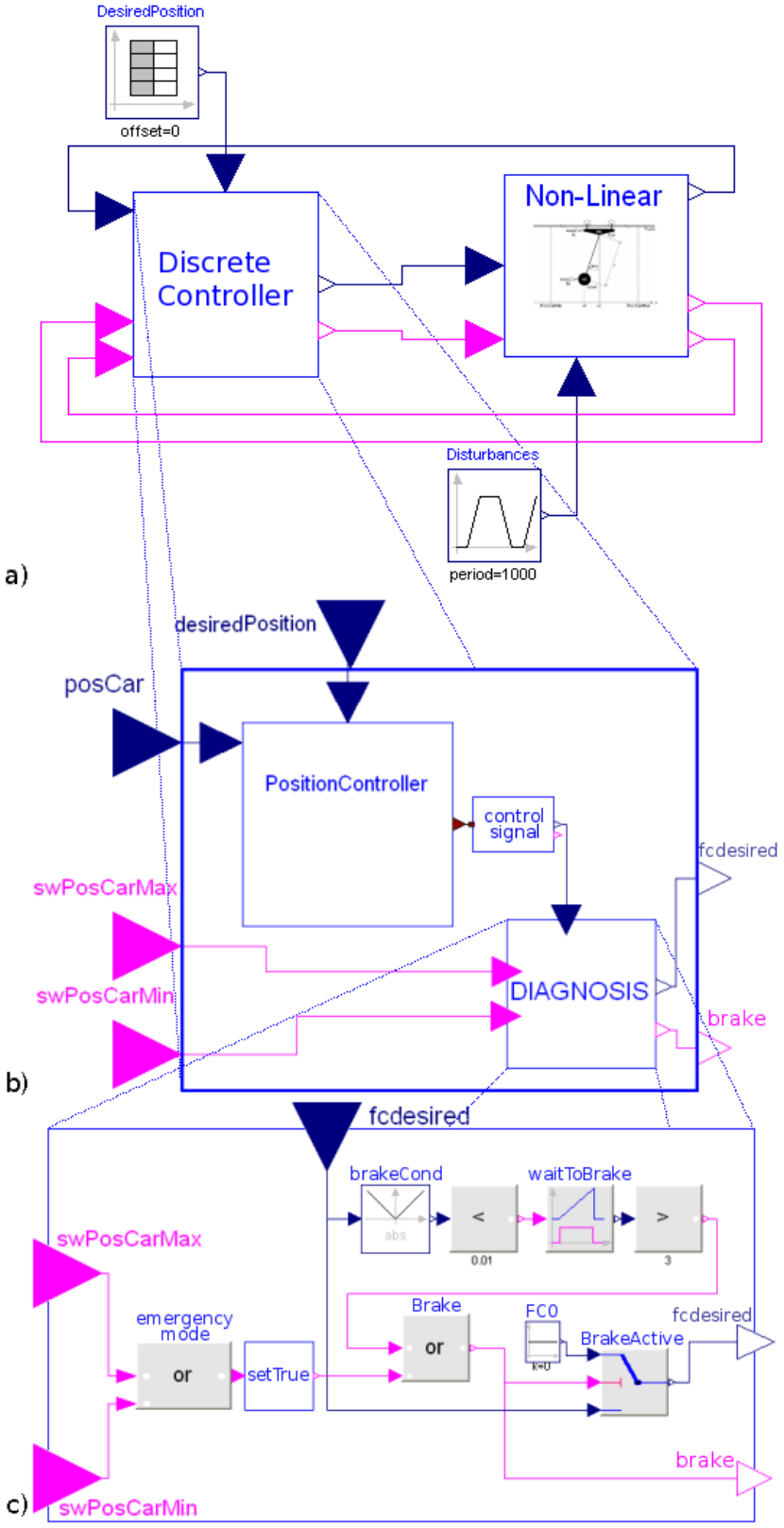


Figure 8.8: "Crane and Embedded Controller" system: a) non-linear system with discrete controller; b) discrete controller implemented with DEVSLib and the MSL; and c) diagnosis module of the controller

$$\delta_{int}(\mathbf{q}_n, u_n) = (\mathbf{q}_{n+1}, u_{n+1})$$
$$\text{where} \begin{cases} \mathbf{q}_{n+1} = (\mathbf{M} - \mathbf{dc}^T)\mathbf{q}_n + \mathbf{d}PosCar + \mathbf{b}f_c^{desired} \\ u_{n+1} = VPosDesired - \mathbf{h}^T\mathbf{q}_{n+1} \end{cases}$$

$$\delta_{ext}(q, u, e, X) = \text{nothing since } X_M = \emptyset$$

$$\delta_{con}(q, u, e, X) = \text{nothing since } X_M = \emptyset$$

$$\lambda(q, u) = max(min(u, ForceMax), -ForceMax)$$

$$ta(q, u) = cycle$$

M, **d**, **c**, and **b** are the parameters of the observer, and V and **h** are the parameters of the regulator. $PosCar$ and $PosDesired$ are continuous-time inputs to the δ_{int} function, as described in Section 5.3.

The "PositionController" executes an internal transition at each controller cycle. The internal transition function calculates the new state of the observer (q_{n+1}), and updates the control signal (u_{n+1}). The output function (λ) uses the parameter $ForceMax$ to saturate the control signal and generate the output that will be sent to the crane system.

The implementation of the "PositionController" is directly extracted from its specification, translating the actions performed by each transition function into Modelica functions. The output port (Y_M) is defined using a Modelica connector of the class "outPort", included in DEVSLib. The input parameters, *PosDesired* and *PosCar*, are defined using connectors from the MSL. Finally, the variables that define the state (S) have to be declared inside the Modelica record that represents the state of an atomic model in DEVSLib.

Interface Model

The output of the position controller is generated as a message. It contains the value of the control signal ($fcdesired$), which has to be translated into a discrete-time signal in order to be checked by the diagnosis module (see Fig. 8.8b). The translation is performed by the "controlSignal" model, which is implemented using the DICO interface model.

Diagnosis Module

The diagnosis module monitors the value of the control signal and the sensors $SwPosCarMax$ and $SwPosCarMin$. If any of the sensors becomes active, the controller enters in $EmergencyMode$ and activates the brake. The brake is also activated when $|f_c^{desired}| < BrakeCondition$ for more than 3 seconds (where $BrakeCondition$ is a parameter of the controller).

This module (cf. Figs. 8.8b and 8.8c) has been implemented using components from the MSL [MSL, 2010]. This demonstrates the compatibility between DEVSLib and previously developed Modelica Libraries [2010].

8.4.3 Simulation Results and Discussion

Three tasks (A, B, and C), described in the definition of the system [Schiftner et al., 2006], have been performed in order to compare this implementation with previous results.

Task A

This task compares the implementation of the linear and the non-linear models without the controller and the brake. The input of the plant ($f_c^{desired}$) is set to 160 N during 15 s, and then to 0 N. The load disturbances initially start at 0 N ($f_d = 0$). At time = 4 s, $f_d = Dest$ for 3 s. The $Dest$ values are -750 N, -800 N, and -850 N. The system is simulated for each $Dest$ value during 2000 s, to reach the steady-state, and the position of the load in each model is compared. The results are shown in Table 8.5.

Table 8.5: Task A results

Dest	Linear	Non-Linear	Difference
-750 N	294.081 m	294.059 m	0.022 m
-800 N	-0.0048651 m	-0.0325238 m	0.0276587 m
-850 N	-294.091 m	-294.18 m	0.089 m

The results obtained are very similar to the ones reported in Schiftner [2006]. The differences between the two models are -0.034 m, 0.013 m, and -0.016 m.

The slight differences between the two implementations are explained by the different implementation of the non-linear model – using the MultiBody library [Otter et al., 2003] instead of plain equations. The use of the DEVS formalism to describe the discrete controller facilitates its understanding and development, in comparison with its description in plain Modelica code.

Task B

The next task describes how the non-linear model and the discrete controller work together. The desired positions for the car are 3 m at time 0 s, -0.5 m at time 16 s, and 3.8 m at time 36 s. The load disturbance is set to -200 N at time 42 s during 1 s. The results include the position of the car, the position of the load, the angle, and the activation of the brake over time. The system is simulated for 60 s. The results are shown in Fig. 8.9, comparing the implementation with DEVSLib with the one presented in Schiftner [2006].

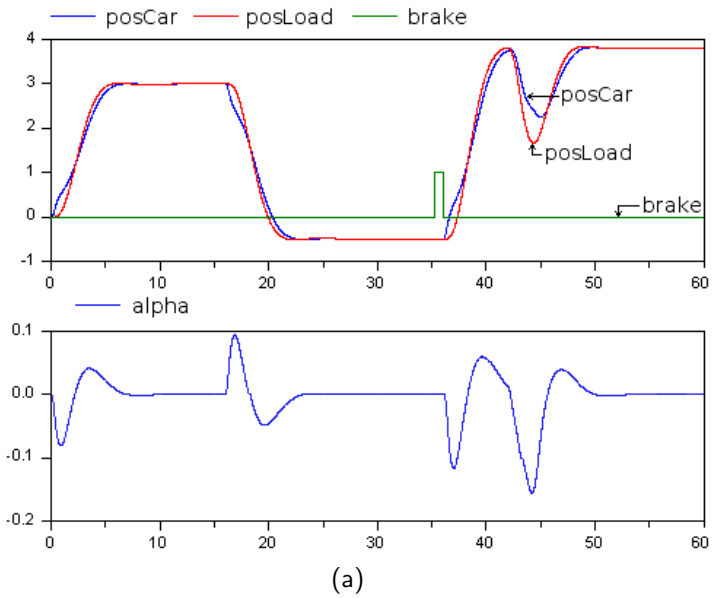
Task C

The last task evaluates the response of the system in case of an emergency stop. The scenario is the same as in the task B, but the value of the load disturbance is 200 N instead of -200 N. The results shown also include the position of the car, the position of the load, the angle, and the state of the brake over the simulation time. The system is simulated for 60 s. The simulation results are shown in Fig. 8.10, also comparing the two implementations.

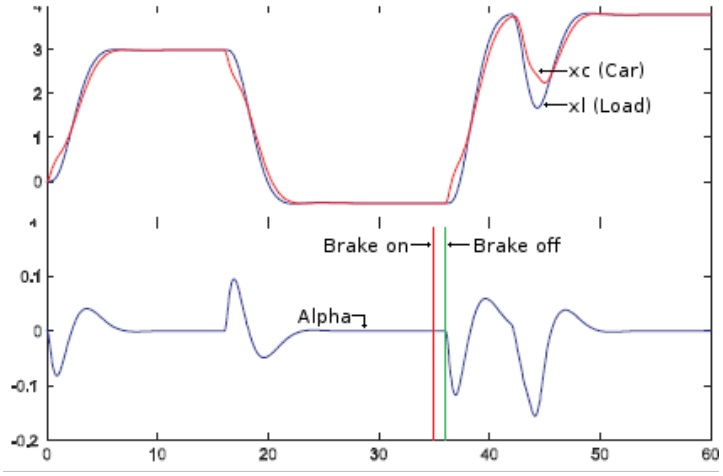
The emergency stop event is detected when the car reaches its maximum movement limit ($PosCarMax$). After that, the car stops and the load oscillates.

In this case there is a slight difference just before the emergency stop. This difference is due to the parameters of the experiment, because in the latter case the load disturbances are set to -200 N at time 42 s and to 200 N at time 46 s resulting in the observed delay of the emergency stop.

Similar comparisons can be performed with the results obtained in previous implementations [Wöckl and Breitenecker, 2003; Schachinge, 2002; Scheikl, 2001; Weidinger and Breitenecker, 2003; Wang and Kazmierski, 2005; Garifullin, 2003].



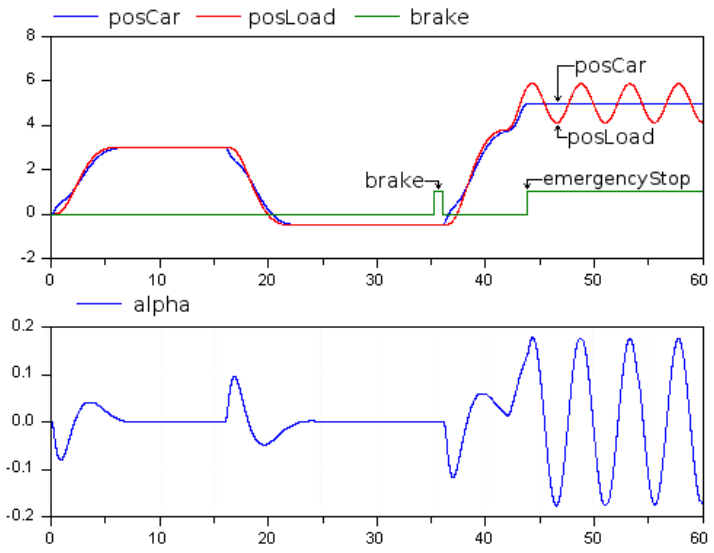
(a)



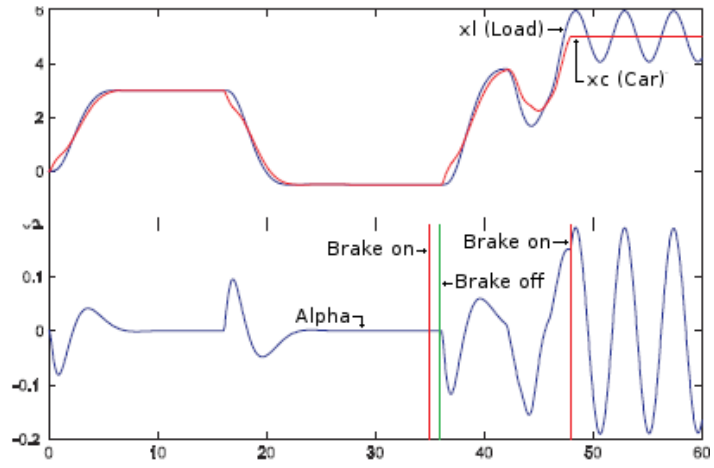
(b)

Figure 8.9: Task B results in: a) DEVSLib; and b) Schiftner [2006]

Their results are equivalent to the ones presented in this section. However, these previous implementations are based on the original definition of the model. Thus, their results are slightly different mainly due to the inclusion of the angle sensor as an additional input to the controller and the different design of the control.



(a)



(b)

Figure 8.10: Task C results in: a) DEVSLib; and b) Schiftner [2006]

8.5 Conclusions

DEVSLib includes interface models to combine P-DEVS models with the rest of the Modelica libraries, which facilitates the development of multi-domain and multi-formalism hybrid models. These interface models can be also used to describe event-based sensors and actuators, which are commonly used in the description of hybrid control systems. Also, the functionalities provided by DEVSLib can be applied to the description of discrete-time and event-based controllers.

DEVSLib has been successfully applied to the description of a supermarket refrigeration system. An event-based controller for the air temperature of the display cases has been developed using DEVSLib. Also, two different controllers for the refrigerant pressure line have been developed. The first approach combines components from the Modelica Standard Library and DEVSLib. The second approach describes the pressure controller as an atomic DEVSLib model. The simulation results of the system using the first control approach are equivalent to the same controller developed using plain Modelica. The results from the second approach are slightly different, due to the discrete-event nature of the whole controller.

The system described in the ARGESIM comparison "crane and embedded controller" has been implemented using Modelica and the DEVSLib library. The simulation results obtained with this implementation are equivalent to the ones obtained by previous implementations of the same system, using different tools. Other tools used to model this system describe the discrete behavior of the controller using formalisms like Finite State Automata, StateCharts, or Petri Nets.

9

Process-Oriented Modeling in Modelica

9.1 Introduction

In order to facilitate the description of models in Modelica following the process-oriented approach, two new Modelica libraries have been developed. The first library, named SIMANLib, reproduces some of the basic modeling functionalities of the SIMAN modeling language [Pegden et al., 1995]. The second library, named ARENALib, reproduces some of the basic modeling functionalities of the Arena simulation environment [Kelton et al., 2007].

The objective of this development is to show the feasibility of modeling discrete-event systems, following the process-oriented approach, using only the Modelica language functionalities. Other authors have combined different tools with Modelica in order to describe discrete-event systems [Remelhe, 2002]. SIMANLib and ARENALib also include functionalities to describe hybrid process-oriented models, in combination with other available Modelica libraries. The design, development and use of SIMANLib and ARENALib are discussed in Chapters 10 and 11. Another library, named RandomLib, has also been developed to facilitate the description of stochastic models (see Chapter 13).

The components of SIMANLib and ARENALib have been described using the P-DEVS formalism, and implemented using the DEVSLib library. However, in order to facilitate the description of models following the process-oriented approach

some additional functionalities are required. These additional requirements, the solutions proposed to accomplish them and their implementation are discussed in this chapter.

9.2 Additional Required Functionalities

Since SIMANLib and ARENALib components are developed using the DEVSLib library, their components are connected using the previously described message passing mechanism (see Chapter 4). The description of process-oriented models in Modelica requires the following additional functionalities:

- First, it is necessary to define and manage the information that describes the entities of the system. This information represents the content of the messages transfered between models.
- Second, simulation results are usually reported using statistical indicators, due to the stochastic nature of some discrete-event systems. Some of these statistical indicators have to be calculated during the simulation and some others at the end. The amount of data that has to be stored to calculate some of these indicators changes depending on the length of the simulation. An structure to store information in Modelica needs to be developed, giving the possibility to increase or decrease the size of the stored data during the simulation run. This storage structure has to be accessible from multiple points in the model, in order to facilitate the insertion and removal of data. The developed information storage structure, named *dynamic objects*, will be used to describe special attributes of the entities, global variables of the model and to store statistical indicators.

The implementation of these functionalities in Modelica is discussed in the following sections.

9.3 Entity Management

Each message, transported between two models, contains the information that describes an entity. DEVSLib by-default message contains two variables: Value and Type. An entity can not be described just using these two variables, and thus, an additional mechanism is required to define entities.

Entities in SIMANLib and ARENALib are described using a Modelica record, named *Entity*, that contains multiple variables (see Table 9.1). These variables are also used to store the information related to the entity during its transit through the system. These variables correspond to some of the main variables used in SIMAN and Arena to manage entities.

In order to associate the *Entity* record with the message passing mechanism, an external library in C, named "entities.c", has been programmed to store the

Table 9.1: Variables of the Entity record in SIMANLib and ARENALib.

Variable Name	Description
HoldCostRate	Cost rate of processing the entity in the system
VACost	Cost of Value Added processes applied to the entity
NVACost	Cost of Non-Value added processes applied to the entity
WaitCost	Cost of Waiting processes (delays, queues, etc.)
TranCost	Cost of Transport processes (transporters, conveyors, etc.)
OtherCost	Other costs associated with the entity
CreateTime	Time of creation for the entity
StartTime	Time of start for the current process
VATime	Accumulated time on Value Added processes
NVATime	Accumulated time on Non-Value Added processes
WaitTime	Accumulated time on Waiting processes
TranTime	Accumulated time on Transport processes
OtherTime	Accumulated time on Other processes
Number	Number of the entity (currently equal to SerialNumber)
SerialNumber	Unique number to identify the entity in the system
Type	Type of the entity
Attributes	Reference to the list of user-defined attributes, using an Assign module
Primary	Defines if the entity is a duplicate or not

records (i.e., entities) in dynamic memory using C structs (similarly to the management of messages in the message passing mechanism). The messages only transport in their Value variable a reference to the external struct, that represents an entity. This external library includes the following functions to manage the stored entities:

ECreate that creates a new entity using a set of initial values.

EDelete that removes an entity from the system and frees the used memory.

EGet can be used to obtain the value of any variable of the entity.

ERead can be used to obtain the value of all the variables of the entity simultaneously.

EUpdate can be used to update the value of all the variables of the entity simultaneously.

An additional problem appears when implementing processes that delay the entity. These models can include a delay time that represents the time spent processing the entity. Since the value of the delay time is usually random, the order of the arrived entities could not correspond to the order of the entities leaving the process. These processes have to include a temporal storage for the entities that are being delayed. Some SIMANLib models include an internal queue, similar to the one used to receive messages from other models, that can be used as a temporal storage for delayed entities. Entities in this temporal queue can be ordered depending on their arrival time, or the time they will finish the delay. In the latter case, the first entity in the queue will be the first to leave the process. If multiple entities have the same finishing time, all of them are removed simultaneously from the queue and leave the process.

9.4 Dynamic Object Management

A *dynamic object* is a two dimensional variable (i.e., a matrix) of real type stored in dynamic memory. Another external library in C, named “objects.c”, has been

programmed to manage dynamic objects. In Modelica, they are represented using an Integer variable, that stores a reference to the object in memory. It is similar to the Entity record described previously, but in this case the C struct stores a two-dimensional matrix of real numbers (e.g., `double **`) and the size of each dimension.

The management of the dynamic objects is performed using the following functions:

ObjCreate	used to create a new object.
ObjDelete	used to delete an existing object and free the memory.
ObjUpdateSize	used to modify the size of a currently existing object.
ObjUpdatePos	used to update the value of one of the positions of the object (e.g., $ObjUpdatePos(obj, 1, 1, -3)$ sets the value of the position $obj[1, 1]$ to -3).
ObjReadPos	used to read the value of one of the positions of the object.
ObjReadI1	used to read the size of the first dimension of the object.
ObjReadI2	used to read the size of the second dimension of the object.

Also, the developed external library supports the description of lists of dynamic objects. These are dynamic lists whose length can be modified during the simulation run, depending on the insertion and removal of objects in the list. The functions included to manage the lists of objects are:

ObjLCreate	used to create a new empty list.
ObjLDelete	used to delete an existing list and free the memory.
ObjLAdd	used to insert a new element in the list. The position of the insertion is defined as a parameter of the function.
ObjLLength	used to obtain the current length of the list.
ObjLReadPos	used to read the value of one position in one of the objects of the list. The object and the position to read are defined as parameters of the function.
ObjLCopy	used to duplicate the contents of one list, generating a copy.

A dynamic object can be used to store any data that needs to be updated from different places in the model, using the mentioned functions. Global variables and user-defined attributes are described using dynamic objects (a detailed description is given in Section 10.4). The statistical indicators are described using multiple dynamic objects arranged into a Modelica record. This record stores the values for the mean, the maximum, the minimum and the number of observations (these values are updated during the simulation). A list of objects is also included in the record to store the values of the observations, and be able to calculate the confidence interval for each indicator at the end of the simulation.

9.5 Conclusions

The description of models following the process-oriented approach in Modelica is supported by the SIMANLib and ARENALib libraries. These two libraries have been developed to reproduce some of the main functionalities of the SIMAN language and the Arena simulation environment.

The description of the components of SIMANLib has been performed using the P-DEVS formalism, and implemented using the DEVSLib library. Thus, SIMANLib models communicate using the developed message passing mechanism. As it will be described, ARENALib components are described using combinations of SIMANLib components.

However, the following additional functionalities are required to support the process oriented approach:

- Management of the information that describes the entities in the system.
- Description of an information storage structure to facilitate the management of variable-size data generated during the simulation run (i.e., statistical indicators, global variables and user-defined attributes for the entities).

These functionalities have been implemented in Modelica by means of external libraries written in C code.

10 The SIMANLib Library

10.1 Introduction

The first approach for the development of ARENALib was to write all its components using plain Modelica code [Sanz et al., 2006]. The use of this approach generated large and complex models that were difficult to understand. The development, maintenance, reutilization, and extension of this library was difficult to perform.

The idea then was to divide the actions performed by each ARENALib module into simpler actions, which combined will offer the same functionality as the original module. The same structure can be observed in the Arena environment, where the modules are based and constructed using a lower level simulation language called SIMAN [Pegden et al., 1995]. The P-DEVS formalism was selected to specify the behavior and interaction between ARENALib components, facilitating its description and implementation.

The objective was to reproduce a subset of the modeling functionalities found in the SIMAN language, building a new Modelica library called SIMANLib. The SIMANLib library has been developed to support basic actions and processes, described as atomic P-DEVS models using the DEVSLib library. SIMANLib contains low-level components for discrete-event system modeling and simulation, following the process-oriented approach. These are low-level components

compared to the modules in ARENALib, which represent the high-level modules for system modeling. SIMANLib and ARENALib components can be hierarchically ordered, due to their consistent specification using the P-DEVS formalism. The functionalities included in SIMANLib and ARENALib can be combined to facilitate the description of systems at multiple abstraction levels.

