



LUND UNIVERSITY

Object-Oriented Modeling and Simulation of Hybrid Systems

Andersson, Mats

1994

Document Version:

Publisher's PDF, also known as Version of record

[Link to publication](#)

Citation for published version (APA):

Andersson, M. (1994). *Object-Oriented Modeling and Simulation of Hybrid Systems*. [Doctoral Thesis (monograph), Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:

1

General rights

Unless other specific re-use rights are stated the following general rights apply:

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: <https://creativecommons.org/licenses/>

Take down policy

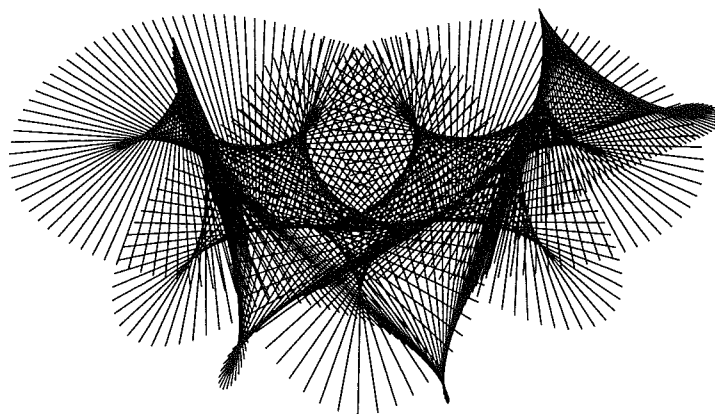
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Object-Oriented Modeling and Simulation of Hybrid Systems

Mats Andersson



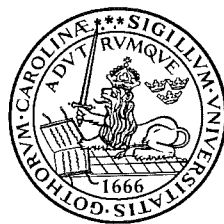
Department of Automatic Control, Lund Institute of Technology

— |

Mats Andersson

Object-Oriented Modeling and Simulation of Hybrid Systems

| —





ISSN 0280-5316
ISRN LUTFD2/TFRT--1043--SE

Department of Automatic Control Lund Institute of Technology Box 118 S-221 00 Lund Sweden		<i>Document name</i> DOCTORAL DISSERTATION		
		<i>Date of issue</i> December 1994		
		<i>Document Number</i> ISRN LUTFD2/TFRT--1043--SE		
<i>Author(s)</i> Mats Andersson	<i>Supervisor</i> Karl Johan Åström and Sven Erik Mattsson			
	<i>Sponsoring organisation</i> NUTEK 87-02503, 89-2740, and 92-03500 TFR 92-956			
<i>Title and subtitle</i> Object-Oriented Modeling and Simulation of Hybrid Systems				
<i>Abstract</i> <p>Models are important in engineering design and simulation. Existing modeling languages are often specialized for a particular engineering domain or they are specialized for a particular engineering task like simulation. Since good models are hard to develop, it is important that the representation and the modeling tools facilitate reuse of models and model components. This thesis defines two model representations: a high-level modeling language called Omola and low-level hybrid model formalism called OmSim. Omola is provided with powerful concepts for creating abstractions and hierarchical decompositions of models. Object-oriented concepts like classes and inheritance are used to support reuse of models and model components. Model behavior is represented as differential and algebraic equations and discrete events. Component interaction is defined by connections that relate terminal interfaces. The meaning of a connection is defined in terms of equations and event propagations, and it depends on terminal attributes. Composite models can be defined and displayed using graphical editors. In order to give a precise meaning to Omola models, a low-level formalism for hybrid models is defined. The formalism, called OHM, represent models as sets of variables, parameters, equations, event conditions, and event actions. It has a well-defined meaning in terms of mathematics and logic and it is intended as a common representation in an integrated environment of tools. An OHM can be analyzed algebraically. Well-known graph theoretical methods can be applied to sort equations and to analyze the degrees of freedom in the model and its index. Superfluous variables and equations can be eliminated to reduce the size of the model. Proper models can be translated into more specific representations suitable for numerical simulation. Methods for this are discussed in the thesis. OmSim is an implementation of an environment supporting development and simulation of Omola models. The architecture of OmSim is presented and some of the important design issues are discussed.</p>				
<i>Key words</i> Modeling; computer simulation; object-oriented modeling; systems representations; hierarchical systems; hybrid systems; simulation languages; computer-aided design; software tools; control engineering.				
<i>Classification system and/or index terms (if any)</i>				
<i>Supplementary bibliographical information</i>				
<i>ISSN and key title</i> 0280-5316		<i>ISBN</i> 0280-5316		
<i>Language</i> English	<i>Number of pages</i> 227	<i>Recipient's notes</i>		
<i>Security classification</i>				

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Fax +46 46 110019, Telex: 33248 lubbis lund.

Object-Oriented Modeling and Simulation of Hybrid Systems



The illustration on the front cover is a simulation of a double pendulum. The pendulum is modeled in Omola and simulated in OmSim. Matlab was used for drawing the picture.



Object-Oriented Modeling and Simulation of Hybrid Systems

Mats Andersson

Department of Automatic Control
Lund Institute of Technology
Lund, December 1994



To Ellen

Department of Automatic Control
Lund Institute of Technology
Box 118
S-221 00 LUND
Sweden

ISSN 0280-5316
ISRN LUTFD2/TFRT-1043-SE

©1994 by Mats Andersson. All rights reserved.
Printed in Sweden by Lunds offset AB
Lund 1994

Contents

Preface	7
Acknowledgements	8
1. Introduction	9
2. Modeling of Dynamic Systems	18
2.1 Introduction	18
2.2 Physical modeling of an electric drive system	22
2.3 Behavioral systems	32
2.4 The essence of object-oriented modeling	37
2.5 Object-oriented modeling environments	42
3. A Guided Tour Through Omola and OmSim	44
3.1 An Omola model of the servo	45
3.2 Simulating the servo	53
3.3 Creating new model variants	54
3.4 Summary	56
4. The Modeling Language Omola	57
4.1 Introduction	57
4.2 The class concept in Omola	61
4.3 Variables	63
4.4 Expressions	65
4.5 Equations	68
4.6 Class inheritance	68
4.7 Variable scope and binding rules	72
4.8 Model representation in Omola	76
4.9 Terminals	77
4.10 Connections	79
4.11 Variable components	84
4.12 Parameters	85
4.13 Model instantiation	86