A description of the architecture, components, functionalities and use of the SIMANLib library is discussed in this chapter. A bank teller model is used to present the construction of new models using SIMANLib. A more complex model of a restaurant is discussed to exemplify the modeling functionalities of the library.

10.2 Library Architecture

Components in SIMANLib are divided, as well as in the SIMAN language, in two groups: *blocks* and *elements*. The blocks represent the dynamic part of the system, and are used to describe its structure and define the flow of entities from their creation to their disposal. The elements represent the static part of the system, and are used to model different components such as entities, resources, queues, etc. The use of a formal specification, using the P-DEVS formalism [Sanz et al., 2007], to describe SIMANLib components helps to understand, develop and maintain them. The message passing communication in P-DEVS is equivalent to the communication between blocks in SIMANLib.

The architecture of SIMANLib is shown in Fig. 10.1. Similarly to DEVSLib, it can be considered that the library is divided in two areas: user's area and developer's area. The user's area is composed of:

- The *User's Guide*, that includes the user-oriented documentation.
- The *Blocks* package (shown in Fig. 10.1b), that contains components to describe the flowchart diagram of the system (detailed in Section 10.3).
- The *Elements* package (shown in Fig. 10.1c), that contains models to specify the static data of the system and the characteristics of its elements (detailed in Section 10.4).

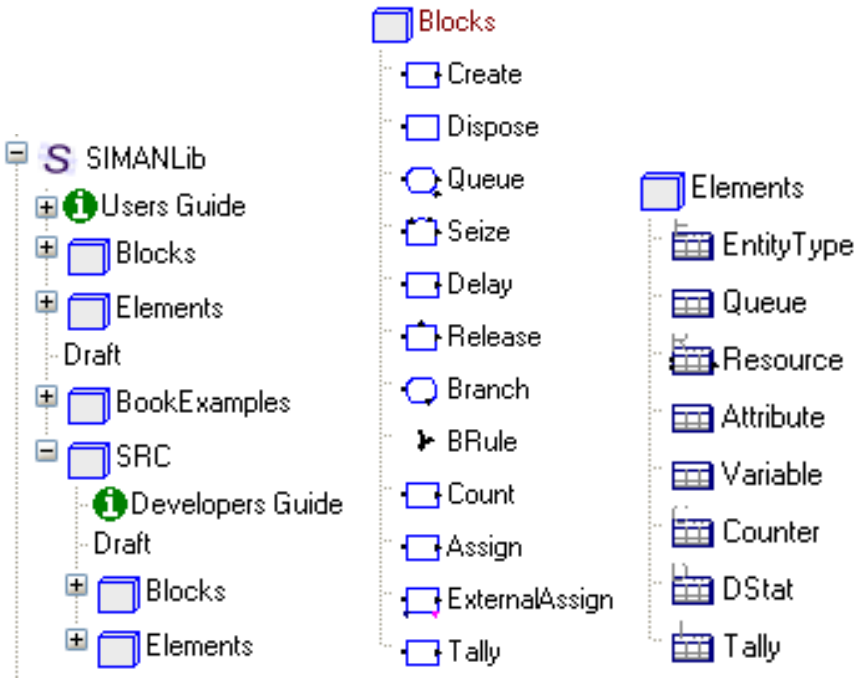


Figure 10.1: SIMANLib library architecture.

– The *Draft* model, that is used as starting point for constructing new process-oriented models using SIMANLib.

– The *BookExamples* package, that contains several case studies described in Pegden et al. [1995]. These examples facilitate the understanding and use of the library. They have been used to validate SIMANLib, comparing the results obtained with the ones obtained simulating the equivalent model in SIMAN/Arena.

The developer's area is encapsulated in the *SRC* package, and contains the developer-oriented documentation and the internal implementation of the components of the library.

10.3 Blocks

The Blocks package in SIMANLib includes models used to describe the flowchart diagram of the system, similarly to the blocks of the SIMAN language. However, due to the complexity and size of the SIMAN language, only some of the basic

blocks are included. The contents of the Blocks package are shown in Fig. 10.1b. The selected blocks correspond to the blocks used to describe the majority of the processes and actions found in logistic systems. This section includes a description of each SIMANLib block.

The description of each block contains a general definition of the represented process (including its interface to communicate with other blocks), the formal specification of the block using the P-DEVS formalism, and an example of behavior on the mostly given situation for the block.

The implementation of the blocks is very close to its P-DEVS definition. Each block is programmed as an atomic DEVSLib model. The transition functions, output and time advance function are programmed following the behavior indicated in the P-DEVS specification of the block.

10.3.1 Create

This block represents a starting point for the flow of entities in the system. The interface of the model is composed of: (1) IN port, used to receive external petitions for entity creation; and (2) OUT port, used to send the created entities. The parameters of this block are: (1) *EntityType* element, used to define the type of the created entities; (2) *Interval*, that describes the amount of time between arrivals (described as a random variate generation function from the RandomLib library, see Chapter 13); (3) *Batch_size*, that represents the number of entities created at each arrival to the system; (4) *Maximum_number_of_batches*, which limits the number of entities created by the block; and (5) *First_creation* time, that sets the simulation time for creating the first entity.

Thus, starting at time equal to *First_creation*, this block creates the *Batch_size* number of entities and sends them to the system. Then, every *Interval* time new entities will be created until the *Maximum_number_of_batches* are created. Each created entity is assigned with a unique identifier or serial number (i.e, *sn*). It can be used to identify a particular entity in the system.

Additionally to this standard behavior, the Create block includes an input port to receive creation petitions. When this input port is connected, the Create

block creates new batches of entities when a new message is received, instead of every *Interval* time. This functionality facilitates the integration of SIMANLib models with other Modelica libraries.

The P-DEVS specification for this block is (when the IN port is not connected):

$X =$ $\emptyset$

$S =$ $\{\text{"}start\text{"}, \text{"}work\text{"}, \text{"}halt\text{"}\} \times N^+$

$Y =$ N^+

$\delta_{int}(phase, n):$ $(\text{"}work\text{"}, BatchSize)$ if $phase == \text{"}start\text{"}$

$(\text{"}work\text{"}, n + BatchSize)$ if $(phase == \text{"}work\text{"})$ and $(n < MaxNumOfBatches)$

$(\text{"}halt\text{"}, MaxNumOfBatches)$ otherwise

$\delta_{ext}(phase, n, e, x):$ nothing since $X = \emptyset$

$\delta_{con}(phase, n, e, x):$ nothing since $X = \emptyset$

$\lambda(phase, n):$ $(\text{"}work\text{"}, n) = send(sn)^{BatchSize}$

$(\text{"}halt\text{"}, n) = \emptyset$

$ta(phase, n):$ $FirstCreation$ if $phase == \text{"}start\text{"}$

$Interval$ if $phase == \text{"}work\text{"}$

∞ otherwise

Where the state contains the current processing situation (i.e., the phase) and the total number of entities generated. The phase can have three values: start, work and halt.

When the IN port is connected, the P-DEVS specification for this block is:

$X =$ N^+

$S =$ $\{\text{"}send\text{"}, \text{"}work\text{"}, \text{"}halt\text{"}\} \times N^+$

$Y =$ N^+

$\delta_{int}(phase, n):$ $(\text{"}work\text{"}, n + BatchSize)$ if $(phase == \text{"}send\text{"})$ and $(n < MaxNumOfBatches)$

$(\text{"}halt\text{"}, MaxNumOfBatches)$ otherwise

$\delta_{ext}(phase, n, e, x)$: ("$send$", n)

$\delta_{con}(s, e, x)$: $\delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(phase, n)$: ("$send$", n) = $send(sn)^{BatchSize}$

("$halt$", n) = $\emptyset$

$ta(phase, n)$: 0 if ($phase$ == "$send$")

∞ otherwise

10.3.2 Dispose

Opposite to the previous block, the Dispose block represents the final point for entities in the system. Each entity, or group of entities, arrived to a Dispose block is deleted and removed from the system. The interface of the model is only composed of the IN port, used to receive entities. This block does not have parameters.

The P-DEVS specification for this block is:

X = N^+

S = {"$wait$", "$busy$"}

Y = $\emptyset$

$\delta_{int}(phase)$: ("$wait$")

$\delta_{ext}(phase, e, x)$: ("$busy$")

$\delta_{con}(s, e, x)$: $\delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(phase)$: nothing since $Y = \emptyset$

$ta(phase)$: ∞ if $phase$ == "$wait$"

0 otherwise

10.3.3 Queue

The Queue block defines a temporal storage place for entities waiting to be processed. The interface of the model is composed of: (1) IN port used to receive new entities; (2) OUT port used to connect with a Seize block; and (3) BALK port used to discard entities when the maximum capacity of the queue is reached.

The parameters for this block are: (1) *BalkConnected*, that indicates if the BALK port is connected with another block or not; (2) *Queue* element, that indicates the Queue element assigned to this block; and (3) *Capacity*, that sets the storage capacity of the queue (a zero value indicates infinite).

Any arrived entity is inserted in the queue (following a certain policy, like FIFO, LIFO, etc., defined by the associated Queue element) and waits for available resources. If the maximum capacity of the queue is reached, the new entity is immediately sent through the BALK port (or removed from the system it the port is not connected). The OUT port of a Queue block must be connected to a Seize block. When the entity is inserted in the queue, the Queue block sends a message to the Seize block that represents a "petition of resources", required to process the entity. Thus, no entities are transferred between these two blocks. The transmitted message carries the information as follows: the type of the message transports the current size of the queue, and the value of the message transports a reference to the storage of entities (which is a dynamic list of entities, as described in Section 9.3). The management performed by the Seize block is detailed below in this section. When the required resources are available, the Seize block extracts an entity from the queue (using the reference transmitted in the message). After that, the entity continues through the flowchart-diagram of the system.

The P-DEVS specification for this block is:

$$
\begin{aligned}
X &= && N^+ \\
S &= && \{\text{"}wait\text{"}, \text{"}send\text{"}, \text{"}balk\text{"}\} \times N^+ \\
Y &= && N^+ \\
\delta_{int}(phase, n) : && && (\text{"}wait\text{"}, n) \\
\delta_{ext}(phase, n, e, x) : && && (\text{"}send\text{"}, n+1) \text{ if } n < Capacity \\
&&&& (\text{"}balk\text{"}, n) \text{ if } n == Capacity \\
\delta_{con}(s, e, x) : && && \delta_{ext}(\delta_{int}(s), 0, x)
\end{aligned}
$$

$$\lambda(phase, n): \quad send(petition) \text{ if } phase = \text{"}send\text{"}$$
$$balk(sn) \text{ if } phase = \text{"}balk\text{"}$$
$$ta(phase, n): \quad \infty \text{ if } phase = \text{"}wait\text{"}$$
$$0 \text{ otherwise}$$

Where the state represents the phase (i.e., waiting for an arrival, sending a resource petition or balking an entity) and the number of elements in the queue. The $send(petition)$ function is used to send a resource petition to the Seize block. The $balk(sn)$ function is used to balk an entity.

10.3.4 Seize

This block represents the operation of seizing a resource to perform some process to an entity waiting in queue. The interface of the model is composed of: (1) IN port, used to receive resource petitions from the Queue block; (2) OUT port, used to send the entities that captured the resource; (3) S port, used to redirect resource petitions to the Resource element; and (4) R port, used to receive the confirmation of a seized resource from the Resource element. The parameters of the block are: (1) *Priority* of selecting the entity to seize the resources between the ones waiting for them; and (2) *ResourceUnits* to be seized (the resource units can not be seized partially, but in a whole set).

The Seize block receives resource petitions through the IN port (which must be connected to a Queue block). The Seize block must also be connected to a Resource element using the S and R ports. The resource petitions are redirected to the Resource element through the S port. The message transmitted with the redirected petition contains the following information: the number of resource units to seize, and the reference to the dynamic queue where the entities are stored. The number of resource units is transmitted using the Type variable of the message. The reference to the dynamic queue is transmitted using the Value variable of the message. These resource petitions are ordered by the Priority assigned to the Seize block. The reference to the queue is used to identify the queue to extract entities when the resource becomes available, since multiple

Queue/Seize blocks can be connected to the same Resource element. When the resource becomes available, the Resource element sends a confirmation message to the R port of the Seize block. Then, the Seize block extracts the first entity in the Queue (i.e., using the reference to the dynamic queue in memory), and sends it through its OUT port.

The P-DEVS specification for this block is:

$X =$ N^+

$S =$ $\{"wait", "seize", "send"\}$

$Y =$ N^+

$\delta_{int}(phase):$ $("wait")$

$\delta_{ext}(phase, e, x):$ $("seize")$ if $x.port == IN$

$("send")$ if $x.port == R$

$\delta_{con}(s, e, x):$ $\delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(phase):$ $send(petition)$ if $phase = "seize"$

$send(sn)$ if $phase = "send"$

$ta(phase):$ ∞ if $phase = "wait"$

0 otherwise

Where the state shows the current situation of the block (i.e., waiting for an arrival, redirecting seize petitions, or sending an entity that already captured the resource). The $send(petition)$ function generates a new message that contains a new seize petition and sends it to the Resource element. The $send(sn)$ function extracts an entity from the queue and sends it to the next block.

10.3.5 Delay

The Delay block represents the time an entity spends being processed. The interface of the model is composed of: (1) IN port, used to receive entities; and (2) OUT port, used to send the processed entities. As parameters, it only has the *Duration* that represents the fuction used to calculate the elapse of time the

entity is being delayed (corresponds to a random variate generation function from the RandomLib library).

Each entity that arrives to a Delay block is assigned with a delay interval. The length of this interval is computed using the function designed with the *Duration* parameter. Once the entity has its delay interval, it is inserted in an internal queue for delayed entities. The order of insertion is calculated depending on the time to leave the block for each entity. So, the first entity in the queue will be the first to leave the Delay block. When the simulation time reaches the time to leave for the first entity in the block, this is extracted from the queue and sent to the next block in the diagram. Multiple entities could have the same time to leave, and thus, this situation is evaluated and, if necessary, multiple entities are extracted form the queue.

The P-DEVS specification for this block is:

$X =$ N^+

$S =$ $\{(\Re^+ \times N^+)\}$

$Y =$ N^+

$\delta_{int}(list):$ $(\{(t_{m+1}, sn_{m+1}), \ldots, (t_n, sn_n)\})$ if $list == \{(t_1, sn_1), \ldots, (t_m, sn_m), \ldots, (t_n, sn_n)\}$ and $\forall i = 1..m | t_i \leq time$

$\delta_{ext}(list, e, x):$ $(\{(Duration, sn)\})$ if $list$ is empty

$order((Duration, sn), list)$ otherwise

$\delta_{con}(s, e, x):$ $\delta_{ext}(\delta_{int}(s), 0, x)$

$\lambda(list):$ $send(sn_i)$ if $list == \{(t_1, sn_1), \ldots, (t_m, sn_m), \ldots, (t_n, sn_n)\}$ and $\forall i = 1..m | t_i \leq time$

$ta(list):$ ∞ if $list$ is empty

$t_1 - time$ if $list == \{(t_1, sn_1), \ldots, (t_n, sn_n)\}$

Where the state of the block contains the list of entities being processed. The *time* is a global variable in the system, common to all the models. The *order*() function is used to insert a new element in the list, ordered by the duration of its delay.

10.3.6 Release

This block is used to release the resources previously seized by an entity. This operation is the opposite to the one performed by the Seize block. The interface of the model is composed of: (1) IN port, used to receive entities; (2) OUT port, used to send the entities after releasing the resource; and (3) R port, used to send release petitions to the Resource element. This block has as parameter: the *ResourceUnits* that have to be released.

Each Release block must be connected to a Resource element using the R port. When the block receives an entity, it sends a release petition to the Resource element connected to the R port. The release petition is represented by a message where the type contains the *ResourceUnits* to be released. The entity is immediately sent through the OUT port to the next block in the diagram.

The P-DEVS specification for this block is:

$X =$	N^+
$S =$	$\{"wait", "release"\} \times \Re^+$
$Y =$	N^+
$\delta_{int}(phase, sn)$:	$("wait", 0)$
$\delta_{ext}(phase, sn, e, x)$:	$("release", x.value)$
$\delta_{con}(s, e, x)$:	$\delta_{ext}(\delta_{int}(s), 0, x)$
$\lambda(phase, sn)$:	$send(releaseUnits)$ and $send(sn)$ if $phase == "release"$
$ta(phase, sn)$:	∞ if $phase == "wait"$ 0 if $phase == "release"$

Where the state is composed of phase (that indicates if the model is waiting for new entities or sending a release petition to the resource element) and the serial number of the last received entity (required to send it using the output function).

10.3.7 Branch and BranchRule

The Branch and BranchRule blocks can be used to divide the flow of entities in the model, depending on certain conditions. SIMAN defines this behavior only with the Branch block. But, in order to allow a variable number of rules for selecting entities, it has been divided in two blocks in SIMANLib. The P-DEVS specification of these, and the following blocks described in this section, is not included because their management of the entities is equivalent to a Delay block with a zero delay-time. The actions performed by each block are included within the external transition function.

The Branch block must be followed by, at least, one BranchRule block. It receives entities and sets a new private attribute, named *maxNumberOfBranches*, whose value is used to limit the number of rules evaluated for the entities across the branch (its value must be at least 1). After that, the received entity is sent to the first BranchRule.

Each BranchRule block represents a particular condition to divide the flow of entities. The interface of the model is composed of: (1) IN port, used to receive entities from the Branch or other BranchRule blocks; (2) OUT port, used to send selected entities; and (3) NEXT port, used to connect with other BranchRule blocks (this port can be connected or not, depending on the required behavior). The conditions for entity flow division of the BranchRule block can be of three types:

1. IF type, that selects the entity if a given boolean condition becomes true,
2. WITH type, that selects the entity following a given probability, and
3. ELSE type, that always selects the entity (this rule is usually located as the last BranchRule block).

When an entity arrives to a BranchRule block, it is evaluated following the defined type of condition. If the entity is selected after evaluating the condition, the block sends it through its OUT port and decrements in one unit the value of the *maxNumberOfBranches* attribute. A duplicate of the entity is sent to the

next BranchRule if the *maxNumberOfBranches* is still greater than zero. If the entity is not selected in this BranchRule, it is sent through the NEXT port (or removed from the system if the port is not connected).

10.3.8 Assign and ExternalAssign

The Assign block represents a point to define the value of the global variables of the system or the user-defined attributes of the entities. Global variables or user-defined atributes are represented using dynamic objects, and thus they are bi-dimentional matrices of real numbers. The parameters of the Assign block are: (1) *variable or attribute* to assign with a new value (represented by a Variable or Attribute element); (2) *position* (row and column) of the attribute or variable; and (3) *value* to assign. When an entity arrives to the block, the *value* is assigned to the indicated *position* of the *variable* or *attribute*. The entity is immediately sent to the next block in the diagram.

Additionally, SIMANLib includes another block called ExternalAssign. The ExternalAssign block includes two additional ports in its interface: Y and CHANGE ports. The value of the Y port corresponds to the value assigned to the attribute or variable each time an entity arrives to the block. Thus, Y is a discrete-time variable of Real type. The boolean value of the CHANGE port switches every time a new entity arrives to the block, and so, the value of Y could have changed. This block can be used to communicate the process-oriented part of a system (described using SIMANLib) with a continuous-time part of the system (described using any other Modelica library).

The ExternalAssign block is described as a P-DEVS coupled model. The internal structure of the model is shown in Fig. 10.2. The SIMANLib Assign block is combined with the SetValue, DUP and DICO models from the DEVSLib library in order to construct this model.

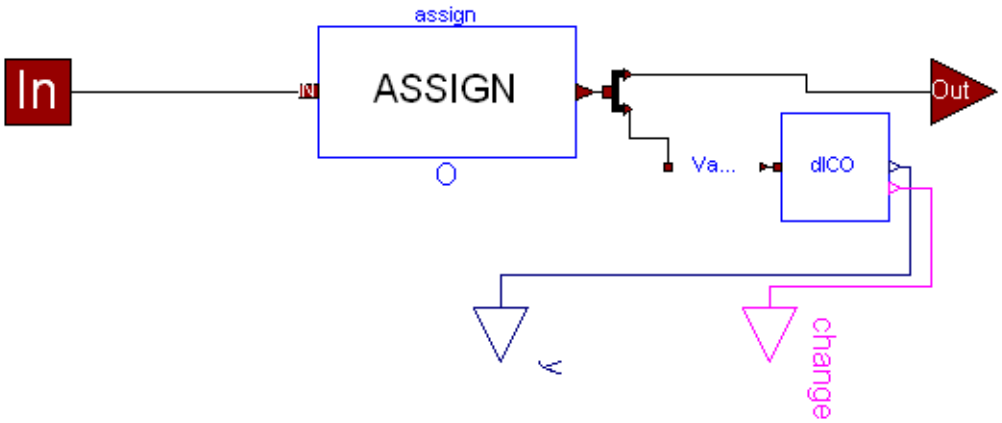


Figure 10.2: SIMANLib ExternalAssign block.

10.3.9 Count

The Count block is used to account the flow of entities in one point of the flowchart-diagram of the system. The parameters of the block are: (1) *Counter* element, that records the value of the count; and (2) *Increment*, used to modify the counter. When a new entity arrives to the Count block, the value stored in the *Counter* element is modified by the *Increment* value (increased or decreased, depending on its sign). The entity is immediately sent to the next block of the diagram.

10.3.10 Tally

The Tally block is used to record a time-dependent statistical indicator. The parameters of the block are: (1) *Value*, that represents a variable used to obtain new observations for the indicator; (2) *Tally* element, that records the observations and values calculated by this block. The *Tally* indicator contains variables to store the average, maximum, minimum, number of observations and last values of the observations. When a new entity arrives to the Tally block, a new observation is obtained from the *Value* variable. Using the value of the new observation, the variables stored in the indicator are updated. The entity is immediately sent to the next block of the diagram.

10.4 Elements

The Elements package contains the components to describe the static information of a process-oriented model. This information corresponds to the types of entities in the system, the characteristics of the queues and resources, the global variables, the attributes and the statistical indicators. As mentioned in the description of some SIMANLib blocks, some of the elements have to be associated with blocks in the flowchart diagram.

The contents of the Elements package are shown in Fig. 10.1c. Each element is described using a Modelica record, that contains the variables to store the information about the element, and a set of functions to manage the information contained in the record. The characteristics and use of each element in the package are described in this section.

10.4.1 EntityType

The EntityType element represents a type of entities that could arrive to the system (e.g., customers, parts, pieces, etc.). This element has to be associated with a Create block. The variables contained in the EntityType element are:

Id	Identification number for the entity type.
HoldCostRate	Cost rate of processing the entity in the system
VACost	Cost of Value Added processes applied to the entity.
NVACost	Cost of Non-Value added processes applied to the entity
WaitCost	Cost of Waiting processes (delays, queues, etc.)
TranCost	Cost of Transport processes (transporters, conveyors, etc.)
OtherCost	Other costs associated with the entity.

These variables are used to initialize the fields *Type*, *HoldCostRate*, *VACost*, *NVACost*, *WaitCost*, *TranCost* and *OtherCost* of an entity (see Table 9.1) when created by the Create block.

The set of functions associated with the EntityType element are used to manage entities in general, and not only this particular element. The included functions are:

NewEntity()	Creates a new entity with a new unique serial number. This function uses the ECreate function to allocate the memory for the new entity.
Duplicate()	Duplicates an existing entity, setting its attribute *Primary* to 0 (which indicates that the entity is a copy).
ECreate()	Allocates memory for a new entity and initializes its attributes.
EDelete()	Removes an existing entity from memory.
EGet()	Reads the value of one of the attributes (or fields) of the entity.
ESet()	Sets a new value of one of the attributes of the entity.
ERead()	Reads all the attributes of an entity simultaneously.
EUpdate()	Updates all the attributes of an entity simultaneously.

10.4.2 Queue

The Queue element represents the characteristics of a queue in the system. This element has to be associated with a Queue block. The variables contained in the Queue element are:

Ranking	Defines the policy used to order the entities in the queue.
AttrNum	Identifies the number of the attribute to use with the LVF and HVF rankings

The Ranking can be of the following types: FIFO, LIFO, LVF (Lower Value First), and HVF (Higher Value First). The LVF and HVF rankings use the value of the attribute identified with the AttrNum variable to order the entities in the queue. The lowest value will be first in the LVF ranking, and the highest value will be first in the HVF policy. Since the management of these variables is

performed by the Queue block, the Queue element does not have any associated function.

10.4.3 Resource

The Resource element represents the resources available in the system that can be used to process entities. Each resource is divided into resource units, which can be seized by the entities.

Multiple processes can share the same resources. The Resource element has been implemented as an atomic P-DEVS model that receives seize and release petitions for the represented resources, and sends confirmations for the captured resources. In this way, multiple Seize and Release blocks could be connected to the same resource element (representing processes that share a resource).

The interface of the model is composed of: (1) S port, used to receive seize petitions from the Seize blocks; (2) R port, used to receive release petitions from the Release blocks; and (3) O port, used to send the confirmation messages to the Seize blocks for the captured resources. The parameter of this block is the *Capacity* of the resource (i.e., the number of available resource units).

The behavior of the model is as follows. When the Resource receives a seize petition containing the number of resource units requested, the model checks if the petition can be satisfied. If so, the model seizes the required units and sends a confirmation message through the O port. The confirmation message will be received by the Seize block, extracting the entity from the queue. If the seize petition can not be satisfied (i.e., not enough resource units are available), the seize petition is stored in a FIFO queue awaiting for released resources. When the Resource model receives a release petition, immediately releases the number of resource units contained in the petition and checks if any of the awaiting seize petitions can be satisfied with the currently available resources.

10.4.4 Objects, Attributes and Variables

The Object element represents user-defined information that is required in the description of the system. This is, user-defined attributes for the entities or global variables of the system. Objects are implemented as dynamic objects (i.e., dynamic two-dimensional matrices of real numbers, described in Section 9.4). The variables contained in the Object element are:

Number	Identify the Object in the system. A different number has to be assigned to each Object in the system.
Rows	Defines the number of rows of the matrix.
Cols	Defines the number of columns of the matrix.
InitialValue	Defines the initial value for the Object.
objType	Defines if the object corresponds to an user-defined attribute (value == 1) or a global variable (value == 0).
P	Contains a reference to the dynamic object in memory.

The functions associated with the Object element are the same as the functions included to manage dynamic objects (see Section 9.4). The initial value for the Object is set at the beginning of the simulation using the InitialValue variable and the ObjUpdate function. At the end of the simulation, the memory allocated for the object is freed.

The Attribute element represents a user-defined attribute, which is assigned to the entities using an Assign block. The value of an attribute can be different for each assigned entity. The Attribute element is implemented extending the Object element, and setting its objType to 1. Like the Objects, each attribute in the model must have a unique number assigned, which will be used to identify it (e.g., using it for ordering the LVF or HVF rankings in a Resource element). The functions associated with the Attribute element are:

get()	Reads the value of an attribute in a particular entity.
set()	Sets a new value for an attribute in a particular entity.

The Variable element represents a global variable in the model. Like the Attribute element, the Variable is implemented extending the Object element and setting its objType to 0. The functions associated with the Variable element are:

get()	Reads the value of the variable.
set()	Sets a new value for the variable.

10.4.5 Counter

The Counter element represents a counter that can be increased or decreased using a Count block. Since a Counter can be modified from multiple Count blocks in the same model, it has been implemented using a dynamic object. The variables contained in the Counter element are:

Name	Defines a user-meaningful name for the counter (e.g., "NumArrivals").
Limit	Defines the simulation limit for the counter (if the limit is reached, the simulation terminates).
OutFile	Defines the name of the output file to write the simulation results ("SIMANLIB_RESULTS.txt" by default).
P	Contains a reference to the dynamic object in memory.

The functions associated with the Counter element are:

get()	Reads the value of the counter.
set()	Sets a new value for the counter.

The value of the Counter is managed by its associated Count blocks, using the mentioned functions. At the end of the simulation, the Counter element writes in the OutFile its name and its final value.

10.4.6 DStat

The DStat element represents a discrete (time-independent) statistical indicator for a discrete variable in the system. The observations of the indicator are ob-

tained when the selected variable changes its value. The element records the last, average, minimum, maximum and number of observation values for the selected variable. The value of each observation is also stored, in order to calculate the confidence interval at the end of the simulation. The DStat elements are not associated with any block of the flowchart diagram. The variables contained in the DStat element are:

Name	Defines a user-meaningful name for the DStat (e.g., "NumWaiting").
Expression	Defines the expression used as discrete variable to observe (it can correspond to a single variable or to a combination of multiple variables (e.g., $queue1.NQ + queue2.NQ$).
OutFile	Defines the name of the output file to write the simulation results ("SIMANLIB_RESULTS.txt" by default).
PLastValue	Contains a reference to the dynamic object used to record the last observed value.
PMean	Contains a reference to the dynamic object used to record the average value.
PMinimum	Contains a reference to the dynamic object used to record the minimum value.
PMaximum	Contains a reference to the dynamic object used to record the maximum value.
PObservations	Contains a reference to the dynamic object used to record the number of observations.
PValues	Contains a reference to the dynamic object used to record the list of observed values.

The functions associated with the DStat element are:

DStatUpdate()	Updates the DStat when a new observation is performed.
DStatVariation()	Calculates the confidence interval using the list of observed values.

When the selected variable, or expression, changes its value, a new observation is performed. DStat element uses the DStatUpdate() function to update the recorded values. At the end of the simulation, the DStat calculates the confidence interval using the DStatVariation() function and writes the results to OutFile.

10.4.7 Tally

The Tally element represents a time-dependent statistical indicator. This is, an statistical indicator calculated from a variable whose observations are obtained at certain points in time independently of the changes in its value. This element has to be associated with a Tally block. The variables contained in the Tally element are:

Name	Defines a user-meaningful name for the Tally (e.g., "NumDispached").
OutFile	Defines the name of the output file to write the simulation results ("SIMANLIB_RESULTS.txt" by default).
PLastValue	Contains a reference to the dynamic object used to record the last observed value.
PMean	Contains a reference to the dynamic object used to record the average value.
PMinimum	Contains a reference to the dynamic object used to record the minimum value.
PMaximum	Contains a reference to the dynamic object used to record the maximum value.
PObservations	Contains a reference to the dynamic object used to record the number of observations.
PValues	Contains a reference to the dynamic object used to record the list of observed values.

The functions associated with the Tally element are:

TallyUpdate()	Updates the Tally when an entity crosses the Tally block.
TallyVariation()	Calculates the confidence interval using the list of observed values.

The Tally block calculates the statistical indicator during the simulation, depending on the flow of entities across it. The Tally element writes the results to the OutFile at the end of the simulation.

10.5 Model Construction Using SIMANLib

The construction of process-oriented models using the SIMANLib library follows the following steps.

1. Create the new blank model by duplicating the *Draft* model included in the library.

2. Describing the flowchart diagram of the system using interconnected SIMANLib blocks. Since the Resource elements must be connected to the Seize and Release blocks of the flowchart diagram, they should be also included in this step.

3. Describe the static information of the model by including the required SIMANLib elements.

4. Finally, configure the parameters of the included blocks and elements to represent the desired simulation experiment.

These steps are detailed with the construction of a simple model: a *bank teller*. In this model, the customers arrive to the bank and wait their turn in the queue. If the teller is idle, the customer is served immediately. Otherwise, the teller will serve the first customer in the queue. When finished, the customer leaves the bank and the teller serves another customer if anyone else is waiting, or waits for a new arrival.

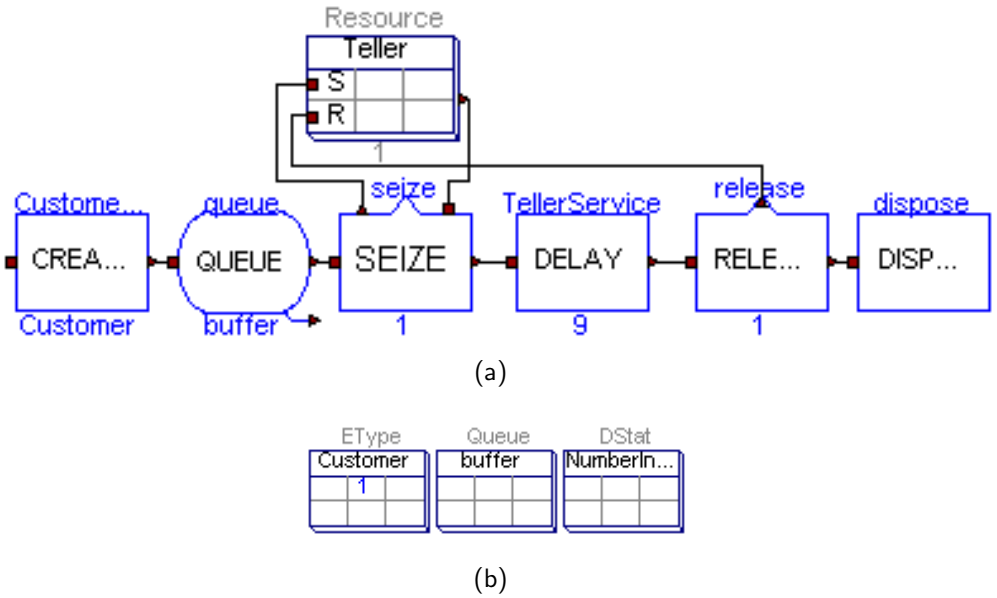


Figure 10.3: Bank teller system modeled using SIMANLib: a) flowchart diagram (blocks); and b) static data (elements).

The first step will be to duplicate the Draft model, creating the model called BankTeller. The second step is to describe the flowchart diagram of the system. This is performed by including and connecting, using drag and drop, the blocks and Resource element shown in Fig. 10.3a. The included blocks are: Create (that represents the arrival of customers), Queue, Seize (that together with the Release manages the availability of the teller), Delay (that represents the delay due to the service time), Release and Dispose (that represents the departure of customers).

The third step is to describe the static information of the system. This is performed by including, also using drag and drop, the required elements (the ones associated with the included blocks, like the EntityType and the Queue, and the other required to calculate statistical indicators, like the DStat). The included elements are (shown in Fig. 10.3): Resource (that represents the teller), EType (that represents the customers), Queue (that describes the organization of the queue) and DStat (that calculates the statistics for the number of customers in queue).

The final step is to configure the parameters of the included components. In this case, the inter-arrival time for customers is set to an exponential(10) distribution, the capacity of the queue is infinite, the number of resources to seize

Table 10.1: Bank teller system simulation results using SIMANLib and SIMAN.

Model	Avg. in Queue	Half-Width
SIMANLib	3.3073	-
SIMAN	3.2089	0.22

and release for each entity is 1, the capacity of the resource is also 1 and the delay time is set to an exponential(8) distribution.

The configuration for this experiment represents an M/M/1 queue system, with an analytical result of 3.2 for the average number of customers in the queue. The statistical indicator for the number of customers in queue is automatically calculated during the simulation run by the DStat element. The results after simulating the system during 10^6 time units using SIMANLib and SIMAN are equivalent (shown in Table 10.1, including the Half-Width interval calculated using SIMAN). The evolution of the number of customers in queue during the simulation is shown in Fig. 10.4. The average of customers in the queue performs fast changes during the beginning of the simulation, but later approaches the 3.2 value that matches with the mentioned analytical solution.

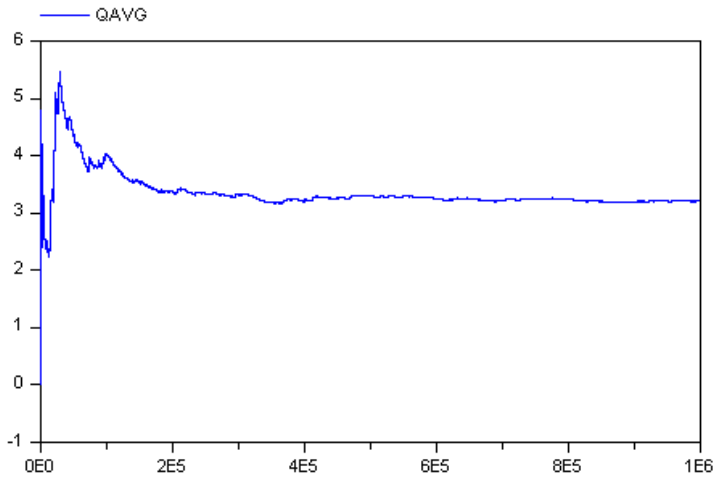


Figure 10.4: Number of customers in queue for the bank teller system modeled using SIMANLib.

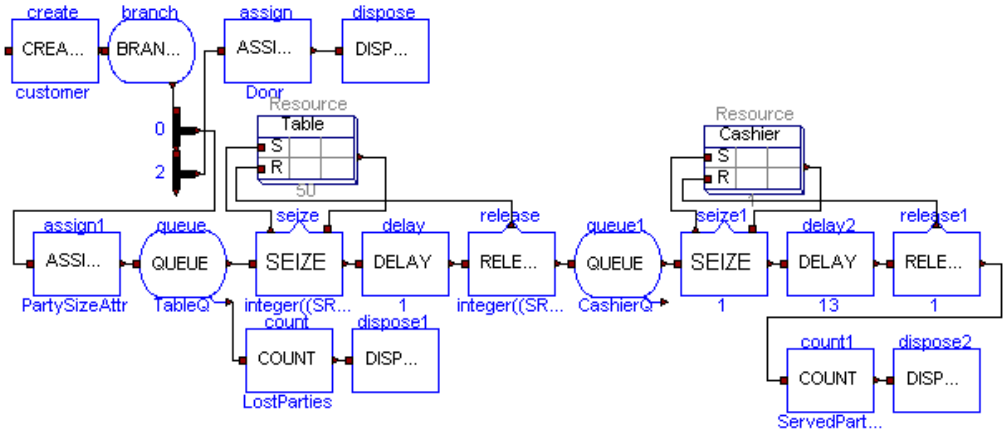


Figure 10.5: Restaurant modeled using SIMANLib.

10.6 Modeling a Restaurant Using SIMANLib

As a case study, the restaurant model described in Pegden et al. [1995] has been composed using SIMANLib. It describes a more complex system, in comparison with the bank teller system, with several processes and a division in the flow of entities.

The behavior of the system is as follows. Customers arrive in groups from 2 to 5 persons and wait for an available table. If there are already 5 groups waiting, the new group leaves the restaurant without waiting. The restaurant has 50 tables. Each table is for two persons, so several tables may be needed for each group. When seated, the group is served and eats. At the end, the group pays the check to the cashier and leaves. The restaurant receives customers from 5 p.m. to 9 p.m., and, after that, waits until all the customers leave.

The flowchart diagram of the constructed model is shown in Fig. 10.5. The arrival of customers has been described using a Create block. This block generates entities, that represent groups of customers, following an exponential distribution. The maximum number of batches generated by this block is set to the value of a global variable named *Door*, and defined using a Variable element. The initial value of *Door* is set to infinity.

Each generated group of customers arrives to a Branch block, with two BRule blocks, used to represent the door of the restaurant. The first BRule checks if

the simulation time is lower than or equal to 240 minutes (that represent the four opening hours of the restaurant). If so, the entity goes to the Assign block where a new attribute is assigned, named *PartySizeAttr*, that represents the size of the group (randomly selected from 2 to 5). Otherwise, the entity goes to the Assign block that sets the value of the *Door* variable to 0 and after that is disposed. The change in the value of the *Door* variable is used to represent the clousure of the restaurant, and prevents from creating more groups of customers because the maximum number of batches in the Create block is also set to 0.

Once a group has entered the restaurant and its size has been assigned, they wait in the Queue block for an available table. If there are already five entities in the queue, the new entity is balked. Balked entities are counted using a Count block, and disposed from the system. The Seize block associated to the Queue calculates the number of tables to seize depending on the size of the arrived group, and sends the resource petition to the Resource element that represents the tables (a resource with 50 available units). Once the tables are seized, the Seize element extracts the group from the queue and sends it to the Delay block that represents the ordering and eating process.

When the group has finished eating, they leave the tables (releasing them using a Release block connected to the Resource element), and go to pay the check. A new Queue is included to represent the groups waiting for paying at the cashier. The cashier is represented using a Resource element with only one available unit. Groups have to seize the cashier (using a Seize block), pay (represented using a Delay), leave the cashier (releasing the seized resource in a Resource block) and finally leave the restaurant. A Counter block is used to account the number of groups served during the day. Four DSTat elements are used to record statistics for the number of groups waiting for tables, the utilization of the tables, the number of groups waiting for the cashier, and the utilization of the cashier.

In order to analyze the system, 30 independent simulation runs, each of 480 minutes, have been performed. Each run records statistics about the number of customers served, the number of busy tables, the number of waiting customers, the number of groups that left without entering and the utilization of the cashier.

Table 10.2: Restaurant simulation results, comparing SIMANLib and SIMAN (in average values).

Indicator (avg.)	SIMANLib	SIMAN
groups served	136.13	135.43
groups lost	15.83	14.00
busy tables	24.49	24.25
groups waiting	0.62	0.72
cashier util.(%)	42.51	41.94

The simulation results after the simulation of the system using SIMANLib and SIMAN are equivalent. These results are shown in Table 10.2. They show a poor utilitation of the resources of the restaurant, where only the half of the tables are used (in average) and the cashier is also idle the half of the time. However, there are still some groups lost, due to the restriction in the queue for waiting an available table (a maximum of five groups can wait in the queue).

10.7 Conclusions

A new library, named SIMANLib, for process-oriented modeling in Modelica has been designed and developed. This library reproduces some of the functionalities for process-oriented modeling found in the SIMAN language. It includes the Create, Dispose, Queue, Seize, Delay, Release, Assign, Branch, Count and Tally blocks, as well as the EntityType, Resource, Queue, Variable, Attribute, Counter, Tally, and DStat elements.

The design of the library has been performed using the P-DEVS formalism to describe the behavior of its components. The use of a mathematical formalism to describe the behavior of the components facilitate the development and maintenance of the library. The implementation of these components has been performed using the DEVSLib library. Thus, SIMANLib models communicate using the developed message passing mechanism.

11 The ARENALib Library

11.1 Introduction

ARENALib provides high-level functionalities for modeling systems following the process-oriented approach. It reproduces some of the modeling functionalities of the Basic Process panel of the Arena simulation environment. The development of the included components, their characteristics and use are detailed in this chapter.

Similarly to Arena, ARENALib is constructed using a combination of the functionalities provided by the SIMANLib library. Components in ARENALib, named modules, are described as coupled P-DEVS models and implemented using SIMANLib blocks and elements. The use of the P-DEVS formalism facilitates the development, maintenance, reuse and extension of both libraries.

11.2 Library Architecture

The general architecture of the library is shown in Fig. 11.1a. Analogously to SIMANLib, ARENALib is divided in two areas: user's area and developer's area. The user's area is composed of the *User's Guide* (that contains the user-oriented documentation of the library), the *Draft* model (used to create new models), the *BasicProcess* package (that contains the modules, used as components to

construct process-oriented models), the *Examples* package (that contains some discrete-event system examples) and the *BookExamples* package (that contains models of systems described in Kelton et al. [2007]). The developer's area is encapsulated into the *SRC* package and contains the internal implementation of the library modules.

ARENALib models are described using a flowchart diagram and some additional information associated to the processes of the diagram. Flowchart diagrams are composed using *flowchart modules*, which are similar to SIMANLib blocks. However, in ARENALib the flowchart modules perform more complex actions than SIMANLib blocks and already include the calculation of several statistical indicators. The static information of the model is described using *data modules*, which are equivalent to some of the SIMANLib elements. The structure of the BasicProcess package, that contains the flowchart and data modules is shown in Fig. 11.1b.

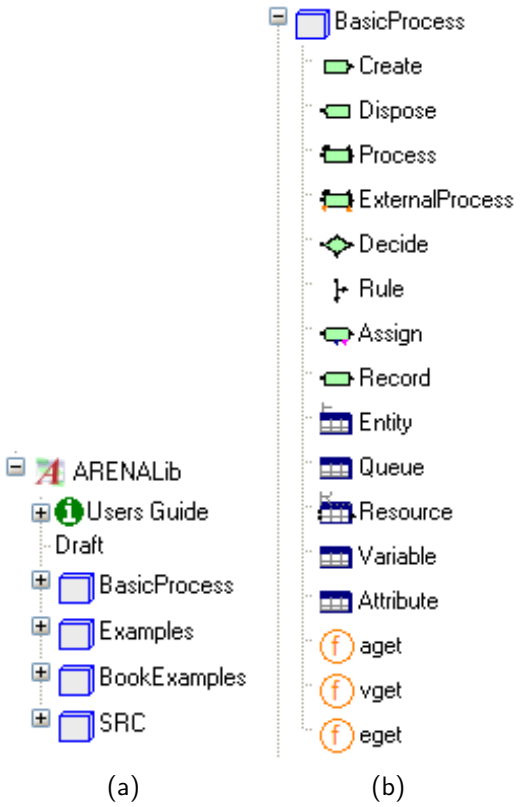


Figure 11.1: ARENALib library: a) general architecture; b) detail of the BasicProcess package.

11.3 Flowchart Modules

The internal description of each flowchart module included in ARENALib is included in this section.

11.3.1 Create

The Create module represents, like the SIMANLib Create block, the starting point for the flow of entities in the system. Thus, it is constructed using a Create block and a counter that records the number of entities created by the module. The internal implementation of the module is shown in Fig. 11.2.

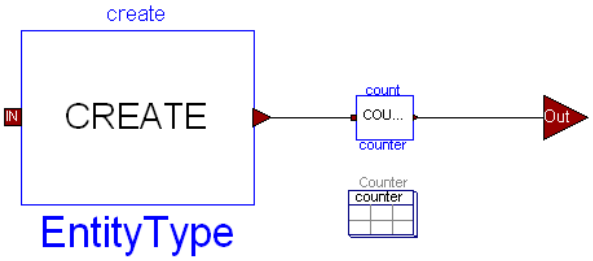


Figure 11.2: ARENALib Create module.

11.3.2 Dispose

Similar to the Create module, the Dispose module is equivalent to the SIMANLib Dispose block. It represents the final point for the flow of entities in the system. The Dispose module is constructed using a Dispose block and a counter that records the number of entities disposed by the module. The internal implementation of the module is shown in Fig. 11.3.

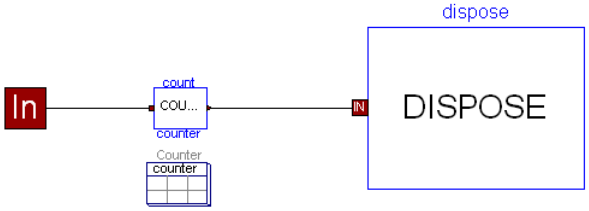


Figure 11.3: ARENALib Dispose module.

11.3.3 Process

The Process module represents any process that can be applied to the entities in the system. Processes can be of the following types: (1) *delay* (that represents a time delay for the entities, like the Delay block), (2) *seize-delay* (that forces the entity to capture a resource and be delayed), (3) *delay-release* (forces a delay for the entity and the release of a previously seized resource), and (4) *seize-delay-release* (represents an entity seizing a resource, being delayed and at the end releasing the resource). The type of process to perform is selected using one of the parameters of the module. The duration of the delay is calculated using a probability distribution function from the RandomLib library.

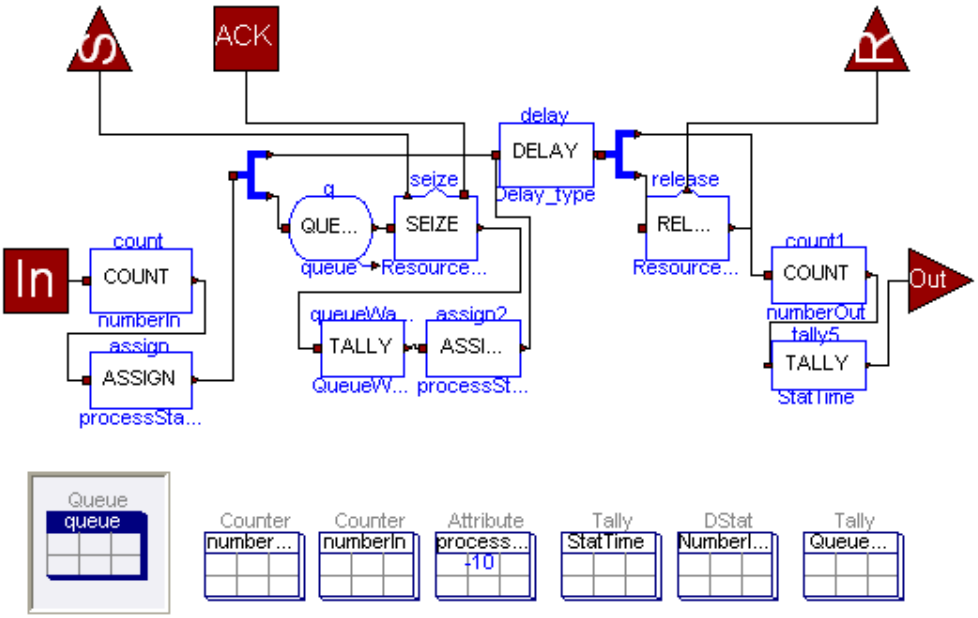


Figure 11.4: ARENALib Process module.