Contents

4.14	Model parameterization	87
4.15	Discussion	90
4.16	Summary	92
5.	Modeling of Discrete Event and Hybrid Systems	94
5.1	Introduction	94
5.2	The sampled data model	97
5.3	Finite-State Automata	97
5.4	Generalized Semi-Markov Processes	99
5.5	The DEVS formalism	102
5.6	Hybrid Automata	105
5.7	Grafcet	107
5.8	Other approaches	111
5.9	A hybrid model for Omola	112
5.10	Simulation of hybrid models	115
5.11	Summary	120
6.	Discrete Events in Omola	121
6.1	Introduction	121
6.2	Defining discrete events	123
6.3	Event synchronization and propagation	129
6.4	A compact syntax for events	134
6.5	Discontinuities in continuous time equations	135
6.6	Examples of discrete and hybrid models	137
6.7	A mode switching physical model	147
6.8	Grafcet in Omola	148
6.9	Summary	154
7.	Modeling and Simulation Environment	155
7.1	Introduction	155
7.2	Modeling environment overview	157
7.3	Representation of classes	159
7.4	Representation of model instances	164
7.5	Simulation environment overview	168
7.6	Open architecture for OmSim	177
7.7	Summary	179
8.	Model Manipulation	181
8.1	Continuous time model	181
8.2	Discrete event model	192
8.3	Detecting continuous events	202
8.4	Summary	208
9.	Conclusions	209
10.	References	214
A.	Omola Grammar	221
B.	The Base Library	225

Preface

When I joined the CACE (Computer-Aided Control Engineering) project in 1986, the main objectives for the project at that time were to explore the possibilities of using increasingly powerful computers and work stations with graphical capabilities, to support control engineering. The focus of the project was soon turned to model representation and model development issues, since good models were recognized as essential in many fields of engineering. An early inspiration was the growing popularity of object-oriented programming. The idea of using an object-oriented programming language to represent dynamic systems was introduced in [Åström and Kreutzer, 1986].

Integrated environments of tools for *software engineering* was an active research area in the eighties. One interactive programming environment resulting from this research was based on the object-oriented language Smalltalk [Goldberg, 1983]. Some of the results on interactive environments are collected in [Barstow *et al.*, 1984] which inspired the CACE project to outline a design of an integrated environment of tools for control engineering [Andersson, 1989a].

Lisp was used in the early prototypes and experiments, but it was soon realized that a specially designed language for object-oriented model representation would be useful. Omola was then designed in late 1988 and presented in [Andersson, 1989c, Andersson, 1989b].

Omola was first used in a thesis by Bernt Nilsson [Nilsson, 1989] to demonstrate the main concepts of an *object-oriented modeling methodology* for chemical process modeling. At that time, there were no tools implemented that could use Omola for any practical purpose like simulation. The language was used for describing and discussing structured models.

In the beginning of 1990, a project was launched to implement a *kernel* for model representation based on Omola. The first release of the prototype environment was available for internal tests in the following year [Andersson *et al.*, 1991]. The tool was called OmSim (Omola Simulation Environment). It contained a graphical model editor and a simulator. Since then, Omola and OmSim have developed successfully and they have been used in several application projects. Some of them are listed in the conclusions.

The following thesis is a document of the CACE project, especially of the parts in which I have been mostly involved. This means, it documents the design of the language Omola and the implementation OmSim.

Acknowledgements

First of all, I would like to express my gratitude to Professor Karl Johan Åström who initially made me interested in the field of Computer Aided Control Engineering and invited me to participate in a very interesting project. He has been a constant source of inspiration and enthusiasm.

I am also very grateful to Sven Erik Mattsson, my closest supervisor and the leader of the CACE project. I admire him for his good humor and broad knowledge. He had made the CACE project into a team-work that I have really enjoyed. Sven Erik is also my closest friend at the Department.

Another member of the CACE team is Bernt Nilsson. He has been the fortunate user of our results and he has also contributed a lot. I specially would like to thank him for coping with the faults and sudden changes in the prototype implementations of OmSim. "Why isn't my very-large-model accepted today? It was okay yesterday!". I also would like thank Bernt for letting me always win on the 10k track at Skrylle. I miss it now.

I would like to thank Tomas Schönthal (another team member) and Dag Brück (a former team member) — it has been a great joy working with both of you. I am also thankful to Karl Erik Årzén who read a major part of the manuscript and gave very valuable comments and advice.

Most of all, I am very grateful to all the people at the Department for making it into such a fantastic place. In particular, I want to mention the ladies at the 5th floor (Agneta, Britt-Marie, Eva, and Eva) for always being very helpful and Leif Andersson for also being very helpful and for providing excellent computing and typesetting facilities.

This work has been supported by the Swedish National Board for Industrial and Technical Development (NUTEK contracts 87-02503, 89-2740, and 92-03500) and by the Swedish Research Council for Engineering Sciences (TFR contract 92-956).

Finally, I would like to thank my parents for always supporting me (*Tack för allt stöd*) and my beloved wife Ellen. To Ellen, I am also very grateful for her careful proofreading of the manuscript (*Täna väga!*). I am sure that all the remaining language errors have entered afterwards.

M.A.

1

Introduction

Models are descriptions of real or imaginary systems. A model formalizes knowledge about a system for a particular purpose. Models are used for analysis and design and they communicate knowledge between people and computers. For that reason, it is essential that models are written in a language with a well-defined syntax and meaning. It is also important that the language is readable by humans and allows the models to be defined in terms that closely relate to the application domain.

Mathematics and logic are languages suitable for representing behavior of models but they are poor on representing structural properties of large models. On the other hand, computer programming languages, especially modern object-oriented languages, are well suited for representing large systems in a structured and modular way, but they are poor on representing the behavior of physical processes. This thesis is concerned with the design of a modeling language that combines the powerful structuring mechanisms of object-oriented programming languages with a mathematical and logical behavior representation.

Physical modeling

Models are used in all kinds of engineering and technical problem solving. A good example of a general model is Newton's second law [Newton, 1687], aimed at describing the relation between the force and the motion of a particle:

$$m \frac{d^2 \bar{p}}{dt^2} = \bar{f} \quad (1.1)$$

The particle's mass is m , its position is \bar{p} , and the force acting on the particle is \bar{f} ; see Figure 1.1. Newton's law was a great step forward in science, since it explained the motion of celestial and terrestrial bodies. The application of the law as a general model for moving bodies was revolutionary, because it introduced the concept of an *internal variable* of

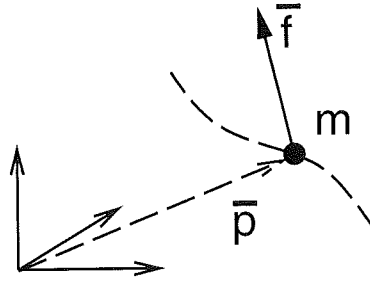


Figure 1.1 Simple system modeled by Newton's second law of motion.

the system, the gravitational force, which cannot be observed directly.

Assume there is a system for which Newton's equation is a valid model. The equation can then be used for solving at least the following practical problems:

- The particle's motion can be computed if the force and the mass are known. This is the typical *simulation problem*.
- The force acting on the particle can be calculated from an observation of the motion, if the particle's mass is known. The problem of computing an internal variable from observed quantities is called the *observer problem*.
- The necessary force to move the particle in a prescribed way can be calculated if the particle's mass is known. This is often called the *servo problem* or the *robot problem*.
- The particle's mass can be calculated if the force is known and the motion is observed. This is the *parameter estimation problem*.

Different numerical algorithms are used for solving the different problems. In order to use the model for a particular problem, the equation must be manipulated algebraically and translated into a form required by the numerical algorithm.

Computers are most often used for solving engineering design and system analysis problems. Traditional computer tools are constructed to solve numerical problems and they require the user to translate the model into a particular problem dependent form. This translation is complicated and error prone for models of large systems. A fundamental idea is to represent the models in a general symbolic form and use the computer for the necessary algebraic manipulations and translations. Some key concepts in modeling and model representation are illustrated by the following examples.

Engineering systems are rarely so simple that they can be described

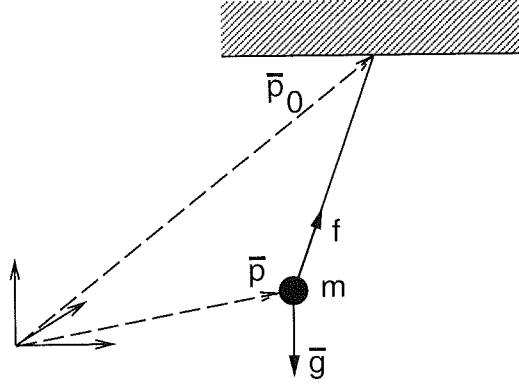


Figure 1.2 A pendulum modeled by Newton's law and two additional equations.

solely by a single equation. A *particle pendulum* is an example of a system that is slightly more complex than a free particle; see Figure 1.2. Equation (1.1) is still valid but the difference is that the motion of the pendulum is constrained. This can be modeled by an additional equation:

$$|\bar{p}_0 - \bar{p}| = l \quad (1.2)$$

where \bar{p}_0 is the position of the vertex and l is the constant length of the pendulum. The particle is also affected by a force:

$$\bar{f} = \bar{g} + f(\bar{p}_0 - \bar{p})/l \quad (1.3)$$

where \bar{g} is the gravitational force and f is the absolute value of the attachment force.

The pendulum model represented by the three equations (1.1), (1.2), and (1.3) can be regarded as a *specialization* of the free particle model represented by equation (1.1) alone. It is very common that new models are constructed as specializations of existing models. It is a very basic method in engineering and science to describe systems as special cases of more general system that are well known or defined elsewhere. Another basic method is to describe systems as a *compositions* of simpler subsystems. For example, a system with two pendulums can be modeled as a composition of two particle pendulum models. The system can be described without explicitly repeating the single pendulum model twice. It is better to refer to the single pendulum model defined earlier and instead focus on describing the *interaction* between the components. Figure 1.3 illustrates

Chapter 1. Introduction

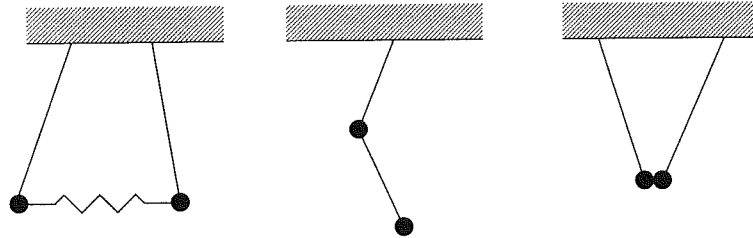


Figure 1.3 Examples of different component interactions in systems with two pendulums. The left figure indicates that the pendulums are connected by a spring, the middle figure shows one pendulum attached to another one, while the right figure indicates two pendulums that are sometimes colliding.

some possible configurations of systems consisting of two pendulums. The systems differ in the way the components interact.

The pendulum examples illustrate three basic principles in system modeling:

- A model represented in a general form can be used for solving different kinds of problems and as a component of different kinds of systems.
- Models are related as generalizations and specializations.
- Models are defined as compositions of interacting subsystems. Models of subsystems are defined separately.

The aim of this thesis is to study what kind of model representation and what kind of computer tools are needed to support these basic principles.