The internal implementation of the module is shown in Fig. 11.4. It can be observed that multiple SIMANLib blocks are required to represent the behavior of the process. The module also includes the Queue element required for the Seize block, and the elements required to automatically calculate the following statistical indicators:

– The number of entities that entered and left the module.

– A Tally indicator for the time the entities are processed.

– A Tally indicator for the time spent waiting in queue.

– A DStat indicator for the number of entities waiting in queue.

Also notice that two Select models, from the DEVSLib library, are used to select the type of process to perform.

11.3.4 ExternalProcess

The ExternalProcess represents a process applied to the entity, whose duration (i.e., the delay time) is externally modeled instead of calculated using a probability distribution. The ExternalProcess is equivalent to the Process module, where the Delay block is substituted by an external model.

The internal implementation of the module is shown in Fig. 11.5. Notice that the Delay block in the middle of the Process (see Fig. 11.4) has been substituted by four models and two additional ports: the DICO, RealToInteger, IntegerToReal and Quantizer models, and the entityStart and entityEnd ports.

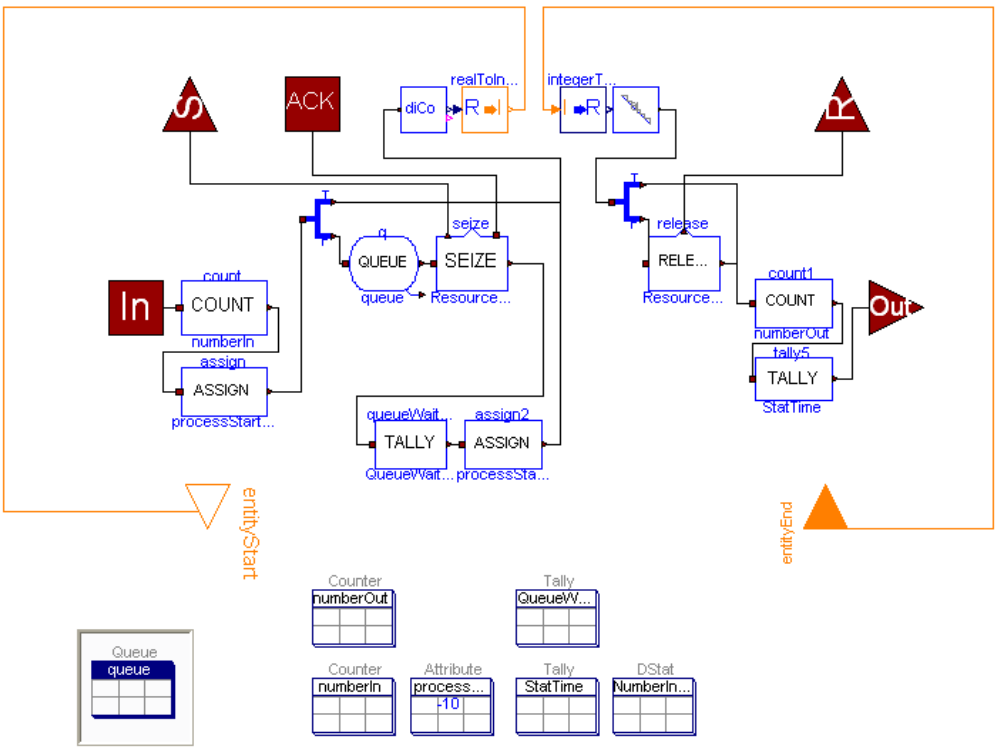


Figure 11.5: ARENALib ExternalProcess module.

When an entity has to be delayed in an ExternalProcess module, the DICO and RealToInteger models translate the message that represents the entity into an Integer value, which is the reference to the memory where the entity is stored (i.e., a pointer). That value is assigned to the entityStart port, which should be connected to an input port of the model that represents the external process. Since the value assigned to the entityStart port is different for each entity, the external process should recognize the change in the value meaning that a new entity is ready to be processed.

The external process should also have an output port connected to the entityEnd port of the ExternalProcess module. After processing the entity, the external process should set the value of the entityEnd port of the ExternalProcess module with the reference (i.e., the pointer) previously received. In case of simultaneous processing of multiple entities, the external process should have a method to record the reference to the entities being processed and identify the beginning and end of the process for each entity individually.

The reference received through the entityEnd port is translated by the IntegerToReal and Quantizer models into a message that represents the processed entity, which continues through the flowchart diagram. An example of the use of this module is described in Chapter 12.

11.3.5 Decide

The Decide module represents a division in the flow of entities following certain conditions or probabilities. Its internal implementation is shown in Fig. 11.6. It is constructed using a Branch block and a BRule block, so only one condition can be checked in the Decide module. In order to allow multiple conditions, ARENALib includes the Rule module, which is equivalent to the BRule block. Additional Rule modules can be connected to the Out2 port of the Decide module.

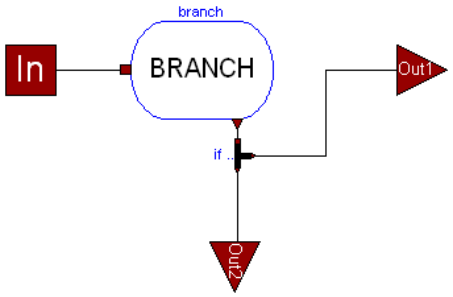


Figure 11.6: ARENALib Decide module.

11.3.6 Assign

The Assign module is equivalent to the ExternalAssign block of the SIMANLib library. It can be used to assign or modify the values of the user-defined attributes or global variables included in the model.

11.3.7 Record

The Record module represents a point in the flowchart diagram to record statistical time-dependent information. Its internal implementation is shown in Fig. 11.7. This module is composed of a Tally and a Counter blocks, which are conditionally declared depending on a parameter, named Type, of the module. If the Type of the module is 1, the Record behaves as a Counter block and so the Tally block is not declared. If the Type is 2, the Record behaves as a Tally block and the Counter is not declared. The module also includes the required Tally and Counter elements.

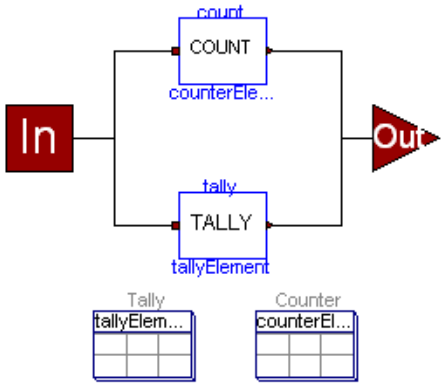


Figure 11.7: ARENALib Record module.

11.4 Data Modules

The additional information required by the flowchart modules is included in its internal implementation. However, models could also include other information that has to be described using data modules. Also, some of the included information can be shared between multiple processes and so, it has to be included separately.

The data modules included in ARENALib are (also see Fig. 11.1b):

- *Entity*, is equivalent to the EntityType element in SIMANLib. It describes the characteristics associated with a type of entities in the system.
- *Queue*, is equivalent to the Queue element in SIMANLib. The use of this data module is not required since it is already included in the Process and ExternalProcess modules.
- *Resource*, is equivalent to the Resource element in SIMANLib. Each Resource module describes a type of resource in the system, and has to be connected with the Process or ExternalProcess modules in the system. Several Process or ExternalProcess modules can be connected to the same Resource, if resource sharing is required.
- *Variable*, is equivalent to the Variable element in SIMANLib. It describes user-defined global variables in the system.
- *Attribute*, is equivalent to the Attribute element in SIMANLib. It describes user-defined attributes for the entities in the system.

The BasicProcess package also includes three functions, named eget(), vget() and aget(), that can be used to read the values of the variables of an Entity (eget), a Variable (vget) or an Attribute (aget). In this way, the values of the variables defined in Entities, Variables or Attributes can be used to configure the parameters of the flowchart modules (e.g., an attribute can be assigned with the time of creation for the entity, and its value used as a parameter to calculate the duration of a process or as a condition for a Decide module).

11.5 System Modeling Using ARENALib

The procedure to construct new process-oriented models using ARENALib is equivalent to the SIMANLib procedure. First, the Draft model has to be used to create the new "empty" model. Second, the flowchart diagram has to be described using flowchart modules. Third, the required data modules have to be included. Finally, the parameters of the modules have to be configured to represent the desired experiment.

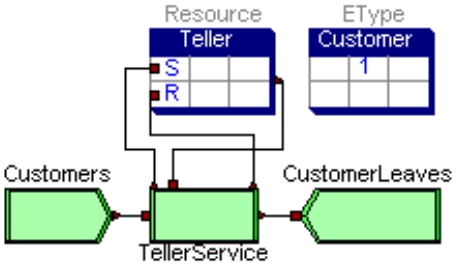


Figure 11.8: Bank teller system modeled using ARENALib.

Table 11.1: Bank teller system simulation results using SIMANLib, ARENALib, Arena and SIMAN.

Model	Avg. in Queue	Half-Width
SIMANLib	3.3073	-
ARENALib	3.1212	-
Arena	3.2089	0.22
SIMAN	3.2089	0.22

The model of the bank teller system, previously modeled using SIMANLib, constructed using ARENALib is shown in Fig. 11.8. A Create, Process and Dispose modules have been included to represent the flowchart diagram. A Resource module, to represent the teller, and an Entity module, to represent the customers, have also been used. Notice that since ARENALib modules provide more functionalities than SIMANLib blocks, less modules are required to represent the same system. The simulation results are equivalent to the ones obtained using SIMANLib, SIMAN and Arena (see Table 11.1). The results obtained using

SIMAN and Arena are equal because Arena uses the same random seed for both models.

11.6 Electronic Factory Model

The electronic assembly and test system described in Kelton et al. [2007] has been composed using ARENALib. The behavior of the system is as follows. Two types of electronic parts (A and B) are received, pre-processed and sealed. Each type has different pre-processing and sealing times. After that, the quality of the the sealed parts is inspected. Correct parts are shipped, and the rest need to be reworked. After the rework process, they are inspected again and classified into salvaged and scrapped.

The flowchart diagram of the developed model is shown in Fig. 11.9. Two Create modules have been used to represent the arrival of each type of part. After the arrival, two attributes are assigned to each entity: "sealer time" and "arrival time". The "sealer time" corresponds to the duration of the sealer process for that entity, and follows a triangular distribution for parts of type A and a weibull distribution for parts of type B. The "arrival time" is recorded to be used in the calculations of the statistics about the time spent in the system when the entity is disposed.

Each part arrives separately to the pre-processing operation, represented with two different Process modules, each one connected to its corresponding Resource data module. When the pre-processing is finished, the parts arrive to the sealing

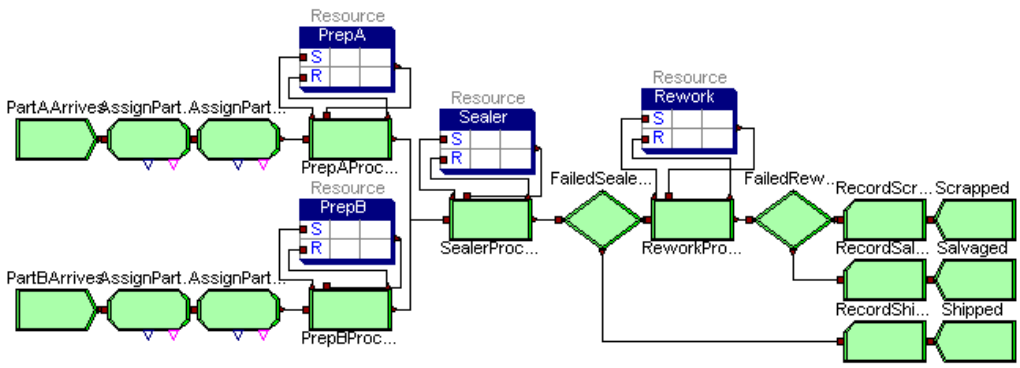


Figure 11.9: Electronic assembly system modeled using ARENALib.

process, which is again represented using a Process module connected to a Resource. The duration of this process is calculated using the value of the "sealer time" attribute of each entity.

The quality inspection after the sealing process is represented using a Decide block. In this block, the 91% of the entities are considered correct and are disposed in the Shipped module. The rest of the entities go to the rework process. After being reworked, a new quality inspection is applied to the entities. In this case, the 80% are considered salvaged and the rest is scrapped. Before disposing each entity, three Record modules calculate tally statistics about the time spent in the system for each class of processed part (shipped, salvaged and scrapped).

The system has been simulated during 50000 time units, in order to evaluate its steady-state behavior. Multiple statistical indicators are automatically calculated by ARENALib. Some of the calculated statistical indicators are shown in Table 11.2 and compared with the results obtained with Arena, including the half-width (H-W) interval. These results show that the bottleneck of the system is the rework process, due to its long duration and waiting times.

Table 11.2: Electronic factory simulation results, comparing ARENALib and Arena (in average values).

Indicator (avg.)	ARENALib	Arena	H-W
Time for Shipped	18.650	19.774	2.273
Time for Salvaged	89.348	81.522	8.715
Time for Scrapped	88.564	78.125	(Insuf)
Sealer.WaitTime	0.447	0.453	0.035
Sealer.ProcessTime	2.609	2.617	(Corr)
Sealer.Utilization	0.601	0.605	0.011
Sealer.NumberInQueue	0.103	0.105	0.007
Rework.WaitTime	40.272	32.974	8.020
Rework.ProcessTime	30.033	28.452	1.823
Rework.Utilization	0.622	0.583	0.043
Rework.NumberInQueue	0.834	0.675	0.180

11.7 Conclusions

The ARENALib library includes high-level functionalities for process-oriented modeling in Modelica. These functionalities replicate some of the functionalities found in the BasicProcess panel of the Arena simulation environment. The construction of models using ARENALib is also similar to Arena, using the drag and drop functionalities provided by Dymola.

ARENALib components have been described as P-DEVS coupled models, and constructed using the SIMANLib library. Thus, SIMANLib and ARENALib components can be hierarchically ordered and combined to construct models at different abstraction levels. The use of a mathematical formalism to describe models facilitates their comprehension, development and maintenance.

ARENALib has been used to construct process-oriented models of a bank teller and an electronic factory. The simulation of these models has been compared with equivalent models constructed using Arena. The results obtained, represented by statistical indicators automatically calculated during the simulation, are equivalent.

12

Hybrid Process-Oriented Modeling

12.1 Introduction

One of the strenghts of the SIMANLib and ARENALib libraries is the possibility to combine process-oriented models with the rest of the available Modelica libraries. This combination facilitates the description of large hybrid multi-domain systems.

This chapter includes a presentation of the functionalities of SIMANLib and ARENALib to interact with other models in Modelica. This presentation is performed by means of three case studies, because these functionalities have been already described in the previous chapters.

12.2 Orange Juice Canning Factory

This model represents an orange juice canning factory. Trucks carrying orange juice arrive to the factory with an exponential inter-arrival time. The juice is pumped into the factory tank, that feeds the canning system. Each truck has to wait for a position in the dock to unload the juice into the factory tank. The docking operation takes around 1 to 2 minutes (uniformly distributed). Once docked, the truck pumps the juice into the tank. If the tank gets full, the truck

must wait for free volume in the tank. When empty, the truck leaves the dock free for another waiting truck.

When the factory tank is not empty, the canning system produces pallets of canned juice. If the tank gets empty, the canning operation stops until new juice is available in the tank.

In this model, the flow of juice from the truck into the tank, and from the tank into the canner have been described as continuous-time model. The amount of juice in the truck is represented using a variable named *truckLevel*, and its variation is represented by the *truckRate* variable. The amount of juice in the tank is represented by the *tankLevel* variable, and its variation by the *tankRate* variable. The amount of juice in the canner is represented by the *cannerLevel* variable, and its variation by the *cannerRate* variable. The Modelica equations that describe the relations between these variables are shown in Listing 12.1.

```
truckRate = der(truckLevel);
tankRate = der(tankLevel);
cannerRate = der(cannerLevel);
```

Listing 12.1: Modelica equations for the orange juice factory.

The entities from the process-oriented part, that represent the trucks and the cans, interact with the equations that represent flows of juice in the system. The system also includes six Variable elements (see Fig. 12.1) that represent the previously mentioned variables. When an entity assigns a new value to a variable, using an ExternalAssign block, the value of the continuous-time variables is also modified. The assignments used to change the *truckRate* depending on the flow of entities crossing the ExternalAssign block are shown in Listing 12.2.

```
when change(truckR1.change) then
  truckRate := -200;
elsewhen change(truckR2.change) then
  truckRate := 0;
elsewhen change(truckR3.change) then
  truckRate := 0;
elsewhen change(truckR4.change) then
  truckRate := -200;
end when;
```

Listing 12.2: Detection of ARENALib external assignments using Modelica code.

When an entity crosses the truckR1 Assign module its "change" port switches its value. That event is detected by the when statement, setting the value of *truckRate* to -200. The same operation is performed with the rest of Assign modules. A similar structure of assignments is used for the other variables.

The flowchart diagram for this system, constructed using SIMANLib, is shown in Fig. 12.1. The figure also includes the required SIMANLib elements, associated with the blocks and to calculate statistics (DStat model named WaitingTrucks).

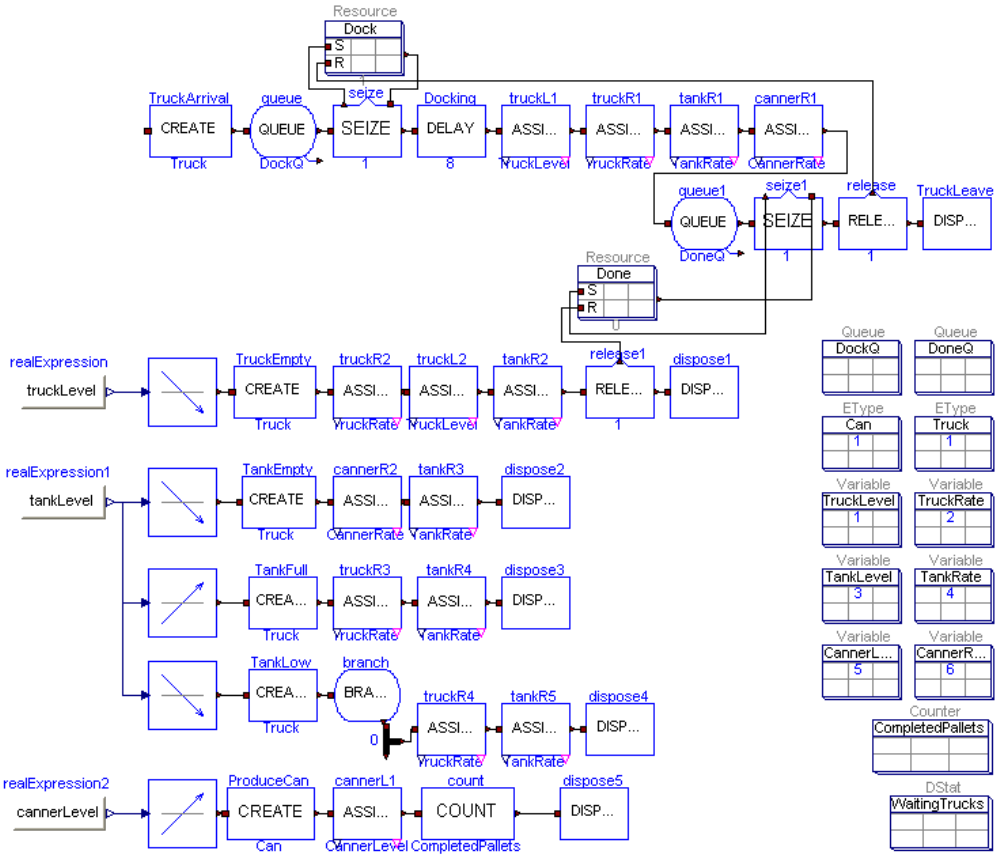


Figure 12.1: Orange juice canning factory modeled using SIMANLib.

The top of the diagram corresponds to the truck docking/undocking operation. When truck is created, seizes the dock, performs the docking operation and assigns new values for the mentioned Variables (the continuous-time variables are also assigned). The time spent unloading the truck is calculated by the continuous-time part, depending on the value of the truckRate variable.

The event of an empty truck is modeled using a CrossDOWN model, from the DEVSLib library, which generates a TruckEmtpy entity due to its connection to

the input port of a Create block. This entity modifies the values of the variables, using ExternalAssign blocks, and forces the truck being unloaded to finish the operation and release the dock (using the Done resource).

The management of the level of the tank (full, empty or available space) is performed similarly to the truck level. Two CrossDOWN and one CrossUP models are used to detect when the tank is empty, has available space to continue being filled or becomes full, respectively. These events produce other variable assignments that also affect the continuous-time variables. The level of the canner, and the production of pallets, is controlled in the same way.

The system has been simulated during 100 minutes. The evolution of the truck, tank and canner levels are shown in Fig. 12.2. Notice that the level of juice in the tank increases while the truck level is decreasing. Also, when the tank level is positive, the canner produces pallets, but when it reaches zero, the canner stops its operation.

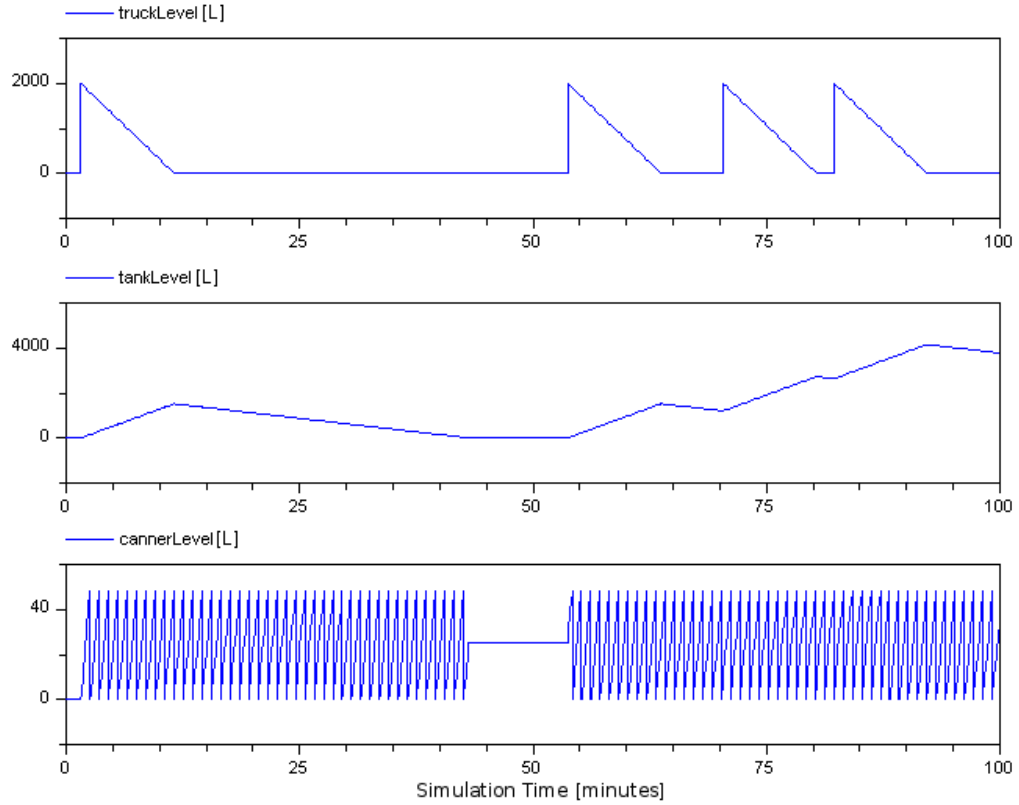


Figure 12.2: Simulation results of the orange juice canning factory.

12.3 Tank-level Control System

This hybrid model represents the level of a tank that is filled at a constant rate of $10l \cdot s^{-1}$, and when the volume reaches $100l$, the tank is emptied also at constant rate. Its behavior is very similar to the previous system (the orange juice factory), but in this case the system is modeled using ARENALib. Two approaches are studied.

In the first approach, the level of the tank has been modeled using a continuous-time variable. This variable is controlled by the entities in the discrete-event part of the model. Initially the flow rate to fill the tank is $10l \cdot s^{-1}$, so the tank starts getting filled. When an entity is created, it is delayed for $10s$, while waiting the tank level to reach $100l$. After that delay, the tank is full and the entity changes the flow rate to $-10l \cdot s^{-1}$. The tank starts emptying. After another $10s$, the tank is empty and the entity re-sets the flow rate to $10l \cdot s^{-1}$, to restart the process.

The flowchart diagram for this model is shown in Fig. 12.3. Notice the Break-Loop model included from the DEVSLib library. The Modelica code for the continuous-time part is shown in Listing 12.3.

```
algorithm
  when change(SetFlowRate.change) then
    tankFlowRate := SetFlowRate.y;
  elsewhen change(SetFlowRate2.change) then
    tankFlowRate := SetFlowRate2.y;
  end when;
equation
 tankFlowRate = der(tankLevel);
```

Listing 12.3: Detection of external assignments to the flow rate in the tank level system.

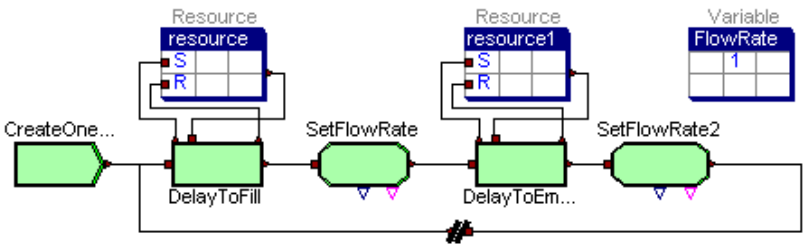


Figure 12.3: Tank-level control system modeled using ARENALib (first approach).

In the second approach, the level of the tank is checked using CrossUP and CrossDOWN models that detect when the tank gets full or empty. Depending on the event detected, the flow rate is changed to $-10l \cdot s^{-1}$ or $10l \cdot s^{-1}$, respectively.

The flowchart diagram for the second approach is shown in Fig. 12.4. In this case, the Process and Resource modules used to calculate the fill/empty time can be removed from the flowchart diagram. That time is calculated using the equation that relates the level of the tank with its first derivative (see Listing 12.3).

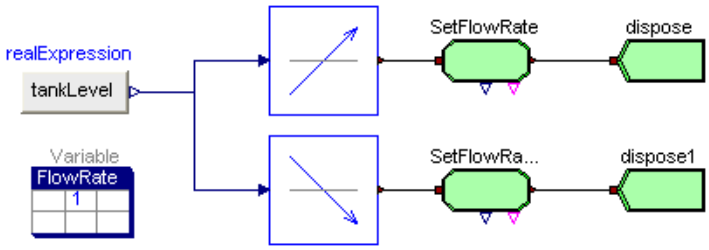


Figure 12.4: Tank-level control system modeled using ARENALib (second approach).

The simulation results for both systems are equivalent. The evolution of the tank level after 100 is shown in Fig. 12.5

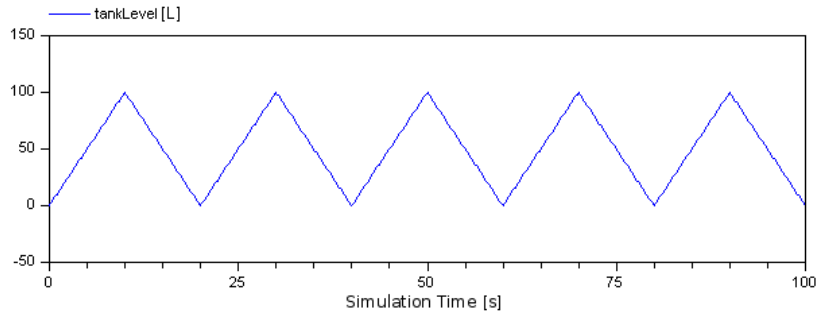


Figure 12.5: Evolution of the tank level for the Tank-level control system.

12.4 Soaking-Pit Furnace System

The soaking-pit furnace system described in Kelton et al. [2007] has been modeled using ARENALib. This system represents a furnace that has 9 slots for heating ingots. Ingots arrive to the system at an exponential arrival time, and are positioned in one of the available slots of the furnace. If there is no available

slot, the ingot must wait. When each ingot reaches its optimal temperature, it is removed from the slot and another ingot enters the furnace.

The behavior of the furnace has been described using a continuous-time model. The equations that describe the temperature of the furnace and the ingots are the following:

$$\dot{T} = 2 \cdot s(2600 - T) \tag{12.1}$$

$$\dot{t_i} = 0.15 \cdot s(T - t_i) \tag{12.2}$$

where T is the temperature in degrees of the furnace and t_i is the temperature of the i^{th} ingot in the furnace.

When a new cold ingot arrives to the furnace, its temperature decreases following Eq. (12.3),

$$T_{new} = \frac{T \cdot s(T - t_{new})}{ingots} \tag{12.3}$$

where T_{new} is the new temperature of the furnace, t_{new} is the temperature of the new ingot and *ingots* is the number of ingots in the furnace.

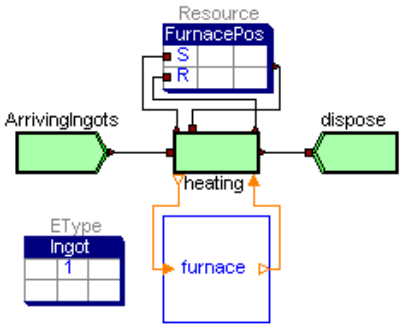


Figure 12.6: Soaking-pit furnace system modeled using ARENALib.

The flowchart diagram of the system is shown in Fig. 12.6. Ingots arrive at the Create module, seize an available slot, are heated and finally leave the system. The heating process is represented using an ExternalProcess module from ARENALib. The discrete-event module represents the operation for seizing a free slot in the furnace, and releasing it when finished. The furnace is modeled using a continuous-time model that includes the Eqs. (12.1), (12.2) and (12.3). A detail of the Modelica code of the furnace model is shown in Listing 12.4. The

reception of new ingots, the re-initialization of slot and furnace temperatures, and the management of the heating for each slot are shown.

```
algorithm
  when IN <> pre(IN) then // new ingot received
    while Posfree[i] > 0 loop // find free slot
      i := mod(i,numFurnacePos)+1;
    end while;
    Posfree[i] := 1; // assign free slot
    ingot[i] := IN;  // record value of input ingot
    newingot := newingot+1; // used to update furnace temp
  end when;
  for i in 1:numFurnacePos loop // check finished ingots
    when itemp[i] >= 2200 then
      Posfree[i] := 0;
      OUT := ingot[i];
    end when;
  end for;

equation
  ftrate = der(ftemp);
  ftrate = 2 * (2600 - ftemp); // furnace temperature
  for i in 1:numFurnacePos loop // slots temperatures
    itrate[i] = der(itemp[i]);
    itrate[i] = if Posfree[i] == 0 then 0 else 0.15 *(ftemp - itemp[i]);
  end for;
  when newingot <> pre(newingot) then // update furnace temperature
    reinit(itemp[i],u);
    reinit(ftemp,(ftemp-(ftemp - u)/sum(Posfree)));
  end when;
```

Listing 12.4: Detail of Modelica code of furnace model.

The soaking-pit furnace system with 9 slots has been simulated during 100 hours. The evolution of the temperatures in the furnace (above) and in each of the slots (below) are shown in Fig. 12.7.

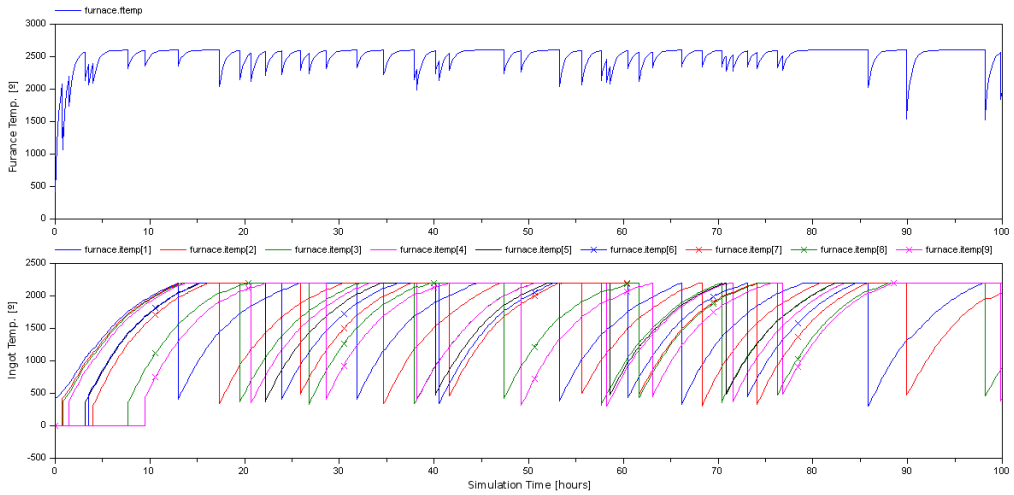


Figure 12.7: Evolution of temperatures in the soaking-pit furnace system.

12.5 Conclusions

SIMANLib and ARENALib include functionalities to construct hybrid process-oriented models in combination with other Modelica models. These functionalities include:

- The detection and communication of the changes performed to some of the variables described in the process-oriented model, using the ExternalAssign block in SIMANLib or the Assign module in ARENALib.
- The detection of event conditions using the functionalities included in DEVSLib, and the generation of entities based on these events using the input port of the Create block.
- The integration of external processes with process-oriented models using the ExternalProcess module included in ARENALib.

DEVSLib, SIMANLib and ARENALib components are compatible (i.e., use the same model communication mechanism), and the functionalities provided by the three libraries can be combined to enhance the description of models.

13 The RandomLib Library

13.1 Introduction

Process-oriented models are sometimes described stochastically [Law, 2007]. A key point in stochastic simulation is the random number generation. Good simulation results depend on a good source for random numbers, that allow to generate statistically good random variates [L'Ecuyer, 2001].

Previous works with Modelica use random number generators (RNGs) [Fritzson and Bunus, 2002; Mikler and Engelson, 2003; Aiordachioaie et al., 2006; Fabricius, 2003; Rubio et al., 2006; Lundvall and Fritzson, 2003]. Most of these authors use their own custom implementation of an RNG.

The RandomLib package is used in conjunction with DEVSLib, SIMANLib and ARENALib to model discrete-event and process-oriented stochastic models of logistic systems. RandomLib contains a Modelica implementation of the Combined Multiple Recursive Generator (CMRG) which is used in Arena [L'Ecuyer et al., 2002]. This helps to validate the models developed using SIMANLib and ARENALib, in comparison with equivalent models constructed using Arena.

RandomLib includes a package for uniform random number generation, called *CMRG*, and another for random variate generation, called *Variates*. It also includes several examples that help to understand its use and integration with other

libraries. These packages, their structure and functionalities are described in this chapter.

13.2 The CMRG package

The CMRG package contains an implementation of the Combined Multiple Recursive Generator described in [L'Ecuyer et al., 2002] and [L'Ecuyer, 1999]. Although available in C, provided by L'Ecuyer [2007], this generator has been implemented in Modelica in order to facilitate its use, comprehension and reutilization in other Modelica libraries. The C implementation has been used for validating the programmed Modelica generator.

The implemented RNG provides the possibility of creating multiple random streams, and sub-streams, that can be considered as independent RNGs [L'Ecuyer et al., 2002]. An brief introduction to this generator has been performed in

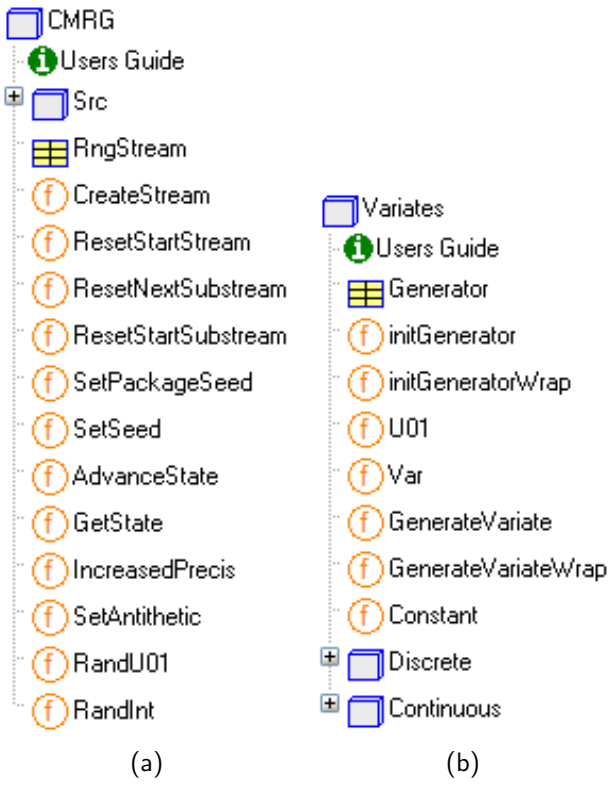


Figure 13.1: RandomLib structure: a) CMRG package; and b) Variates package.

Section 2.7.3, and a detailed description can be found in L'Ecuyer [2001]. The period of the generator is close to 2^{191}, and can be divided into disjoint streams of length 2^{127}. At the same time, each stream can also be divided into 2^{51} adjacent sub-streams, each of length 2^{76}. Each random variable can be assigned with a different stream, thus facilitating the execution of independent replications or the application of variance reduction techniques [Law and Kelton, 2000].

The architecture of the package is shown in Fig. 13.1a. The CMRG package is composed of: a) user's guide; b) developer package, named *Src*; and c) *RngStream* and a set of functions to manage it.

The main class in the CMRG package is the *RngStream* (see Fig. 13.1a). It represents a single stream of sequential random numbers. As mentioned above, an RngStream is divided into adjacent sub-streams. It is described using a Modelica record that contains the components described in Table 13.1 (which correspond to some of the generator states described in Section 2.7.3).

The CMRG package includes several functions to manage the RngStream. The *CreateStream* function is used to initialize a new RngStream, using the current seed. In RandomLib, the seed is automatically managed. It is stored is a text-file, named "CMRGSeed.txt". The CreateStream function reads the seed from that file to initialize the new RngStream. After initializing the stream, the

Table 13.1: Components of the RngStream record.

Cg	Indicates the current position in the stream. It is used to generate the next random number, and updated after the generation.
Bg	Indicates the start of the current stream. It can be used to reinit the state of the stream to its initial position, and repeat the sequence of random numbers.
Ig	Indicates the position of the next sub-stream in the current stream.
$Anti$	Boolean flag used to generate antithetic random numbers ($1 - u$, instead of u).
$IncPrec$	Also a boolean flag to generate random numbers in increased precision. Since Modelica only have one precision for Real numbers, this flag only makes the CMRG to generate alternate random numbers from the ones in the stream.

function updates the seed in the CMRGSeed.txt file. Thus, each call to the CreateStream function provides a new initial random stream. If the file is not found, the function creates it with the default seed ({12345,12345,12345,12345,12345,12345}), and initializes the RngStream with it. The user can specify a particular seed using the *SetPackageSeed* and *SetSeed* functions.

The *ResetStartStream*, *ResetNextSubstream* and *ResetStartSubstream* functions can be used to reset the state of the RngStream to the beginning of the current stream, the beginning of the next sub-stream or the beginning of the current sub-stream, respectively. The *AdvanceState* function can be used to arbitrarily change the state of the RngStream, independently from the stream and sub-stream division of the generator. *GetState* returns the current state of the RngStream. *SetAntithetic* and *IncreasedPrecis* change the value of the *Anti* and *IncPrec* flags.

Finally, *RandU01* and *RandInt* can be used to generate uniform random numbers (inside the (0,1) interval) and integer random numbers, respectively. A more detailed description of these functions and its parameters can be found in the documentation of the library.

The developer package contains the internal implementation of the CMRG. This package includes constants and functions to perform the required calculations to manage the state of the RngStreams, generate uniform random numbers and manage the seed. This implementation follows the structure of the C code provided by L'Ecuyer, and it has no utility for an standard user.

13.2.1 Uniform Random Number Generation

The steps required to generate uniform random numbers using RandomLib and the CMRG generator are:

1. Declare an RngStream that will be the source of random numbers.
2. Initialize the declared RngStream using the CreateStream function.
3. Generate random numbers using the declared RngStream and the RandU01 or RandInt functions.

The Examples package, included in RandomLib, contains several examples of random uniform and random variates generation in order to facilitate the use of the library. One of the included examples, named "CMRGSimple", is shown in Listing 13.1. In this example, a RngStream (g) is created and initialized using the CreateStream function. The *init* variable is used to execute the CreateStream function only once, because Dymola may execute the statements around the *initial*() condition several times during the initialization of the model. After that, when the simulation time reaches 1, the $RandU01(g)$ function is used to generate five different uniform random numbers that are stored in the indexed positions of the vector u. Notice that the RandU01 function returns two values: the uniform random number, and the updated state of the stream, that has to be stored in the RngStream g in order to generate future random numbers.

```
model CMRGSimple "generates 5 random numbers from an RngStream"
  CMRG.RngStream g "RngStream";
  Real u[5] "vector of random numbers";
  Boolean init( start = false);
algorithm
  // initialization of the RngStream
  when initial() and not init then
    g := CMRG.CreateStream();
    init := true;
  end when;
  when time <= 1 then
    // generation of uniform random numbers.
    for i in 1:5 loop
      (u[i],g) := CMRG.RandU01(g);
    end for;
  end when;
end CMRGSimple;

// Results:
// u = {0.988831,0.760616,0.855857,0.546418,0.868702}
```

Listing 13.1: Uniform random number generation using RandomLib.

Another example of uniform random numbers is shown in Listing 13.2. In this case, two different RngStreams are declared ($g1$ and $g2$). However, during the initialization both RngStreams are equaled and represent the same stream (both will provide the same random numbers). Six uniform random numbers are generated from $g1$ and stored in $u1$. On the other hand, only three uniform random numbers are generated from $g2$ (and stored in $u2$), which is then reset to the start of the sub-stream before generating another three uniform random

numbers (also stored in $u2$). Notice that $u2$ contains the same three uniform random numbers, due to the reset of the state of $g2$ to the start of the sub-stream. Also notice that $u1$ and $u2$ contain the same three first uniform random numbers, due to the equal initialization of both RngStreams.

```
model usingCMRG
  // declaration of the streams
  RandomLib.CMRG.RngStream g1,g2;
  Real u1[6],u2[6];
algorithm
  // stream initialization
  when initial() then
    g1:= RandomLib.CMRG.CreateStream();
    g2:= g1; // g1 and g2 represent the same stream
  end when;
  when time <= 0 then
    // generation of random numbers from g1
    for i in 1:6 loop
      (u1[i],g1):= RandomLib.CMRG.RandU01(g1);
    end for;
    // generation of random numbers from g2
    for i in 1:3 loop
      (u2[i],g2):= RandomLib.CMRG.RandU01(g2);
    end for;
    // reset g2 to the start of the sub-stream
    g2:= RandomLib.CMRG.ResetStartSubstream(g2);
    // Given that g2 was reset,
    // the same numbers are generated
    for i in 4:6 loop
      (u2[i],g2) := RandomLib.CMRG.RandU01(g2);
    end for;
  end when;
end usingCMRG;

// Results:
// u1 = {0.171491, 0.853403, 0.0237293, 0.501558, 0.179811, 0.252283}
// u2 = {0.171491, 0.853403, 0.0237293, 0.171491, 0.853403, 0.0237293}
```

Listing 13.2: Use of several RngStreams to generate uniform random numbers.

13.3 The Variates Package

The Variates package includes functionalities to generate random variates from continuous and discrete probability distributions. The structure of the package is shown in Fig. 13.1b. The package is composed of:

- The *Generator* record, that represents the source of uniform random numbers that will be used to generate variates. It is assigned by default to the RngStream record of the CMRG package.

– The *initGenerator* function is used to initialize the uniform random number generator (by default, assigned to the CreateStream function of the CMRG package). The *initGeneratorWrap* function performs the same action, returning the components of the RngStream (see Table 13.1) separately, instead of as a Modelica record.

– The *U01* function is used to obtain a uniform random number from a declared generator. It is assigned by default to the RandU01 function of the CMRG package.

– The *Var* function is the prototype (i.e., abstract function) for the variate generation functions, contained in the Continuous and Discrete packages.

– The *GenerateVariate* and *GenerateVariateWrap* functions help to generate random variates. They return a random variate from a probability distribution, selected using one of the parameters of the function.

– The *Constant* function simply generates a constant value.

– The *Continuous* and *Discrete* packages contain the functions to generate random variates. The included probability distribution functions are shown in Fig. 13.2.

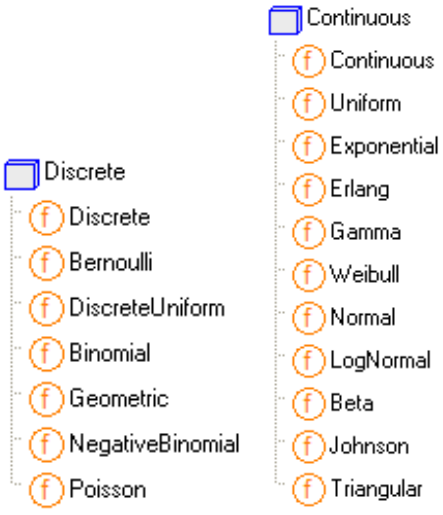


Figure 13.2: Discrete and continuous probability distribution functions included in RandomLib.

13.3.1 Random Variates Generation

Each random variate generation function extends the Var function. Each function receives as inputs a declared Generator (as source for uniform random numbers), and from one to four parameters depending on the probability distribution function (e.g., the exponential distribution function receives one parameter, and the triangular distribution function receives three). The algorithms used in the implementation of each function are described in the documentation included in the library. A method used to validate these functions has been to fit the generated variates with their distributions by using Arena Input Analyzer.

The generation of random variates using RandomLib is similar to the generation of uniform random numbers. The following steps have to be performed:

1. Declare a Generator, that will be used as source for uniform random numbers.
2. Initialize the declared Generator using the initGenerator function.
3. Generate random variates either using the GenerateVariate function, or directly calling the functions contained in the Continuous and Discrete packages.

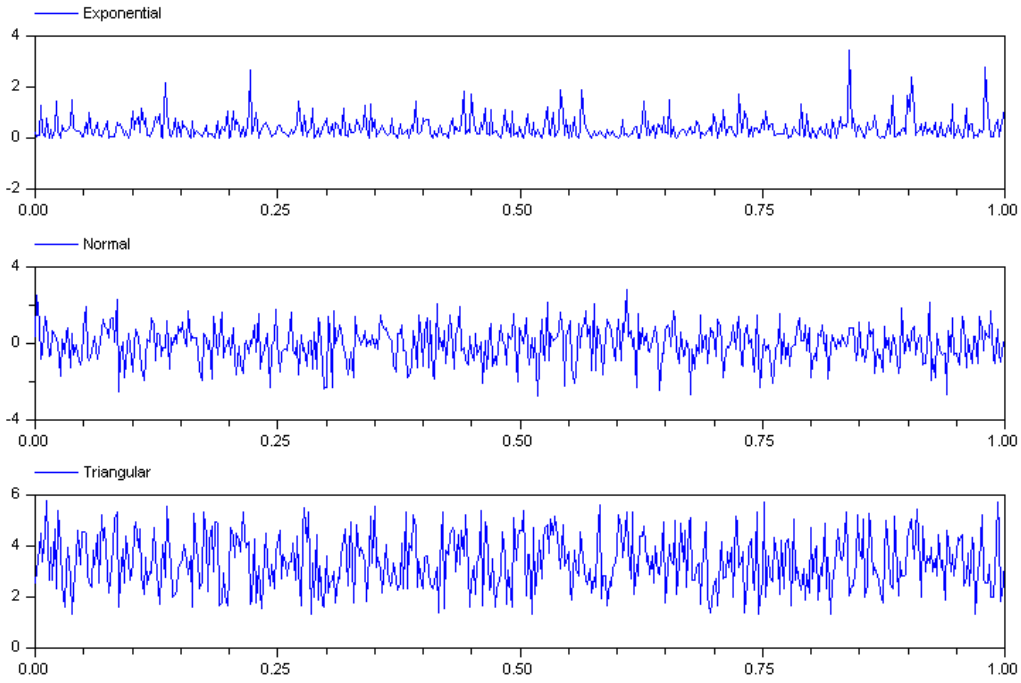
```
model VariatesSimple  "5 random variates with Expo(5) distrib."
  Variates.Generator g "RNG";
  Real u[5] "vector of random variates";
algorithm
  // initialization of the RngStream
  when initial() then
    g := Variates.initGenerator();
  end when;
  when time <= 0 then
    for i in 1:5 loop
      // generation of variates.
      (u[i],g) := Variates.Continuous.Exponential(g,5);
    end for;
  end when;
end VariatesSimple;

// Results:
// u =  {6.99771, 4.23703, 1.83142, 6.94764, 1.45234}
```

Listing 13.3: Random variates generation using RandomLib.