Object-oriented modeling

Object-oriented programming is a methodology in software engineering that is suitable for handling large systems. A basic idea in object-oriented programming is to view a system as a set of interacting objects. An object encapsulates data, behavior, and structure. Object-oriented program development typically starts with a top-down decomposition of the problem into objects, then the interfaces of the objects are defined, and finally their internal behavior is specified.

Object-oriented methodology can be applied to modeling of dynamic systems. Consider the rotating electric drive system in Figure 1.4. The system consists of an electric DC motor mechanically connected to a rotating load. The coupling between the rotor of the motor and the load is flexible in order to avoid strain in the rotating shafts and bearings. The purpose of the system is to drive the load at a certain prescribed rotating speed.

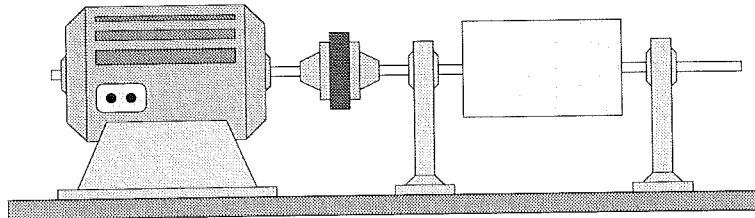


Figure 1.4 Rotating electric drive system.

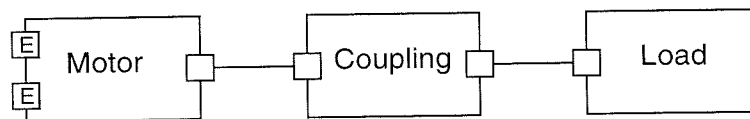


Figure 1.5 Decomposition diagram of the electric drive model. Interfaces marked 'E' are electric terminals while the remaining interfaces are rotating shafts.

It is natural to decompose the electric drive into three main components: the motor, the coupling device, and the load. From the physical structure of the electric drive system, it is realized that interaction occurs between the motor and the coupling and between the coupling and the load. The interaction is materialized by the rotating shafts. A shaft propagates a mutual torque and a common rotational speed between the components.

The complete system also interacts with the environment. In this simple example, the environment may consist of a controlled power supply for the DC motor. The electric power is supplied through two electric terminals; each one defined by an electric current and a voltage. A diagram showing the decomposition of the electric drive model is shown in Figure 1.5. The interfaces of the components are represented in the diagram as small squares. There are two types of interfaces: the electric terminals and the rotating shafts.

Having defined the top level decomposition and the interfaces, it is time to define the internal descriptions of the components. DC motors are often modeled by an equivalent electric circuit diagram and a rotating inertia with a viscous damping. A graphical representation of this decomposition is shown in Figure 1.6. The diagram contains iconic representations of a resistor, an inductor, a rotating inertia, etc. These are basic electric and mechanical components whose behavior can be described by simple mathematical relations.

One aspect object-oriented modeling methodology, illustrated by the

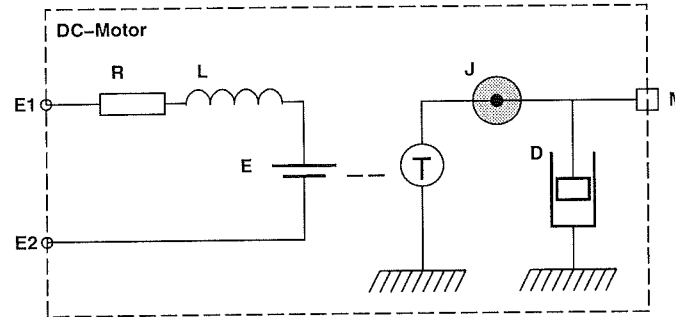


Figure 1.6 Decomposition diagram of the DC motor model. The left part of the diagram is a standard electric circuit diagram. The right part of the diagram represents the mechanical rotor consisting of a torque generator (T), a rotating inertia (J), and a viscous damping (D).

electric drive model, is that structure and interaction are defined *before* the behavior representations of the basic components are considered. Other aspects of object-oriented modeling discussed in the thesis are concerned with generalization and specialization of models in order to support reuse.

The need for a new modeling paradigm

Today's simulation languages are based on standards developed decades ago [Strauss *et al.*, 1967] when computers were much less powerful and symbolic formula manipulations and interactive user interfaces were much less developed. Many ideas were actually derived from analog computer simulation. The need for a more powerful model representation and better computer tools is obvious when real systems are considered. For example, an industrial scale chemical process may consist of thousands of components such as mechanical valves and pumps, chemical reactors and distillation columns, storage tanks, energy supply and recovery systems, electronic and computerized control systems, etc. Every component is normally described by several variables and equations and they are interacting in a complicated way. Engineering such complex systems is possible only because parts of the system can be considered in isolation. Even though relatively simple local models can be used in the design of many subsystems, models of the complete system are also needed. Global optimization of the process's operating conditions is economically significant. Recirculation of energy, material, and waste products is important for the preservation of the environment. This results in systems with very complicated interaction between the different stages in the process. Sim-

ulation is a basic tool for analyzing the behavior of such complex systems. Simulation is also useful for training operators and engineers.

Hybrid systems

Modeling and simulation of Discrete Event Dynamic Systems (DEDS) and Continuous Variable Dynamic Systems (CVDS) have developed as two separate cultures. Models used in automatic control are traditionally CVDS, based on differential and difference equations. DEDS are often used in modeling of manufacturing systems. However, real systems are usually characterized by a combination of continuous time and discrete event behavior. For this reason, there is a need for a model representation that combines the two paradigms. Models combining DEDS and CVDS are called *hybrid models*.

There is an emerging interest in control of DEDS. A DEDS model of a controlled process can be viewed as an abstraction of the real, basically continuous time, process. Methods for formal analysis and synthesis of DEDS control systems are still poorly developed compared to methods for CVDS. This makes it even more important to have modeling and simulation tools that can handle hybrid systems.

Scope of the thesis

This thesis results from a project in Computer Aided Control Engineering (CACE). One of the goals of the project has been to explore the possibilities to create an integrated architecture of software tools to support an engineer in the design and analysis of control systems. Models were recognized as being central at all stages of the engineering design process. However, it is hard to develop good models, and existing design and analysis tools provide poor support. Available simulation languages have their origins in the sixties when computers were much less powerful. Therefore, the CACE project turned the focus towards better model representations and more powerful development tools.

Figure 1.7 outlines an environment for computer aided control engineering. The model database is regarded as the central module. A graphical user interface is used for browsing the model database, for displaying and editing models, and for defining new structured models. Tools for symbolic manipulations are used for translating the models into representations suitable for solving various design problems. Numeric tools are utilized for design computations. Simulation tools are used for studying model behavior and for design verification. This thesis is mainly concerned with the representations used in the model database, the user interface for supporting model development, and the simulation tools.

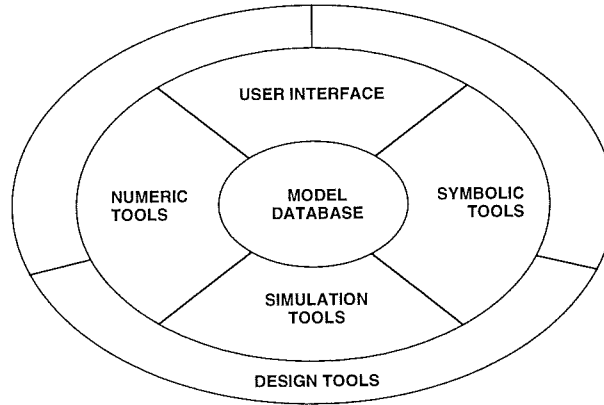


Figure 1.7 An integrated environment for computer aided control engineering (from [Andersson, 1990]).

The main topic of this thesis concerns the design of an object-oriented model representation and a modeling language for continuous time and discrete event dynamic systems. The language is called *Omola* which can be read out as *Object-oriented MOdeling LAnguage*. Omola was first outlined in [Andersson, 1989b, Andersson, 1989a, Andersson, 1989c] and described in some more detail in [Andersson, 1990]. The first version of the language was designed to represent continuous time models only. This thesis extends Omola to also represent discrete event and mixed models, thus making it a tool for working with hybrid systems. A more thorough rationale for the design of the language is also given.

Model representation in Omola is based on hierarchical model decomposition with structured component interfaces. Model behavior is defined in a declarative way based on algebraic and differential equations. Discrete behavior is described in terms of time and state events, and event propagation. A mathematical formalism for representing hybrid models is defined and used as a semantic basis for Omola models. Model reuse and database structuring are supported by object-oriented concepts of generalization and inheritance.

This thesis also documents and discusses the implementation of an environment for model development and simulation based on Omola. The environment is called OmSim (Omola Simulation Environment). The data structures used for representing Omola models, and the architecture of the model manipulation and simulation tools are presented. The symbolic manipulation methods used for transforming Omola models to efficient simulation code are also discussed.

Thesis outline

The thesis starts by giving a general introduction to physical modeling and object-oriented model structuring in Chapter 2. It is followed by another introductory chapter which is a guided tour through the OmSim modeling environment. Chapter 4 defines the basics of Omola as a language for representing structured continuous time models. Chapter 5 introduces the concept of discrete event and hybrid systems and gives an overview of formal representations of such models. It also defines a hybrid model formalism used as a basis for behavior representation in Omola. Chapter 6 extends Omola to also represent discrete event and hybrid models. Chapter 7 presents the architecture of the OmSim simulation environment. Chapter 8 discusses the symbolic model manipulations and structure analysis performed in order to prepare for efficient simulation of hybrid models. Finally, Chapter 9 contains the conclusions of the thesis.

2

Modeling of Dynamic Systems

Models formalize knowledge about systems. Models are needed in most kinds of engineering and design. They are used as a basis for formal design methods and for simulation experiments. Rothenberg writes [Rothenberg, 1989]:

A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger, and irreversibility of reality.

Modeling is a difficult task and it often requires expertise in the field of the particular engineering domain. Therefore, it is important that models are represented in a way that make them reusable for different purposes and by different people with less expertise in the field.

The first section of this chapter introduces the concept of structured physical modeling. An approach to model representations used in control theory called *behavioral systems* is discussed in the second section. The final section introduces the main ideas behind object-oriented modeling and model representation.

2.1 Introduction

Models are constructed for many different purposes. The modeling process itself is useful because it helps the modeler to structure and organize fragments of knowledge, so that he or she can get a deeper understanding of the system. Another important use of models is as a way of documenting the system and as a vehicle for transferring knowledge from the modeler

to other people. In the view of *knowledge engineering* and *expert systems* [Harmon and King, 1985], a model is a *deep knowledge* representation.

Structure is important in modeling and all kinds of engineering. A system is divided into parts of convenient sizes where each part can be analyzed or designed separately. Models should be represented in a way that reflects the topology of the system.

A drawback of most existing simulation languages is that they do not represent models on a format that encourages reuse of model components. They force the modeler to decompose a system, so that it fits in a particular representation framework dictated by, for example, a particular simulation algorithm. Object-oriented programming and software engineering tools support a methodology where the *structure* of the program is defined *before the implementation*. This means that conceptually clear module interfaces can be specified independently of the implementation. This is a desirable property also for a model representation environment.

There are different kinds of models for different purposes. This thesis is mainly concerned with *physical models* that can be represented by ordinary differential and algebraic equations, and discrete events. Related types of models are briefly mentioned and the concepts of physical modeling and structured representation are introduced in the following.