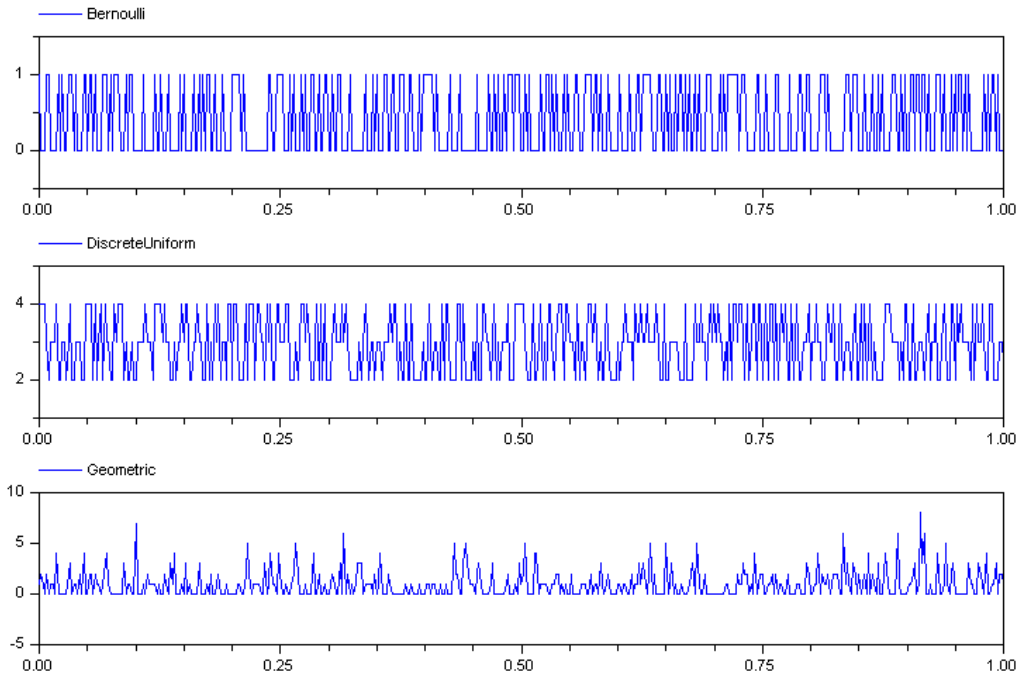
An example of random variates generation is shown in Listing 13.3. This model generates five random variates that follow the exponential probability distribution with mean 5. The declared generator is called g, and initialized using the initGenerator function within the *initial*() condition in Dymola. After that, five variates are generated using a for loop. Each iteration of the loop generates one variate by calling the $Exponential(g, 5)$ function (notice that the second parameter represents the mean), that also updates the state of the generator in order to generate future variates.

```
model VariatesSimple2 "A discrete variate of each distribution"
  Variates.Generator g "RNG";
  Real ConstantVariate;
  Real Bernoulli;
  Real DiscreteUniform;
  Real Binomial;
  Real Geometric;
  Real NegativeBinomial;
  Real Poisson;
  Real Uniform;
  Real Exponential;
  Real Erlang;
  Real Gamma;
  Real Weibull;
  Real Normal;
  Real LogNormal;
  Real Beta;
  Real Johnson;
  Real Triangular;
algorithm
  // initialization of the RngStream
  when initial() then
    g := Variates.initGenerator();
  end when;
  (ConstantVariate,g) := Variates.GenerateVariate(1,g,0.4);
  (Bernoulli,g) := Variates.GenerateVariate(2,g,0.4);
  (DiscreteUniform,g) := Variates.GenerateVariate(3,g,2,4);
  (Binomial,g) := Variates.GenerateVariate(4,g,4,0.3);
  (Geometric,g) := Variates.GenerateVariate(5,g,0.5);
  (NegativeBinomial,g) := Variates.GenerateVariate(6,g,6,0.2);
  (Poisson,g) := Variates.GenerateVariate(7,g,4.5);
  (Uniform,g) := Variates.GenerateVariate(8,g,2,4);
  (Exponential,g) := Variates.GenerateVariate(9,g,0.4);
  (Erlang,g) := Variates.GenerateVariate(10,g,0.5,4);
  (Gamma,g) := Variates.GenerateVariate(11,g,0.4,0.7);
  (Weibull,g) := Variates.GenerateVariate(12,g,0.4,0.8);
  (Normal,g) := Variates.GenerateVariate(13,g,0,1);
  (LogNormal,g) := Variates.GenerateVariate(14,g,0,1);
  (Beta,g) := Variates.GenerateVariate(15,g,0.6,0.4);
  (Johnson,g) := Variates.GenerateVariate(16,g,0.1,0.5,1,4);
  (Triangular,g) := Variates.GenerateVariate(17,g,1,3,6);
end VariatesSimple2;
```

Listing 13.4: Another example of random variates generation using RandomLib.



(a)



(b)

Figure 13.3: Some random variates generated by model VariatesSimple2, using RandomLib: a) continuous distributions and; b) discrete distributions.

Another example of random variate generation is shown in Listing 13.4. In this case, the GenerateVariate function is used to generate variates following multiple probability distributions. The first parameter of this function indicates, using an integer number, the probability distribution.

Notice that only one Generator (g) is used to generate all the variates. The evolution of the variates generated following the distributions Exponential(0.4), Normal(0,1) and Triangular(1,3,6) is shown in Fig. 13.3a. Also, the evolution of the variates generated following the distributions Bernoulli(0.4), DiscreteUniform(2,4) and Geometric(0.5) is shown in Fig. 13.3b.

13.3.2 Use of Another RNG

The included variate generation functions use by default the CMRG generator as source of uniform random numbers. However, any other Modelica library for uniform random number generation can be used.

The procedure to configure the Variates package in order to use another RNG is the following:

1. Redeclare the record that represents the generic generator (Generator). The record must include the data required to describe the state of the generator. For instance, a generator that reads random numbers from a text file could describe its state with the name of the file and the index of the next number to generate.

2. Redeclare the generator initialization function (initGenerator). The redeclared initGenerator function must return as output a Modelica record of the previously redeclared Generator class.

3. Redeclare the generic uniform random number generation function (U01). The redeclared U01 function must receive as input a Modelica record of the Generator class, and return as outputs a Real number (i.e., the generated uniform random number) and the updated state of the generator (i.e., a Modelica record of the Generator class).

13.4 Conclusions

The RandomLib library provides functionalities to generate random uniform numbers and random variates in Modelica. It can be used in combination with the DEVSLib, SIMANLib and ARENALib libraries to describe discrete-event stochastic models. It can also be used together with any other Modelica library.

The random number generator included in the library is the Combined Multiple Recursive Generator proposed by Pierre L'Ecuyer, and included in Arena as a source for uniform random numbers. RandomLib uses this RNG to generate random variates. Functions for discrete and continuous probability distributions are included. Any other RNG can also be adapted to be used with RandomLib.

14

Conclusions and Future Research

The conclusions of this dissertation, as well as some ideas for future research are presented in this chapter.

14.1 Conclusions

The conclusions reached in the development of this dissertation are the following:

- The combination of the P-DEVS formalism and the process-oriented modeling approach with the object-oriented modeling methodology facilitates the description of hybrid dynamic systems. The use of a mathematical formalism to describe the behavior of the discrete-event part of hybrid models facilitates their development, maintenance and reuse.

- The requirements needed to describe P-DEVS and process-oriented models using EOO languages, and particularly the Modelica language, have been analyzed and identified. The first requirement concerns the description of discrete-event model behavior. The second requirement concerns the description of model communications following a message passing mechanism. Finally, the third requirement concerns the description of model interfaces to combine discrete-event models with models from other Modelica libraries.

– The differences between the model communication mechanisms in P-DEVS and EOO languages have been analyzed. A message passing mechanism to be used in EOO languages has been proposed, specified and designed.

  The proposed message passing mechanism has been partially implemented using the current Modelica functionalities. Three alternatives to describe the communication mechanism have been evaluated and developed. A model of the semaphore synchronization method has been developed in Modelica, during the study of these alternatives. The approach selected for the final implementation uses dynamic memory to transmit messages between models. The developed message commmunication mechanism includes a default message type and the operations required to manage it.

  The developed mechanism has been used to describe the P-DEVS model communication in Modelica. It will facilitate the description of P-DEVS and process-oriented models.

– A new Modelica library, named DEVSLib, has been designed and developed to support the P-DEVS formalism. DEVSLib includes functionalities to describe atomic and coupled P-DEVS models. It also includes interface models to combine P-DEVS models with models developed using other Modelica libraries. DEVSLib models also support continuous-time inputs for the transition functions, in order to facilitate the combination with continuous-time models. Thus, DEVSLib can be used to describe discrete-event, continuous-time (using the QSS integration methods) and hybrid models.

  The construction of models using DEVSLib is close to their P-DEVS formal specification. It is performed by describing the elements of the tuple. The communication between DEVSLib models is performed using the implemented message passing mechanism.

– The application of the functionalities included in DEVSLib for describing hybrid control systems have been analyzed. DEVSLib can be used to describe event-based sensors and actuators. The library can be also used to

describe discrete-time and discrete-event controllers, represented as atomic or coupled P-DEVS models. These functionalities, when combined with the continuous-time modeling functionalities included in Modelica, facilitate the description of hybrid control systems.

– The requirements to describe process-oriented models in Modelica have been analyzed. Model communication between process-oriented models is similar to the communication between P-DEVS models, and so, the developed message passing mechanism can be used to facilitate their description. However, additional functionalities are required, like the description of the entities in the system and the management of variable-size data structures, used to store user-defined information and statistical indicators. These functionalities have been implemented in Modelica by means of two additional external libraries, named "entities.c" and "objects.c". The former has been developed to manage the information required to describe the entities and their flow through the system. The latter has been developed to manage dynamic objects, that describe variable-size matrices and lists of matrices stored in dynamic memory.

– Two new Modelica libraries, named SIMANLib and ARENALib, have been designed and developed to support the description of models following the process-oriented approach. These new libraries reproduce some of the functionalities provided by the SIMAN language and the Arena simulation environment.

  The description of the components of both libraries has been performed using the P-DEVS formalism. The use of a mathematical formalism to describe models facilitate their comprehension, development and maintenance.

  The development of SIMANLib has been performed using the DEVSLib library, since SIMANLib components are defined as atomic P-DEVS models. The development of ARENALib has been performed using the SIMANLib library, similarly to how Arena components are described by means

of SIMAN constructs. Thus, SIMANLib and ARENALib components can be hierarchically ordered and combined to construct models at different abstraction levels.

– The functionalities included in the SIMANLib and ARENALib libraries have been extended to facilitate the description of hybrid process-oriented models. The Create and ExternalAssign blocks from SIMANLib, and the Assign and ExternalProcess modules from ARENALib can be used to combine process-oriented models with other Modelica models.

– Another new Modelica library, named RandomLib, has been developed to provide uniform random numbers and random variates generation functionalities. RandomLib, in combination with DEVSLib, SIMANLib and ARENALib, can be used to describe stochastic models of logistic systems. RandomLib includes an implementation of the CMRG random number generator, also used in Arena, in order to facilitate the validation of the developed models by comparison with equivalent models constructed using Arena or SIMAN. RandomLib also includes random variates generation functions, from multiple discrete and continuous probability distributions.

– Finally, several case studies have been developed to present the functionalities and use of the implemented libraries. Models of an automatic teller machine, a predator-prey system, an opto-electical communication system, a supermarket refrigeration system, a crane and embedded controller system, a restaurant, an electronic assembly factory, an orange juice canning factory, a tank level controller system and a soaking-pit furnace system have been described.

14.2 Future Research

Some ideas for future research work could be:

– The improvement of the simulation performance of discrete-event and hybrid models in Modelica, described using the P-DEVS formalism.

- The analysis and definition of an standard language to describe P-DEVS models. Such a language should be independent from any programming language or simulation tool, and could be used to share and reuse models in different environments. The combination and integration of models described using this new language with EOO languages has also to be analyzed and defined. A first approach could be the automatic translation of its models into DEVSLib models, that could be simulated using Dymola. Another approach could be the description of this new language as a subset of Modelica.

- The extension of the functionalities included in SIMANLib and ARENALib, by developing additional components.

- The graphical representation of the execution of process-oriented models in Modelica, to allow a better understanding of the behavior of the model during the simulation.

Bibliography

Agrawal, G. P. [1997], *Fiber-Optic Communication Systems*, 2^{nd} edn, Wiley-Interscience, New York, NY, USA.

Aiordachioaie, D., Nicolau, V., Munteanu, M. and Sirbu, G. [2006], On the noise modelling and simulation, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 369–375.

Andersson, M. [1989], Omola - an object-oriented language for model representation, Technical report, TFRT 7417, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Andersson, M. [1994], Object-Oriented Modeling and Simulation of Hybrid Systems, PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Andreasson, J. [2003], VehicleDynamics library, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 11–18.

ARGESIM [2009], 'ARGE simulation news group website'.
URL: *http://www.argesim.org*

ARS Lab [2010], 'ARS Lab - advanced real-time simulation laboratory', Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada.

Åström, K. J., Elmqvist, H. and Mattsson, S. E. [1998], Evolution of continuous-time modeling and simulation, *in* 'Proceedings of the 12^{th} European Simulation Multiconference (ESM'98)', Manchester, UK, pp. 9–18.

Atkinson, K. E. [1989], *An Introduction to Numerical Analysis*, 2^{nd} edn, John-Wiley & Sons, New York, NY, USA.

Augustin, D. C., Fineberg, M. S., Johnson, B. B., Linebarger, R. N., Sansom, F. J. and Strauss, J. C. [1967], 'The SCi continuous system simulation language (CSSL)', *Simulation* **9**, 281–303.

Banks, J., Carson, J. S. and Nelson, B. L. [1996], *Discrete-Event System Simulation*, 2^{nd} edn, Prentice Hall.

Barros, F. J. [1995], Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation, *in* 'Proceedings of the 27^{th} Conference on Winter simulation', Arlington, VA, USA, pp. 781–785.

Barros, F. J. [2002*a*], 'Modeling and simulation of dynamic structure heterogeneous flow systems', *SIMULATION: Transactions of The Society for Modeling and Simulation International* **78**(1), 18–27.

Barros, F. J. [2002*b*], 'Towards a theory of continuous flow models', *International Journal of General Systems* **31**(1), 29–39.

Barros, F. J. [2003], 'Dynamic structure multi-paradigm modeling and simulation', *ACM Transactions on Modeling and Computer Simulation* **13**(3), 259–275.

Barton, P. L. and Pantelides, C. C. [1994], 'Modeling of combined discrete/continuous processes', *AIChE Journal* **40**, 966–979.

Beltrame, T. [2006], Design and development of a Dymola/Modelica library for discrete event-oriented systems using DEVS methodology, Master's thesis, ETH Zürich.

Beltrame, T. and Cellier, F. E. [2006], Quantised state system simulation in Dymola/Modelica using the DEVS formalism, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 73–82.

Biere, M., Gheorghe, L., Nicolescu, G., O'Connor, I. and Wainer, G. [2007], Towards the high-level design of optical networks-on-chip. formalization of opto-electrical interfaces, *in* 'Proceedings of the 14^{th} IEEE International Conference on Electronics, Circuits and Systems (ICECS 2007)', Marrakesh, Morocco, pp. 427–430.

Brenan, K. E., Campbell, S. L. and Petzold, L. R. [1989], *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, North-Holland, New York, NY, USA.

Breuneuse, A. P. J. and Broenink, J. F. [1997], 'Modeling mechatronic systems using the SIDOPS+ language', *Simulation Series* **29**(1), 301–306.

Brière, M., Carrel, L., Michalke, T., Mieyeville, F., O'Connor, I. and Gaffiot, F. [2004], Design and behavioral modeling tools for optical network-on-chip, *in* 'Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)', IEEE Computer Society, Washington, DC, USA, p. 10738.

Brière, M., Drouard, E., Mieyeville, F., Navarro, D., O'Connor, I. and Gaffiot, F. [2005], Heterogeneous modelling of an optical network-on-chip with SystemC, *in* 'Proceedings of the 16^{th} IEEE International Workshop on Rapid System Prototyping (RSP'05)', IEEE Computer Society, Washington, DC, USA, pp. 10–16.

Brière, M., Girodias, B., Bouchebaba, Y., Nicolescu, G., Mieyeville, F., Gaffiot, F. and O'Connor, I. [2007], System level assessment of an optical NoC in an MPSoC platform, *in* 'Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)', Nice, France, pp. 1084–1089.

Bush, V. [1931], 'The differential analyzer: A new machine for solving differential equations', *Journal of the Franklin Institute* **212**, 447–488.

Butcher, J. C. [2003], *Numerical Methods for Ordinary Differential Equations*, 2^{nd} edn, Wiley, Chichester, UK.

Campbell, S. L., Chancelier, J.-P. and Nikoukhah, R. [2006], *Modeling and simulation in Scilab\Scicos*, Springer, New York, NY, USA.

Casella, F. and Leva, A. [2003], Modelica open library for power plant simulation: Design and experimental validation, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 41–50.

Casella, F. and Richter, C. [2008], Externalmedia: A library for easy re-use of external fluid property code in Modelica, *in* 'Proceedings of the 6^{th} International Modelica Conference', Bielefeld, Germany, pp. 157–161.

Cassandras, C. G. and Lafortune, S. [1999], *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, Norwell, MA, USA.

Cellier, F. E. [1979], Combined Continuous/Discrete System simulation by Use of Digital Computers: Techniques and Tools, PhD thesis, ETH Zurich, Switzerland.

Cellier, F. E. [1996], Object-oriented modeling: Means for dealing with system complexity, *in* 'Proceedings of the 15^{th} Benelux Meeting on Systems and Control', Mierly, The Netherlands, pp. 53–64.

Cellier, F. E. [2008], World3 in Modelica: Creating System Dynamics models in the Modelica framework, *in* 'Proceedings of the 6^{th} International Modelica Conference', Bielefeld, Germany, pp. 393–400.

Cellier, F. E., Clauss, C. and Urquia, A. [2007], Electronic circuit modeling and simulation in Modelica, *in* 'Proceedings of the 6^{th} Eurosim Congress on Modelling and Simulation', Ljubljana, Slovenia, pp. 1–10.

Cellier, F. E., Elmqvist, H. and Otter, M. [1996], Modeling from physical principles, *in* W. Levine, ed., 'The Control Handbook', CRC Press, Boca Raton, FL, USA.

Cellier, F. E., Elmqvist, H., Otter, M. and Taylor, J. H. [1993], Guidelines for modeling and simulation of hybrid systems, *in* 'Proceedings of the IFAC World Congress', Sydney, Australia.

Cellier, F. E. and Greifeneder, J. [2008], ThermoBondLib - a new Modelica library for modeling convective flows, *in* 'Proceedings of the 6^{th} International Modelica Conference', Vol. 1, Bielefeld, Germany, pp. 163–172.

Cellier, F. E. and Kofman, E. [2006], *Continuous System Simulation*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Cellier, F. E. and Nebot, A. [2005], The Modelica bond graph library, *in* 'Proceedings of the 4^{th} International Modelica Conference', Vol. 1, Hamburg, Germany, pp. 57–65.

Chow, A. C. H. [1996], 'Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator', *Transactions of the Society for Computer Simulation International* **13**(2), 55–67.

Codecà, F. and Casella, F. [2006], Neural network library in Modelica, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 549–557.

D'Abreu, M. C. and Wainer, G. A. [2005], M/CD++: Modeling continuous systems using Modelica and DEVS, *in* 'Proceedings of the 13^{th} IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems', pp. 229–236.

Darnell, P. A. and Kolk, R. A. [1990], An interactive simulation and control design environment, *in* 'Proceedings of the 1990 European Simulation Symposium', Ghent, Belgium, pp. 56–60.

Dassault Systemes [2009], 'Computer aided three dimensional interactive application'.
URL: *http://www.catia.com*

David, R. and Alla, H. [2001], 'On hybrid Petri Nets', *Discrete Event Dynamic Systems* **11**(1-2), 9–40.

Dijkstra, E. W. [1965], Over seinpalen (ewd74). circulated privately.
URL: *http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF*

Donida, F., Ferretti, G., Savaresi, S. M., Schiavo, F. and Tanelli, M. [2006], Motorcycle dynamics library in Modelica, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 157–166.

Dynasim AB [2006], 'Dymola, dynamic modeling laboratory. user's manual'.
URL: *http://www.dymola.com/*

EA International [2010], 'EcosimPro - EL modeling language'.
URL: *http://www.ecosimpro.com*

Elmqvist, H. [1978], A Structured Model Language for Large Continuous Systems, PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H., Cellier, F. E. and Otter, M. [1993], Object-oriented modeling of hybrid systems, *in* 'Proceedings of the European Simulation Symposium', Delft, The Netherlands.

Elmqvist, H., Cellier, F. E. and Otter, M. [1994], Object-oriented modeling of power-electronic circuits using Dymola, *in* 'Proceedings of the CISS - First Joint Conference of International Simulation Societies', Zurich, Switzerland.

Elmqvist, H., Mattsson, S. E. and Otter, M. [1998], Modelica – the new object-oriented modeling language, *in* 'Proceedings of the 12^{th} European Simulation Multiconference', Manchester, UK, pp. 127–131.

Elmqvist, H. and Otter, M. [1994], Methods for tearing systems of equations in object-oriented modeling, *in* 'Proceedings of the European Simulation Multiconference (ESM'94)', Barcelona, Spain, pp. 326–332.

Euc [2009], 'Euclides web-site'.
URL: *http://www.euclides.dia.uned.es/*

Fabricius, S. [2003], Modeling and Simulation for Plant Performability Assessment with Application to Maintenance in the Process Industry, PhD thesis, ETH Zurich, Switzerland.

Fabricius, S. M. and Badreddin, E. [2002*a*], Hybrid dynamic plant performance analysis supported by extensions to the Petri Net library in Modelica, *in* 'Proceedings of the 4^{th} Asian control Conference (ASCC)', Singapore, pp. 41–50.

Fabricius, S. M. and Badreddin, E. [2002*b*], Modelica library for hybrid simulation of mass flow in process plantes, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 225–234.

Felgner, F., Agustina, S., Bohigas, R. C., Merz, R. and Litz, L. [2002], Simulation of thermal building behavior in Modelica, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 147–154.

Ferreira, J. and de Oliveira, J. E. [1999], Modelling hybrid systems using StateCharts and Modelica, *in* 'Proceedings of the 7^{th} IEEE International Conference on Emerging Technologies and Factory Automation', pp. 1063–1069.

Fishman, G. S. [2001], *Discrete-Event Simulation: Modeling, Programming, and Analysis*, Springer, New York, NY, USA.

Föllinger, O. [1985], *Regelungstechnik (5. Auflage)*, Hüthig, Heidelberg.

Frey, P. and O'Riordan, D. [2000], Verilog-AMS: Mixed-signal simulation and cross domain connect modules, *in* 'Proceedings of the 2000 IEEE/ACM International Workshop on Behavioral Modeling and Simulation', Washington, DC, USA, pp. 103–108.

Fritzson, P. [2003], *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Computer Society Pr.

Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Saldamli, L., Johansson, H. and Karström, A. [2002], The open source Modelica project, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 297–306.

Fritzson, P. and Bunus, P. [2002], Modelica – a general object-oriented language for continuous and discrete-event system modeling, *in* 'Proceedings of the 35^{th} Annual Simulation Symposium', pp. 14–18.