Qualitative and quantitative models

Models can be classified in two main groups: *qualitative models* and *quantitative models*. Qualitative models are made to answer questions about the qualitative behavior of the system like ‘What happens if the temperature is increased?’ or ‘What is a possible explanation for the low level in tank B?’. The behavior of a qualitative model is often described in terms of *qualitative variables* which take symbolic values like *low*, *medium*, and *high*. Qualitative physics is a research area within artificial intelligence (AI) where the goal is to capture common sense and engineering knowledge in qualitative models, so that conclusions about the behavior of a system can be inferred [Forbus, 1988]. Reasoning about a qualitative model is often referred to as *qualitative simulation* [Kuipers, 1986]. A qualitative simulation often results in a set of possible behaviors that can be inferred from the model. Examples of applications of qualitative models are alarm analysis and fault detection [Larsson, 1992].

This thesis is concerned with quantitative models. Quantitative modeling requires more detailed knowledge about the system. The purpose of a quantitative model is to give numerical answers to questions. Most variables of a quantitative model have values that are real numbers and the behavior is represented by real-valued algebraic expressions.

Continuous and discrete time models

A *continuous time model* is based on variables that are continuous functions of time. The variables are defined at every point within the range of time where the model is valid. These kind of models are sometimes called *Continuous Variable Dynamic Systems* (CVDS). A *discrete time model* is based on variables that are defined at a finite number of time points within any time interval. These kind of models are sometimes called *Discrete Event Dynamic Systems* (DEDS) or *sampled data models*.

Many real systems are more properly modeled as a combination of continuous time and discrete event behavior. These kind of models are called *combined models* [Cellier, 1979] or *hybrid models*. Discrete event and hybrid models are discussed further in Chapter 6.

Continuous time models can be classified in two groups: *lumped parameter models* which can be described by ordinary differential equations (ODEs) and *distributed parameter models* which are described by partial differential equations (PDEs). Distributed parameter models are often approximated by lumped parameter models since they can be analyzed and simulated more efficiently with standard numerical software. Only lumped parameter models are discussed in this thesis.

Obtaining and representing models

One way of obtaining models is often referred to as *black box modeling*. The system is viewed as a black box where some of the variables can be affected or measured. Modeling the system consists of selecting a general, parameterized mathematical representation and then tuning the parameters, so that behavior predicted by the model coincides with measurements from the real system. The process is called *system identification* and *parameter estimation* [Johansson, 1993].

Another approach to modeling, which in some sense is the opposite of black box modeling, is called *physical modeling*. Physical models are based on fundamental physical laws obtained from *first principles*, i.e., conservation laws for energy, matter, and momentum, and *phenomenological* equations. Physical modeling consists of dividing a system into fundamental physical mechanisms. The result is a set of equations or a *mechanistic model* defined as a network of ideal physical components and standard building blocks. For example, an electric circuit diagram is a network of ideal electric components like resistors, capacitors, inductors, and transistors. A mechanical model may be divided into a set of connected ideal springs, dampers, point masses, etc.

Each component in the mechanistic model is often simple enough to be represented by just a couple of equations. The network of the model

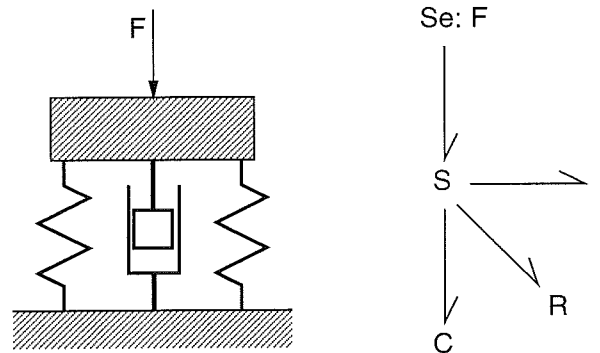


Figure 2.1 A mechanical system and its bondgraph representation. The mass is represented by an inertia element (I), the spring is represented by a capacitance (C), and the damper is represented by a resistance (R). The external independent force (F) is represented by an effort generator (Se). From [Thoma, 1975].

represents basic relations between the variables referred to by each component. *Flowsheet models*, used in process engineering, also fall into the category of mechanistic models. In this case the building blocks may consist of complex subunits like pumps, chemical reactors, and distillation columns. The flowsheet network explicitly represents flows of matter, energy, and control information.

A kind of physical modeling that focuses on the energy flows of the system is *bond graph modeling* [Karnop and Rosenberg, 1975]. An advantage with bond graphs is that they provide a common graphical language for modeling in different domains such as electric, mechanical, and chemical systems. A bond graph is a network of components and bonds. A bond represents energy interaction and consists of two variables: an effort variable and a flow variable. In mechanical systems, the effort variable is normally force and the flow variable is velocity, while in the electric systems the effort variable is voltage and the flow variable is electric current. The vertices of a bond graph consist of elements belonging to a small set of standard element types, and two kinds of *junctions*. Example of standard elements are *energy dissipation* (resistance element), *flow storage* (capacitance element), and *effort storage* (inductance element). The junction elements are *parallel junction* where the efforts are equal and the sum of the flows is zero, and *series junction* where the flows are equal and the efforts sum to zero. Figure 2.1 shows a simple mechanical system and its bond graph representation.

2.2 Physical modeling of an electric drive system

This section demonstrates by an example how a mathematical representation of a physical model can be structured in order to get a reusable model that is easy to understand and modify. The example is based on the electric drive unit introduced in Chapter 1 and shown in Figure 1.4. The electric drive is first analyzed and a mathematical model for the complete system is defined. The model is derived from first principles, i.e., using fundamental laws of mechanics and electric circuits. The model is then divided into components and the interfaces are determined.

It should be noted that the normal procedure of object-oriented modeling means that a system is first divided into components down to a level where each component is simple enough. Then the interfaces are defined and finally the equations of every *primitive* model component are defined. However, in this section, the modeling process starts by obtaining the total set of equations and the modularization is done afterwards. This is done in order to demonstrate important aspects of behavior representations based on general equations.

The first step in creating a model of the electric drive is to identify which aspects of the system that must be captured by the model. This mainly consists of selecting a set of relevant variables of the system, that the model must describe. The selection is guided by the intended use of the model and by the purpose of the real system. The main use of the electric drive is to rotate the load at a certain specified velocity. The system can also be used as a power generator or an electric brake. This means that quantities like the angular speed of the load and the voltage and the electric current of the DC motor are fundamentally important in the model. In this selection process we implicitly choose to neglect the effect of other quantities and to exclude them from the model. In our example, the temperatures in the system are not modeled.

The mechanical parts of the system are first considered. They mainly consist of two rotating masses: the motor armature and the load. In this first approximation, the coupling unit is assumed to have no inertia, or rather, that its inertia is included partly in the motor and partly in the load. Newton's law for rotating inertias results in the following equations; the first one for the motor and the second one for the load:

$$J_1 \frac{d\omega_1}{dt} = \tau_m - D_1 \omega_1 - \tau_1 \quad (2.1)$$

$$J_2 \frac{d\omega_2}{dt} = \tau_1 - D_2 \omega_2 - \tau_2 \quad (2.2)$$

The variables ω_1 and ω_2 are the angular velocities of the motor and the

2.2 Physical modeling of an electric drive system

load respectively, τ_m is the torque provided by the electric windings of the motor, and τ_1 is the torque transmitted to the load. The torque τ_2 is an external torque acting on the load. The constant parameters J_1 and J_2 represent the inertias of the motor and the load while D_1 and D_2 are damping coefficients representing the mechanical energy losses.

The coupling unit is modeled as a linear torsional spring and a damping proportional to the difference in speed between the motor and the load. This results in the following equation:

$$\tau_1 = K\delta + D(\omega_1 - \omega_2) \quad (2.3)$$

where the variable δ represents the torsion, i.e., the difference in angle between the motor and the load. The spring coefficient K and the damping coefficient D are constant parameters. The variable δ can be regarded as a state variable of the coupling unit¹, defined by the differential equation:

$$\frac{d\delta}{dt} = \omega_1 - \omega_2 \quad (2.4)$$

The rotating armature of the DC motor can be represented by the electric circuit diagram in Figure 2.2. The motor is assumed to have a constant magnetization. The voltage u_m is the total voltage across the motor's electric terminals, while the current i_m is the current through the armature coil. One equation is obtained from the electric circuit:

$$u_m = K_m\omega_1 + L_m \frac{di_m}{dt} + R_m i_m \quad (2.5)$$

The term $K_m\omega_1$ represents the voltage induced in the rotating armature coil. The motor torque τ_m is proportional to the armature current i_m according to the equation:

$$\tau_m = K_i i_m \quad (2.6)$$

The equations (2.1) to (2.6) constitute a mathematical model of the electric drive unit. The model contains eight unknown variables and six

¹ Instead of using the angular difference δ as a state variable, the angles of the motor and the load shafts could have been used directly. However, since these angles tend to grow constantly when the shafts are rotating, this representation will eventually give numerical problems when the small angle difference is computed. This consideration is based on good modeling and simulation practice but is irrelevant to the points illustrated in this chapter.

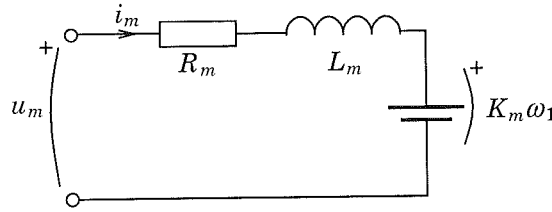


Figure 2.2 Equivalent electric diagram of the DC motor armature circuit.

equations. This means that the model itself does not determine any particular behavior, rather it represents *the set of all possible behaviors* obeying the physical laws stated by the model. Mathematically the model is an underdetermined system of equations. Two additional constraints are needed in order to solve the differential equations, so that a particular behavior can be observed, given necessary initial conditions. The additional constraints are imposed on the system by the environment. In other words, the electric drive system must be regarded as a part of a larger system, the environment, with which it is interacting. An important point is that *the model is versatile* in the sense that it does not imply exactly how the system is interacting with the environment. For example, the environment may impose a particular voltage u_m and a torque τ_2 . Given suitable initial values, all the other variables are then determined by the model. Another possibility is that the environment imposes the current i_m and the angular velocity ω_2 on the system.

Creating an abstract model

The mathematical model constructed so far is merely a set of equations. However, they represent an entity that is separated from the environment and can be regarded as an *object*. The model is intended to be used in different environments as a *module*. A well designed module is an *abstraction* of a subsystem. It means that it should be possible to use the module without knowing it in all details. In order to use a model as an abstract module it must be provided with an abstract interface. The interface must allow the user of the model to focus on the important aspects of how the system interacts with environment and disregard how the behavior is actually represented.

An abstract interface of a model includes a subset of its variables. These variables are called *external variables* or *terminals* in the following. The remaining variables are called *internal variables*. They belong to the model's *realization*, i.e., the internal representation of the behavior. Variables that influence or are directly influenced by the environment are

2.2 Physical modeling of an electric drive system

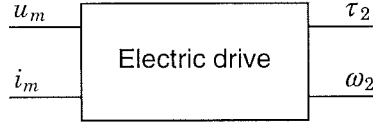


Figure 2.3 Acausal block representing the electric drive model.

chosen as external variables. Internal variables are typically auxiliary variables, introduced for convenience in the physical modeling process.

In the electric drive example, the variables selected as external variables are the electric quantities u_m and i_m of the motor, and the angular velocity ω_2 and the external torque τ_2 of the load. Any of these selected external variables can potentially be directly determined by the environment of the system. This is not the case for the internal variables ω_1 , τ_1 , τ_m , and δ . A graphical picture of the abstract drive unit model is shown in Figure 2.3.

The difference between the model in Figure 2.3 and a block in a block diagram representation is that no causality arrows are associated with the ports. The model is not implying any *computational causality*. If either one of the electric quantities and either one of the mechanical quantities are determined by the environment, then the other two quantities are determined by the model. There are four possibilities to assign computational causalities to the ports. It depends on the environment which one is used to compute the behavior of the system.