Fritzson, P., Viklund, L., Fritzson, D. and Herber, J. [1995], 'High-level mathematical modelling and programming', *IEEE Software* **12**(4), 77–87.

Garifullin, M. [2003], 'An object-oriented hybrid approach to ARGESIM comparison 'C13 Crane and Embedded Control' with AnyLogic', *Simulation News Europe* **37**, 29.

Giambiasi, N. and Carmona, J. C. [2006], 'Generalized discrete event abstraction of continuous systems: GDEVS formalism', *Simulation Modelling Practice and Theory* **14**(1), 47 – 70.

Grace, A. C. W. [1991], SIMULAB, an integrated environment for simulation and control, *in* 'Proceedings of the 1991 American Control Conference', Boston, MA, USA, pp. 1015–1020.

Himmelspach, J. and Uhrmacher, A. M. [2009], The JAMES II framework for modeling and simulation, *in* 'Proceedings of the 2009 International Workshop on High Performance Computational Systems Biology', Trento, Italy, pp. 101–102.

Hong, J. S., Song, H.-S., Kim, T. G. and Park, K. H. [1997], 'A real-time discrete event system specification formalism for seamless real-time software development', *Discrete Event Dynamic Systems* **7**, 355–375.

Hrúz, B. and Zhou, M. [2007], *Modeling and Control of Discrete-Event Dynamic Systems*, Springer, London, UK.

IEEE [1997], Standard VHDL analog and mixed-signal extensions, Technical Report 1076.1, IEEE.

Ioannou, P. G. and Martinez, J. C. [1999], Who serves whom? dynamic resource matching in an activity-scanning simulation system, *in* 'Proceedings of the 1999 Winter Simulation Conference', Phoenix, AZ, USA, pp. 963–970.

ITI GmbH [2009], 'SimulationX'.
URL: *http://www.simulationx.com/*

Jackson, A. S. [1960], *Analog Computation*, McGraw-Hill, New York, NY, USA.

Jacobs, P. H., Lang, N. A. and Verbraeck, A. [2002], D-SOL; a distributed Java based discrete event simulation architecture, *in* 'Proceedings of the 2002 Winter Simulation Conference', San Diego, CA, USA, pp. 793–800.

Jeandel, A., Boudaud., F. and Larivière, E. [1997], *ALLAN Simulation release 3.1 description*, M.DéGIMA.GSA1887. GAZ DE FRANCE, DR, Saint Denis La plaine, France.

Kelton, W. D., Sadowski, R. P. and Sturrock, D. T. [2007], *Simulation with Arena*, 4^{th} edn, McGraw-Hill, Inc., New York, NY, USA.

Kiviat, P. J. [1969], Digital computer simulation: Computer programming languages, Technical report, RAND Memo RM-5883-PR. RAND Corporation. Santa Monica, California.

Kloas, M., Friesen, V. and Simons, M. [1995], 'Smile - a simulation environment for energy sytems', *System Analysis Modelling Simulation* **18–19**, 503–506.

Kofman, E. [2004], 'Discrete event simulation of hybrid systems', *SIAM Journal on Scientific Computing* **25**(5), 1771–1797.

Kofman, E. and Junco, S. [2001], 'Quantized-state systems: a DEVS approach for continuous system simulation', *Transactions of the Society for Computer Simulation International* **18**(3), 123–132.

Kofman, E., Lee, J. and Zeigler, B. P. [2001], DEVS representation of differential equation systems: Review of recent advances, *in* 'Proceedings of 2001 European Simulation Symposium', Marseille, France.

Kriger, J. [2002], 'A very simple alarm clock with a pendulum in CD++'.
URL: *http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm*

Kwon, Y. W., Park, H. C., Jung, S. H. and Kim, T. G. [1996], Fuzzy-DEVS formalism: concepts, realization and applications, *in* 'Proceedings of the AIS'96', pp. 227–234.

Lackner, M. R. [1962], Toward a general simulation capability, *in* 'Proceedings of the Spring Joint Computer Conference', San Francisco, CA, USA, pp. 1–14.

Larsen, L. F. S., Izadi-Zamanabadi, R. and Wisniewski, R. [2007], Supermarket refrigeration system - benchmark for hybrid system control, *in* 'Proceedings of the European Control Conference', Kos, Greece, pp. 113–120.

Larsson, M. [2000], ObjectStab - a Modelica library for power system stability studies, *in* 'Proceedings of the Modelica Workshop 2000', Lund, Sweden, pp. 13–22.

Law, A. M. [2007], *Simulation Modelling and Analysis*, 4^{th} edn, McGraw-Hill, New York, NY, USA.

Law, A. M. and Kelton, W. D. [2000], *Simulation Modelling and Analysis*, 3^{rd} edn, McGraw-Hill Education - Europe.

L'Ecuyer, P. [1999], 'Good parameters and implementations for combined multiple recursive random number generators', *Oper. Res.* **47**(1), 159–164.

L'Ecuyer, P. [2001], Software for uniform random number generation: distinguishing the good and the bad, *in* 'Proceedings of the 33^{rd} Conference on Winter Simulation (WSC'01)', pp. 95–105.

L'Ecuyer, P. [2007], 'CMRG source code web page'.
URL: *http://www.iro.umontreal.ca/ lecuyer/myftp/streams00/*

L'Ecuyer, P., Simard, R., Chen, E. J. and Kelton, W. D. [2002], 'An object-oriented random-number package with many long streams and substreams', *Oper. Res.* **50**(6), 1073–1075.

Liu, Q. and Wainer, G. [2007], 'Parallel environment for DEVS and Cell-DEVS models', *SIMULATION* **86**(6), 449–471.

LMS International [2009], 'Imagine.Lab AMESim'.
URL: *http://www.lmsintl.com/imagine-amesim-intro*

Lotka, A. J. [1925], *Elements of Physical Biology*, Williams and Wilkins, Baltimore.

Lundvall, H. and Fritzson, P. [2003], Modelling concurrent activities and resource sharing in Modelica, *in* 'Proceedings of the 44^{th} Conference on Simulation and Modeling (SIMS 2003)'.

Lynch, N., Segala, R. and Vaandrager, F. [2003], 'Hybrid I/O autamata', *Information and Computation* **180**(1), 103–157.

Maplesoft [2009], 'MapleSim'.
URL: *http://www.maplesoft.com/products/maplesim/*

Martin-Villalba, C., Urquia, A. and Dormido, S. [2008], 'An approach to virtual-lab implementation using Modelica', *Mathematical and Computer Modelling of Dynamical Systems* **14**(4), 341–360.

Martinez, J. C. and Ioannou, P. G. [1995], Advantages of the activity scanning approach in the modeling of complex construction processes, *in* 'Proceedings of the 1995 Winter Simulation Conference', Arlington, VA, USA, pp. 1024–1031.

MathCore Engineering AB [2009], 'MathModelica System Designer'.
URL: *http://www.mathcore.com/products/mathmodelica/*

Mattsson, S. E., Elmqvist, H. and Broenink, J. [1998], 'Modelica – an international effort to design the next generation modeling language', *Journal A, Benelux Quarterly Journal on Automatic Control* **38**(3), 16–19.

Mattsson, S. E., Otter, M. and Elmqvist, H. [1999], Modelica hybrid modeling and efficient simulation, *in* 'Proceedings of the 38^{th} IEEE Conference on Decision and Control', pp. 3502–3507.

Mieyeville, F., Brière, M., O'Connor, I., Gaffiot, F. and Jacquemod, G. [2004], 'A VHDL-AMS library of hierarchical optoelectronic device models', *Languages*

for system specification: Selected contributions on UML, SystemC, system Verilog, mixed-signal systems, and property specification from FDL'03 pp. 183–199.

Mikler, J. and Engelson, V. [2003], Simulation for operation management: Object oriented approach using Modelica, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 207–214.

Mitchell, E. E. L. and Gauthier, J. S. [1976], 'Advanced continuous simulation language (ACSL)', *Simulation* **26**(3), 72–78.

Modelica [2010], 'Modelica web site'.
URL: *http://www.modelica.org*

Modelica Association [2009], 'Modelica - a unified object-oriented language for physical systems modeling. language specification (v. 3.1)'.
URL: *http://www.modelica.org/documents*

Modelica Libraries [2010], 'Modelica free and comercial libraries'.
URL: *http://www.modelica.org/libraries*

Mosterman, P. J., Otter, M. and Elmqvist, H. [1998], Modelling Petri Nets as local constraint equations for hybrid systems using Modelica, *in* 'Proceedings of the Summer Computer Simulation Conference', pp. 314–319.

MSL [2010], 'Modelica standard library'.
URL: *http://www.modelica.org/libraries/Modelica*

Nance, R. E. [1993], 'A history of discrete event simulation programming languages', *ACM SIGPLAN Notices* **28**(3), 149–175.

Nance, R. E. and Sargent, R. E. [2002], 'Perspectives on the evolution of simulation', *Operations Research* **50**(1), 161–172.

Nikoukhah, R. [2007], Hybrid dynamics in Modelica: Should all events be considered synchronous, *in* 'Proceedings of the 1^{st} International Workshop on Equation-Based Object-Oriented Languages and Tools', Berlin, Germany, pp. 37–48.

Nikoukhah, R. and Furic, S. [2008], Synchronous and asynchronous events in Modelica: proposal for an improved hybrid model, *in* 'Proceedings of the 6^{th} International Modelica Conference', Bielefeld, Germany, pp. 677–678.

Nutaro, J. [1999], 'ADEVS - a discrete event system simulator', Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson.

URL: *http://www.ece.arizona.edu/ nutaro/index.php.*

O'Connor, I. [2004], Optical solutions for system-level interconnect, *in* 'Proceedings of the 2004 International Workshop on System Level Interconnect Prediction (SLIP'04)', Paris, France, pp. 79–88.

O'Connor, I., Tissafi-Drissi, F., Navarro, D., Mieyeville, F., Gaffiot, F., Dambre, J., de Wilde, M., Stroobandt, D. and Briere, M. [2006], 'Integrated optical interconnect for on-chip data transport', *Circuits and Systems, 2006 IEEE North-East Workshop on* pp. 209–209.

Olsson, H. [2005], External interface to Modelica in Dymola, *in* 'Proceedings of the 4^{th} International Modelica Conference', Hamburg, Germany, pp. 603–611.

Otter, M., Årzén, K.-E. and Dressler, I. [2005], StateGraph - a Modelica library for hierarchical state machines, *in* 'Proceedings of the 4^{th} International Modelica Conference', Hamburg, Germany, pp. 569–578.

Otter, M. and Elmqvist, H. [1995], The DSblock model interface for exchanging model components, *in* 'Proceedings of the 1995 EUROSIM Conference', Vienna, Austria, pp. 505–510.

Otter, M., Elmqvist, H. and Mattsson, S. E. [1999], Hybrid modeling in Modelica based on the synchronous data flow principle, *in* 'Proceedings of the 10^{th} IEEE International Symposium on Computer Aided Control System Design', pp. 151–157.

Otter, M., Elmqvist, H. and Mattsson, S. E. [2003], The new Modelica Multi-Body library, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 311–330.

Page, E. H. [1994], Simulation Modeling Methodology: Principles and Etiology of Decision Support, PhD thesis, Virginia Polytechnic Institute and State University, Virginia, USA.

Pantelides, C. C. [1988], 'The consistent initialization of differential-algebraic systems', *SIAM J. SCI. STAT. COMPUT.* **9**(2), 213–231.

Pegden, C. D., Sadowski, R. P. and Shannon, R. E. [1995], *Introduction to Simulation Using SIMAN*, McGraw-Hill, Inc., New York, NY, USA.

Petzold, L. R. [1983], A description of DASSL: A differential-algebraic system solver, *in* R. S. Stepleman, ed., 'Scientific Computing', North-Holland, Amsterdam, pp. 65–68.

Pidd, M. [2004], *Computer Simulation in Management Science*, 5^{th} edn, John Wiley & Sons, Chichester, West Sussex, UK.

Prähofer, H. [1991], 'System theoretic formalisms for combined discrete-continuous system simulation', *International Journal of General Systems* **19**(3), 219–240.

Quesnel, G., Duboz, R. and Ramat, E. [2008], 'The Virtual Laboratory Environment - an operational framework for multi-modelling, simulation and analysis of complex dynamical systems', *Simulation Modelling Practice and Theory* **17**(4), 641–653.

Ragazzini, J. R., Randall, R. H. and Russell, F. A. [1964], 'Analysis of problems in dynamics by electronic circuits', *Simulation* **3**(3), 54–65.

Razavi, B. [2002], *Design of Integrated Circuits for Optical Communications*, McGraw-Hill, New York, NY, USA.

Reichl, G. [2003], WasteWater - a library for modeling and simulation of wastewater treatment plants, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 1–10.

Remelhe, M. A. P. [2002], Combining discrete event models and Modelica - general thoughts and a special modeling environment, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 203–207.

Rimvall, M. and Cellier, F. E. [1986], 'Evolution and perspectives of simulation languages following the CSSL standard', *Modeling, Identification and Control* **6**, 181–199.

Robinson, S. [2005], 'Discrete-event simulation : from the pioneers to the present, what next?', *Journal of the Operation Research Society* **56**(6), 619–629.

Robinson, S., Nance, R. E., Paul, R. J., Pidd, M. and Taylor, S. J. [2004], 'Simulation model reuse: definitions, benefits and obstacles', *Simulation Modelling Practice and Theory* **12**(7-8), 479 – 494. Simulation in Operational Research.

Rubio, M. A., Urquia, A., Gonzalez, L., Guinea, D. and Dormido, S. [2005], FuelCellLib - a Modelica library for modeling of fuel cells, *in* 'Proceedings of the 4^{th} International Modelica Conference', Vol. 1, Hamburg, Germany, pp. 75–82.

Rubio, M. A., Urquia, A., Gonzalez, L., Guinea, D. and Dormido, S. [2006], GAPILib - a Modelica library for model parameter identification using genetic algorithms, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 335–341.

Saadawi, H. [2004], 'An automatic teller machine (atm) in cd++'.
URL: *http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm*

Sahlin, P., Brign, A. and Sowell, E. F. [1996], The neutral model format for building simulation (v. 3.02), Technical report, Dept. of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden.

Sanz, V., Cellier, F. E., Urquia, A. and Dormido, S. [2009], Modeling of the ARGESIM "Crane and Embedded Controller" system using the DEVSLib Modelica library, *in* 'Proceedings of the 3^{rd} IFAC Conference on Analysis and Design of Hybrid Systems', Zaragoza, Spain.

Sanz, V., Jafer, S., Wainer, G., Nicolescu, G., Urquia, A. and Dormido, S. [2009], Hybrid modeling of opto-electrical interfaces using DEVS and Modelica, *in* 'Proceedings of the DEVS Integrative M&S Symposium, Spring Simulation Multiconference', San Diego, CA, USA.

Sanz, V., Urquia, A. and Dormido, S. [2006], ARENALib: A Modelica library for discrete-event system simulation, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 539–548.

Sanz, V., Urquia, A. and Dormido, S. [2007], DEVS specification and implementation of SIMAN blocks using Modelica language, *in* 'Proceedings of the Winter Simulation Conference 2007', Washington, D.C., USA, pp. 2374–2374.

Sarabia, D., Capraro, F., Larsen, L. F. and de Prada, C. [2009], 'Hybrid NMPC of supermarket display cases', *Control Engineering Practice* **17**(4), 428–441.

Schachinge, D. [2002], ''C13 Crane and Embedded Control' MATLAB hybrid approach', *Simulation News Europe* **35/36**.

Scheikl, J. [2001], ''C13 Crane and Embedded Control' - MATLAB numerical simulation event-oriented model', *Simulation News Europe* **31**.

Scheikl, J., Breitenecker, F. and Bausch-Gall, I. [2002], 'Comparison c13 crane and embedded control – definition', *Simulation News Europe* **35/36**, 69 – 71.

Schiftner, A. [2006], 'A Modelica approach to ARGESIM comparison 'Crane and Embedded Control' (c13 rev.) using the simulator Dymola', *Simulation News Europe* **16**(1), 30.

Schiftner, A., Breitenecker, F. and Ecker, H. [2006], ''Crane and Embedded Control' – definition of an ARGESIM benchmark with implicit modelling, digital

control and sensor action. revised definition – comparison 13revised', *Simulation News Europe* **16**(1), 27 – 29.

Shacham, A., Bergman, K. and Carloni, L. [2007], On the design of a photonic network-on-chip, *in* 'Proceedings of the First International Symposium on Networks-on-Chip (NOCS 2007)', Princeton, NJ, USA, pp. 53–64.

Shah, S. C., Floyd, M. A. and Lehman, L. L. [1985], MATRIX$_X$: Control design and model building CAE capability, *in* M. Jamshidi and C. J. Herget, eds, 'Computer-Aided Control Systems Engineering', Elsevier, Amsterdam, The Netherlands.

Stallings, W. [2000], *Operating Systems: Internals and Design Principles*, 4^{th} edn, Prentice Hall, Englewood Cliffs, NJ, USA.

Steinmann, W. and Zunft, S. [2002], Techthermo – a library for Modelica applications in technical thermodynamics, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 217–224.

Sun, W. [2001], 'DEVS model representing a simple automobile factory'.
URL: *http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm*

Tanenbaum, A. S. [2001], *Modern Operating Systems*, 2^{nd} edn, Prentice Hall, Englewood Cliffs, NJ, USA.

Tarjan, R. [1972], 'Depth-first search and linear graph algorithms', *SIAM Journal on Computing* **1**(2), 146–160.

Tocher, K. D. and Owen, D. G. [1960], The automatic programming of simulations, *in* 'Proceedings of the Second International Conference on Operational Research', Aix-en-Provence, France, pp. 50–68.

Urquia, A. [2000], Modelado Orientado a Objetos y Simulación de Sistemas Híbridos en el Ámbito del Control de Procesos Químicos, PhD thesis, Facultad de Ciencias, UNED, Madrid, Spain.

Urquia, A., Martin, C. and Dormido, S. [2005], 'Design of SPICELib: a Modelica library for modeling and analysis of electric circuits', *Mathematical and Computer Modelling of Dynamical Systems* **11**(1), 43–60.

van Beek, D. A. and Rooda, J. E. [2000], 'Languages and applications in hybrid modelling and simulation: Positioning of Chi', *Control Engineering Practice* **8**(1), 81–91.

Vangheluwe, H. L. M. [2000], DEVS as a common denominator for multi-formalism hybrid systems modelling, *in* 'Proceedings of the IEEE International Symposium on Computer-Aided Control System Design', IEEE Computer Society Press, pp. 129–134.

Volterra, V. [1931], Variations and fluctuations of the numbers of individuals in animal species living together, *in* R. Chapman, ed., 'Animal Ecology', McGraw-Hill, New York, pp. 409–448.

Wainer, G. [2002], 'CD++: A toolkit to develop DEVS models', *Software: Practice and Experience* **32**(13), 1261–1306.

Wainer, G. and Giambiasi, N. [2001], Timed Cell-DEVS: modelling and simulation of cell spaces, *in* H. S. Sarjoughian and F. E. Cellier, eds, 'Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies', Springer Verlag, New York, NY, USA.

Wang, L. and Kazmierski, T. [2005], 'VHDL-AMS - based hybrid approach to ARGESIM comparison 'C13 Crane and Embedded Control' with SystemVision', *Simulation News Europe* **43**, 30.

Weidinger, W. and Breitenecker, F. [2003], 'A classic CNS - solution to ARGESIM comparison 'C13 Crane and Embedded Control' using MATLAB', *Simulation News Europe* **38/39**.

Wetter, M. [2009], A Modelica-based model library for building energy and control systems, *in* 'Proceedings of the Eleventh International IBPSA Conference', Glasgow, Scotland, pp. 652–659.

Wöckl, J. and Breitenecker, F. [2003], 'A directly programmed solution to ARGESIM comparison 'C13 Crane and Embedded Control' with MATLAB', *Simulation News Europe* **37**.

Zeigler, B. P. [1976], *Theory of Modelling and Simulation*, John Wiley & Sons, Inc.

Zeigler, B. P. [1989], 'DEVS representation of dynamical systems: Event-based intelligent control', *Proceedings of the IEEE* **77**(1), 72–80.

Zeigler, B. P. [2006], Embedding DEV&DESS in DEVS, *in* 'Proceedings of the DEVS Integrative M&S Symposium', Huntsville, AL, USA.

Zeigler, B. P., Kim, T. G. and Prähofer, H. [2000], *Theory of Modeling and Simulation*, Academic Press, Inc., Orlando, FL, USA.

Zeigler, B. P. and Lee, J. [1998], Theory of quantized systems: Formal basis for DEVS/HLA distributed simulation environment, *in* 'Proceedings of SPIE', pp. 49–58.

Zeigler, B. P., Moon, Y., Kim, D. and Kim, J. G. [1996], DEVS-C++: A high performance modelling and simulation environment, *in* 'Proceedings of the 29^{th} Annual Hawaii International Conference on System Sciences', pp. 350–359.

Zeigler, B. P. and Sarjoughian, H. S. [2003], 'Introduction to DEVS modeling & simulation with JAVA: Developing component-based simulation models'.
URL: *http://www.acims.arizona.edu/PUBLICATIONS/*

Zimmer, D. [2006], A Modelica library for multibond graphs and its application in 3D-mechanics, Master's thesis, ETH Zürich.

A

Semaphores in Modelica

A.1 Introduction

Previous works describing Classic DEVS models in Modelica are Fritzson [2003] and Beltrame and Cellier [2006]. These works define the transmission of a DEVS message as a change in the value of a boolean variable. Thus, only one message can be transferred at a given time instant between two models using the same port. The behavior of Classic DEVS formalism is different from the P-DEVS formalism, where several messages can be transmitted at the same time instant (i.e., simultaneously) using the same ports. Also, several messages can be transfered using different ports.

The synchronization of message transmission, similarly to the mechanism used in TCP/IP communication, facilitates the transmission of simultaneous messages between ports. Modelica itself does not provide any method or feature for solving synchronization problems. A mutex mechanism was implemented in Modelica by Lundvall and Fritzson [2003], and was applied to the dining philosophers problem. Several methods have been described in the literature to solve synchronization problems, such us mutex, semaphores, monitors and message queues. The analysis and development of a Modelica model that implements the semaphore synchronization mechanism is discussed in this appendix.

The semaphore mechanism was proposed by Dijkstra in 1965 [Dijkstra, 1965]. It is a widely spread method for process synchronization in operating systems [Stallings, 2000; Tanenbaum, 2001] and general purpose programming languages (like C, C++ or Java). Semaphores have been chosen due to its simplicity and ease of use.

The developed semaphore model has been used to synchronize the message transmission between P-DEVS models in the DEVSLib library. The development of this mechanism is based on the mentioned work on mutex synchronization in Modelica [Lundvall and Fritzson, 2003]. However, due to the poor performance obtained with the use of this method, it was not selected for the final implementation of the message passing mechanism in DEVSLib.

A.2 Semaphore Mechanism Description

A semaphore is represented by an integer variable. There are two types of semaphores: 1) *binary*, whose value can only be 0 or 1; and 2) *general*, which can get any positive integer value (including 0).

A semaphore can not be accessed directly. There are two operations to access a semaphore: a) *P (wait)*; and b) *V (signal)* [Dijkstra, 1965]. These operations are atomic. The execution of one of these operations restricts the access to the semaphore to any other operation, even if they were executed simultaneously.

The *P* operation decreases the value of the semaphore in a given quantity (1 for binary semaphores and a user-defined quantity for general ones). If the current value of the semaphore is greater or equal to that quantity, then the value is decreased and the semaphore is captured by the model that executed the operation. Otherwise, the model is blocked and it must wait until the value increases.

The *V* operation is the opposite to the previous one. It increases the value of the semaphore in a given quantity. If the value is greater than zero (i.e., there are no models waiting), the value of the semaphore is increased by that quantity. Otherwise, there is at least one model waiting and then the semaphore is obtained

by the first model waiting for an available quantity of semaphore value. In the case of binary semaphores, the first waiting model will always obtain the semaphore.

Once the access to the semaphore is obtained (after a P operation), the required actions with the shared resources protected by the semaphore are performed (i.e., the critical section). When the critical section is finished, the semaphore is released (performing a V operation).

A.3 Modelica Semaphore Model

The Modelica implementation of the semaphore mechanism is composed of two components included in a Modelica package named *Semaphores*: 1) the semaphore model *Sem*; and 2) the connector *SemPort* (see source code in Section A.5).

The *SemPort* connector contains four variables: p, v, *okp* and *okv*. The first two represent the previously described semaphore operations P and V. The other correspond to boolean variables used to notify the successful end of the executed operation.

The *Sem* model has two parameters, n and *initValue*, that correspond to the number of models connected to the semaphore and its initial value. Each semaphore has an array of n public connectors (i.e., SemPorts) for connecting with the models.

A model has to include a SemPort, in order to connect with one of the public SemPorts of the semaphore model. To access the semaphore, the p and v values of the SemPort included by the model have to be used.

The execution of a P operation is performed setting the value of the p variable of the connector to the number of semaphore units to capture. In a boolean semaphore, that value is always set to 1. The value of the v variable has to be set to 0 at the same time. If the operation is not performed immediately, due to unavailable semaphore units, the semaphore inserts the operation in the list of waiting processes. When the P operation is performed, and the semaphore is captured, it sets to "true" the *okp* variable of the connector.

The execution of a V operation is analogous to the P operation. The value of the v variable of the connector is set to the number of semaphore units to release. Also, the value of the p variable has to be set to 0. If any model is waiting for the released semaphore, it captures the semaphore and leaves the waiting queue. When the V operation is finished, the semaphore sets to "true" the *okv* variable of the connector.

A.3.1 Mutual Exclusion

The behavior of the developed model is described using an example of mutual exclusion between two processes. The processes are defined as Modelica models (their code is shown in Listing A.1).

```
model process
  parameter Integer num = 1;
  parameter Real timeInCS = 10;
  Semaphores.SemPort sem;
  Real timeNA( start = Modelica.Constants.inf);
  Boolean criticalSection;
  Boolean idle( start = true);
  Boolean nextAction( start = false);
  Boolean waiting( start = false);
algorithm
  when idle then   // ready for next critical section
    idle := false;
    waiting := true;
    sem.p := 1; sem.v := 0; // P operation
  end when;
  when pre(sem.okp) then // semaphore available, enter critical section
    waiting := false;
    criticalSection := true;
    timeNA := time + timeInCS;
  end when;
  when pre(nextAction) then   // end of critical section, free semaphore
    criticalSection := false;
    idle := true;
    sem.v := 1; sem.p := 0; // V operation
  end when;
equation
  nextAction = time >= timeNA;
end process;
```

Listing A.1: Modelica code of a process in the mutualExclusion model.

The two processes, P1 and P2, want to access simultaneously to a shared resource. It is necessary to use a semaphore S to synchronize the correct access to the resource in mutual exclusion. First of all, it is necessary to initialize the semaphore with the value 1, indicating that the resource is accessible. Both

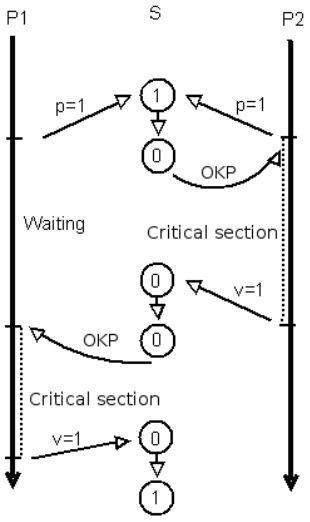


Figure A.1: Access to shared resource in mutual exclusion using semaphores.

processes will start simultaneously trying to get the semaphore. They perform a P operation with the semaphore. Only one of them, say P2, will get the semaphore, decreasing its value to 0. The other one, P1, will be blocked and starts waiting. P2 will be signaled with a true value in its *okp* variable of the connector. At this moment, P2 captures the semaphore and can execute its critical section (i.e., using the shared resource exclusively). In this case, the critical section is represented with a waiting time of 10 seconds (the value of the "timeInCS" parameter). After the critical section, P2 has to free the semaphore performing a V operation. The semaphore will detect the V operation, so it will wake up P1, which is still waiting. P1 receives the *okp* value thought the connector, indicating that now it has captured the semaphore and starts its critical section. When finished, P1 also has to free the semaphore. This behavior is graphically represented in Fig. A.1.

```
model mutualExclusion
  process p1;
  process p2( num=2);
  Sem s(initValue = 1);
equation
  connect(p1.sem,s.port[1]);
  connect(p2.sem,s.port[2]);
end mutualExclusion;
```

Listing A.2: Mutual exclusion model using a binary semaphore in Modelica.

The system has been modeled as shown in Listing A.2. The two processes are connected to a single semaphore, which represents the shared resource that has to be accessed in mutual exclusion. The model has been simulated during 100 seconds and the results are shown in Fig. A.2. Both processes alternate their state in the critical section, due to the use of the semaphore.

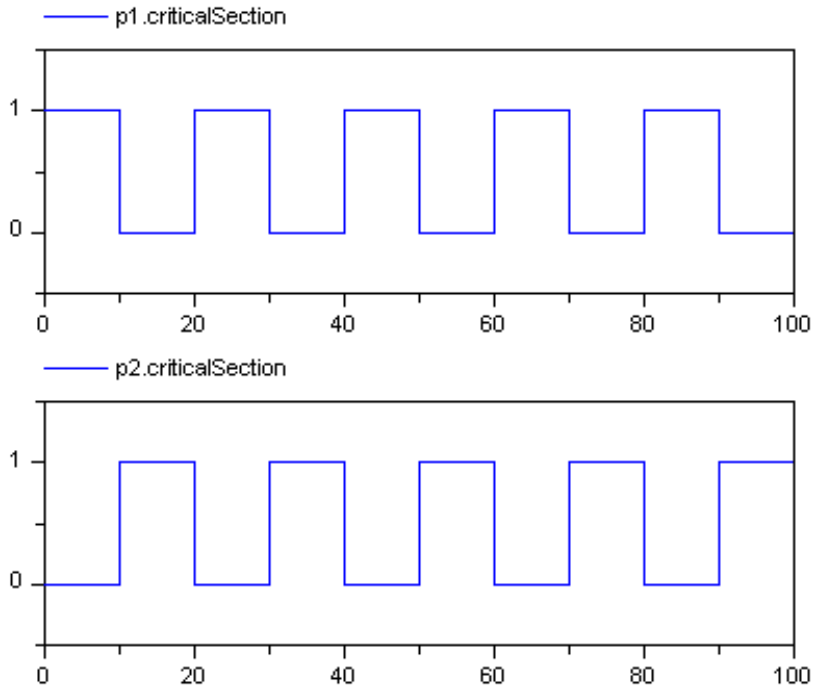


Figure A.2: Results of mutual exclusion model (processes alternate their critical sections).

A.3.2 Dining Philosophers

The dining philosophers represents a classic problem on concurrent activities and synchronization. It was proposed by E. J. Dijkstra in 1965.

The description of the problem is as follows. Five philosophers sitting at a table are doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each pair of adjacent philosophers, and so, each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. Each

philosopher can only use the forks on his immediate left and immediate right. The philosophers never speak to each other, which creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork (or vice-versa). Starvation might also occur independently of deadlock, if a philosopher is unable to acquire both forks because of a timing problem.

The problem has been modeled using Modelica and the developed semaphore model. A diagram of the model is shown in Fig. A.3a. Using the object-oriented functionalities provided by Modelica, the system can be easily adapted to other configurations. A model of a table with nine philosophers is shown in Fig. A.3b.

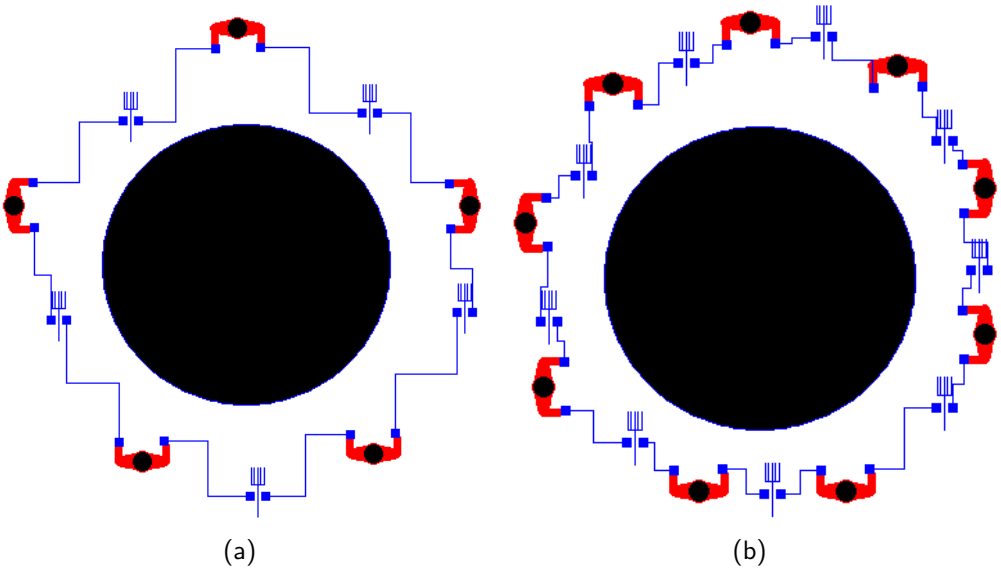


Figure A.3: Dining philosophers problem modeled using Modelica: a) five philosophers; and b) nine philosophers.

The model is composed of five "philosopher" models, and five "fork" models, connected as described before (see Fig. A.3a). Each philosopher is connected to the forks at his left and right hands. The connector, named *Hand*, contains three variables: *P*, of type SemPort, used to connect to the semaphores in the forks; *mine*, of type Integer, that represents the state of the philosopher; and *his*, of type Integer, that represents the state of the closest philosopher to this hand. The model of the fork is composed of: a semaphore, named *sem*, that represents the resource (i.e., the fork) that has to be captured (with start value of 1); and two connectors of type *Hand*, to connect with the philosophers adjacent

to the fork. The model of the philosopher is composed of: two connectors of type *Hand*, to connect with the adjacent forks; parameters and variables, to manage its state; and the code required to represent its behavior. The Modelica code for a philosopher is shown in Listing A.3.

```
model Philosopher
  parameter Integer num; // philosopher number
  parameter Real eatDelay = 1;
  parameter Real thinkDelay = 1;
  Integer state( start = 1); //(1 = thinking, 2= hungry, 3 = eating)
protected
  Real finishEating(start = Modelica.Constants.inf);
  Real finishThinking(start = 1);
  Boolean think, ishungry;
  constant Integer thinking = 1;
  constant Integer hungry = 2;
  constant Integer eating = 3;
public
  Hand Right;
  Hand Left;
algorithm
  when pre(ishungry) then // HUNGRY
   state := hungry;
    if mod(num,2) == 0 then
      Right.P.p := 1; Right.P.v := 0; // capture right fork first
      Left.P.p := 1; Left.P.v := 0;   // capture left fork
    else
      Left.P.p := 1; Left.P.v := 0;   // capture left fork first
      Right.P.p := 1; Right.P.v := 0; // capture right fork
    end if;
  end when;
  when pre(Right.P.okp) and pre(Left.P.okp) then // EATING
    state := eating;
    finishEating := time + (RandomUniform(time) + eatDelay);
  end when;
  when pre(think) then
    if mod(num,2) == 0 then
      Right.P.v := 1; Right.P.p := 0; // leave right fork first
      Left.P.v := 1; Left.P.p := 0;   // leave left fork
    else
      Left.P.v := 1; Left.P.p := 0;  // leave left fork first
      Right.P.v := 1;Right.P.p := 0; // leave right fork
    end if;
  end when;
  when pre(Right.P.okv) and pre(Left.P.okv) then // THINKING
    state := thinking;
    finishThinking := time + (RandomUniform(time) + thinkDelay);
    finishEating := Modelica.Constants.inf;
  end when;
end Philosopher;
```

Listing A.3: Philosopher model.

Each philosopher starts thinking. After a period of time (thinkDelay) the philosopher becomes hungry and tries to capture the forks. Each philosopher has a number assigned, that decides which fork has to capture first (in order to avoid

deadlocks) using a P operation. After capturing the first fork, the philosopher captures the other fork. If any fork is not captured, the philosopher will wait in the queue of the corresponding semaphore until it becomes available. When the philosopher captures both forks, he starts eating for a randomly defined period of time. After eating, the philosopher leaves the forks (using V operations) and starts thinking. After thinking, he will become hungry again.

The model has been simulated during 100 time units. The evolution of the state of each philosopher is shown in Fig. A.4. No deadlock nor starvation problems appear in the proposed solution. The results for the table with nine philosophers are equivalent to the ones obtained with five philosophers.

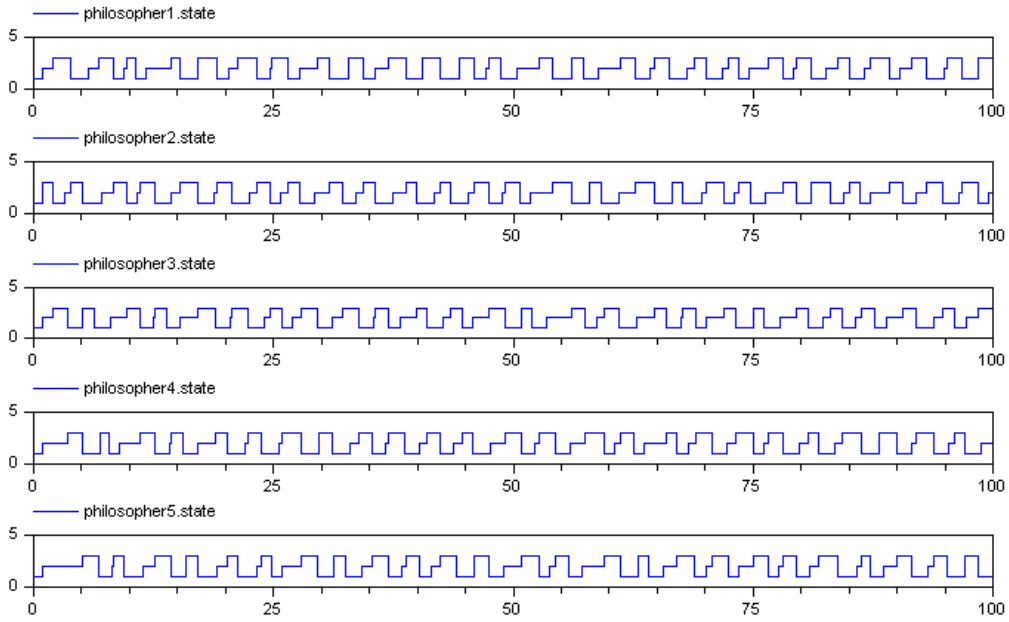


Figure A.4: Simulation results for the dining philosophers problem modeled using Modelica.

A.4 Synchronization of DEVS Message Communication Using Semaphores

The communication among AtomicDEVS (see Section 5.3) models has been implemented using the previously described Modelica semaphores. Input and Output ports in AtomicDEVS are composed of two SemPorts (*sync* and *ack*) and

the transmitted message (*event*). The structure of the communication using semaphores and the ports of the AtomicDEVS model is shown in Figure A.5.

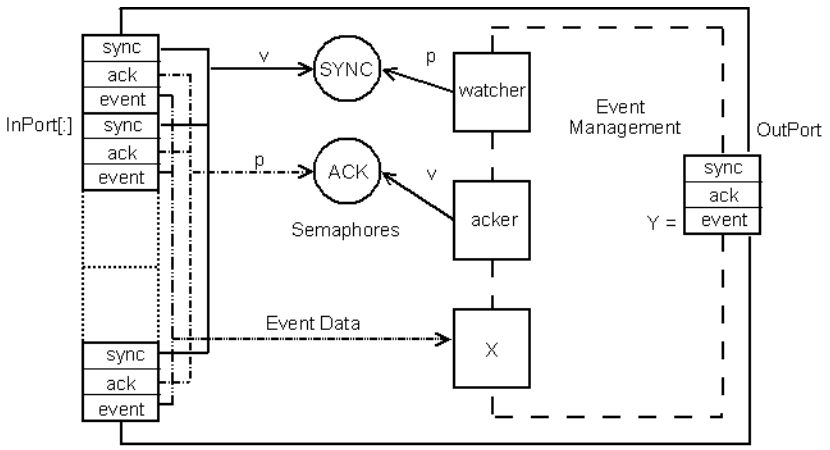


Figure A.5: Internal structure of the AtomicDEVS model.

Two general semaphores, *SYNC* and *ACK*, are used to synchronize the communication of messages among AtomicDEVS models. The *sync* and *ack* SemPorts declared in each input port are connected to these semaphores. The sender and the receiver must use the *SYNC* and *ACK* semaphores for signaling the message transmission and its successful reception. This mechanism is similar to the mutual exclusion problem previously described, and corresponds to a synchronous communication mechanism implemented using asynchronous methods (i.e., the semaphores).

The value of the *SYNC* semaphore represents the number of messages received. The *ACK* semaphore is only used to signal the correct reception of the messages. Initially, both semaphores have value 0.

The receiver performs a P operation over the *SYNC* semaphore to receive messages. If any message is available (value greater than 0) the receiver will read the message. Otherwise, it will wait in the queue of the semaphore until any message is received. When a message arrives, the receiver will: (1) decrease the value of the *SYNC* semaphore in one unit, (2) increase the value of the *ACK* semaphore (using a V operation) to signal the correct reception of the message; and (3) when ready to read more messages, decrease the value of the *SYNC* semaphore to receive more messages (using another P operation). If simultaneous

messages are received, they will be read using sequential P operations over the *SYNC* semaphore.

Every time the sender needs to send a message, it has to perform a V operation over the *SYNC* semaphore, increasing its value, to communicate the transmission of the new message. Simultaneously, it has to perform a P operation over the *ACK* semaphore, and wait for the confirmation of the correct reception of the message. When the confirmation is received (the sender leaves the waiting queue of the *ACK* semaphore), it becomes ready to send more messages.

A simple example of this kind of communication mechanism is shown in Listing A.4. The model is composed of three senders and one receiver, whose communication is synchronized using the described mechanism. In the example, the *SYNC* semaphore is named "sr". The source code of the receiver and sender is shown in Listing A.5.

```
model SenderReceiver
  sender s1(i=1);
  sender s2(i=2,delay=2);
  sender s3(i=3,delay=3);
  receiver r;
  Sem sr(n=4);
  Sem ack(n=4);
equation
  connect(s1.send,sr.port[1]);
  connect(s2.send,sr.port[2]);
  connect(s3.send,sr.port[3]);
  connect(r.receive,sr.port[4]);
  connect(s1.ack,ack.port[1]);
  connect(s2.ack,ack.port[2]);
  connect(s3.ack,ack.port[3]);
  connect(r.ack,ack.port[4]);
end SenderReceiver;
```

Listing A.4: SenderReceiver model communication using semaphores in Modelica.

The SenderReceiver model has been simulated during 10 seconds. The first sender (s1), sends a message every second. The second sender (s2), sends a message every two seconds. The third sender (s3), sends a message every three seconds. The evolution of the number of messages sent and received by each model is shown in Fig. A.6. Notice that simultaneous messages are transmitted and received correctly (e.g., three messages are received at time 6).

```
model sender
  parameter Integer i;
  parameter Integer delay = 1;
  SemPort send;
  SemPort ack;
  Boolean ini( start = true);
  Real x;
  Real nextsend( start = 1000000);
algorithm
    when ini then // send new message
      send.v := 1;  ack.p := 1;
      ini := false;
      x := x + 1;
    end when;
    when pre(ack.okp) then // ACK received
      send.v :=0;   ack.p :=0;
      nextsend := time +delay;
    end when;
    when pre(nextsend) <= time then // wait to send next message
      ini := true;
    end when;
equation
  send.p = 0;
  ack.v = 0;
end sender;

model receiver
  SemPort receive;
  SemPort ack;
  Real x( start = 1);
  Boolean ini( start = true);
algorithm
    when ini then // receive messages
      receive.p := 1;  ack.v := 0;
      ini := false;
    end when;
    when pre(receive.okp) then // confirm reception
      ack.v := 1;  receive.p :=0;
      x := x + 1;
      ini := true;
    end when;
equation
  receive.v = 0;
  ack.p = 0;
end receiver;
```

Listing A.5: Sender and receiver models.

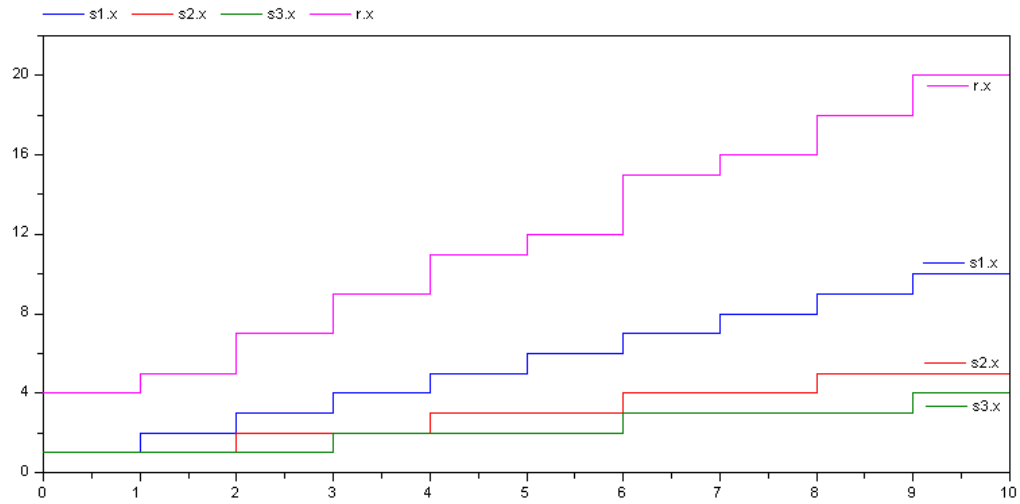


Figure A.6: Simulation results for the SenderReceiver model.

A.5 Semaphore Model Source Code

This section includes the Modelica source code for the Sem (semaphore) and semPort (connector) models.

```
model Sem "model of an IPC semaphore"
  parameter Integer num = 1;
  parameter Integer n = 2 "number of connections to the semaphore";
  parameter Integer initValue = 0 "initial value for the semaphore";
  Integer value(start = initValue) "current semaphore value";
  SemPort port[n] "connection ports";
  Integer releasesSem "last port that releases the semaphore";
  Integer getsSem "last port that gets the semaphore";
protected
  Integer p[n];
  Integer v[n];
  Boolean okp[n];
  Boolean okv[n];
  Integer waiting[n]
         "processes waiting on the semaphore for a number of resources";
  Integer waitorder[n] "order for the processes waiting";
  Integer waitpos( start = 1);
  Integer j;
equation
  p = port.p;
  v = port.v;
  okp = port.okp;
  okv = port.okv;
algorithm
  for i in 1:n loop
    // ***** SIGNAL (V)
    when pre(v[i]) > 0 then
      releasesSem := i;
      value := value + pre(v[i]);
      // find first process waiting in the queue
      j := 1;
      while j < waitpos loop
```

```
          if (waiting[waitorder[j]] <= value) and
             (waiting[waitorder[j]] > 0) then
            value := value - waiting[waitorder[j]];
            waiting[waitorder[j]] := 0;
            okp[waitorder[j]] := true;
            getsSem := waitorder[j];
            for k in j:waitpos-1 loop
              if k < n then
                waitorder[k] := waitorder[k+1];
              else
                waitorder[k] := 0;
              end if;
            end for;
            waitpos := waitpos -1;
            j := waitpos;
          else
            getsSem := 0;
            okp[waitorder[j]] := false;
            j := j +1;
          end if;
        end while;
        okv[i] := true;
      end when;
      // ***** WAIT (P)
      when p[i]  > 0 then
       if value == 0 then
          waiting[i] := p[i];
          waitorder[waitpos] := i;
          waitpos := waitpos +1;
          okp[i] := false;
       else
          getsSem := i;
          value := value - p[i];
          waiting[i] := 0;
          okp[i] := true;
       end if;
      end when;
      // ***** restore OK signal
      when {p[i] == 0 and (pre(p[i]) > 0) and value == pre(value),
            v[i] == 0 and (pre(v[i]) > 0) and value == pre(value)} then
        okp[i] := false;
        okv[i] := false;
      end when;
    end for;
  end Sem;

  connector SemPort
    Integer p;
    Integer v;
    Boolean okp( start = false);
    Boolean okv( start = false);
  end SemPort;
```

Listing A.6: Modelica source code of the Sem and SemPort models.