Decomposing the model

The model of the electric drive unit contains six equations. The system consists of three main parts: the motor, the coupling device, and the load. It is not easy for a user who only sees the equations, to understand how they are derived or how they can be modified to reflect a change in some part of the real system. For example, in order to change the model of the coupling, only two of the equations have to be considered but this fact is not explicitly reflected in the structure of the model. In fact, the model lacks internal structure apart from the raw equations.

It is desirable to decompose the model into submodels. A natural decomposition appeared already when the equations were derived from first principles. Equations (2.1), (2.5), and (2.6) are naturally associated with the DC motor, equations (2.3) and (2.4) naturally belong to the coupling device, while the equation (2.2) belongs to the load. Figure 2.4 shows an acausal block diagram of the decomposed model. The total set

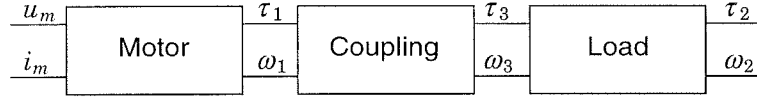


Figure 2.4 Decomposition of the electric drive model.

of equations are divided into three groups, one for each submodel:

$$\left. \begin{aligned} J_1 \frac{d\omega_1}{dt} &= \tau_m - D_1 \omega_1 - \tau_1 \\ u_m &= K_m \omega_1 + L_m \frac{di_m}{dt} + R_m i_m \\ \tau_m &= K_i i_m \end{aligned} \right\} \quad (\text{motor})$$

$$\left. \begin{aligned} \frac{d\delta}{dt} &= \omega_1 - \omega_3 \\ \tau_1 &= K\delta + D(\omega_1 - \omega_3) \\ \tau_1 &= \tau_3 \end{aligned} \right\} \quad (\text{coupling}) \quad (2.7)$$

$$\left. \begin{aligned} J_2 \frac{d\omega_2}{dt} &= \tau_3 - D_2 \omega_2 - \tau_2 \\ \omega_3 &= \omega_2 \end{aligned} \right\} \quad (\text{load})$$

Some of the variables considered as internal variables of the complete electric drive model are now appearing as external variables of the model components. Two additional variables, τ_3 and ω_3 , with two trivial equations, have been introduced. This was not necessary in order to decompose the model but it created a nice symmetry which is reflected in the diagram. The new variables also made the decomposed model more general in the sense that the coupling device need not be free of inertia and the load need not be a stiff shaft, as it is assumed in the original model. These restrictions can now be relaxed in the submodels without the need to change the decomposition. The decomposition made it possible to modify the model by replacing single components. Another way to decompose the electric drive model is shown in Figure 2.5. This decomposition is finer than previous three block structure. The graph shows an *idealized physical model* representation of the system. It consists of nodes and arcs where the nodes represent connections and the arcs represent idealized physical components like resistors, inductors, inertia-free springs, and viscous dampings (dash pots). This kind of model is sometimes referred to as a *mechanistic* model. Each component in the mechanistic model is described by a few equations. The nodes also have unambiguous interpre-

2.2 Physical modeling of an electric drive system

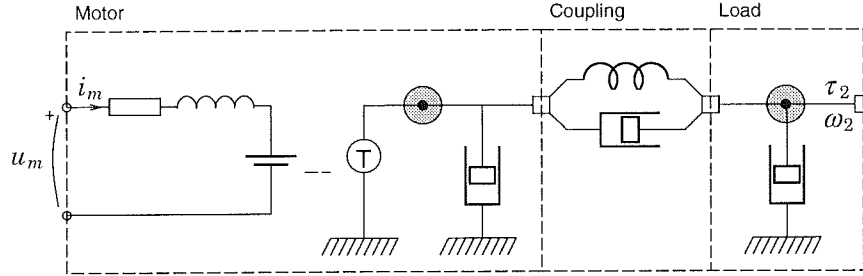


Figure 2.5 Object diagram of an idealized physical model of the electric drive.

tations as equations, relating the variables of the connected components. It is a straightforward procedure to translate the mechanistic model in Figure 2.5 into the equivalent mathematical model (2.7).

Different application domains use different, more or less standardized, mechanistic model representations. Figure 2.5 includes components from two different domains: electric circuits and one-dimensional rotational mechanics. Electric circuit diagrams are well standardized while mechanical diagrams are non-standard. The interpretation of the nodes in the mechanistic model graph is different for the different physical domains. A node in an electric circuit diagram represents an ideal, resistance-free electric wire. Components connected to the node must have the same electric potential and Kirchhoff's current law applies, stating that the sum of all current entering the node is zero. A node in the mechanical diagram of this example represents a cut in the mechanical system. The components connected to the node have the same angles and angular velocities, and the sum of the torques acting on each connected component is zero.

The mechanistic model in Figure 2.5 can be viewed as another level of decomposition of the three block model in Figure 2.4. The interconnections between the motor and the coupling, and between the coupling and the load, are indicated by the dashed vertical lines in Figure 2.5. Note that this kind of model decomposition is also acausal, i.e., inputs and outputs are not explicit in the model. Also note that the electric circuit diagram of the mechanistic model first appeared in Figure 2.2 and was used as a tool for deriving one of the original model equations. With a model representation accepting electric diagrams directly, the model developer need not bother about the actual equations. For electric circuit modeling there are special purpose simulation environments like SPICE [Nagel, 1975] that allows the model to be defined in terms of standard components.

Most mechanistic models consist of a small set of standard component types. Each component model contains only a few equations. Within a particular modeling domain, a handful of standard components are usually enough to model most systems within the domain. This makes it cost efficient to have *libraries of standard components*. Such libraries are easy to use and reduces the risk for errors compared to when equations are written directly.

The components of the mechanistic model are *proper objects*. For example, the torsional spring component used in the model of the coupling unit, contains a definition of its interface, a parameter for the spring coefficient, and a constitutive equation. It may correspond to a real physical component and it has a graphical representation in the diagram. More details about the representation of the components of the mechanistic model are given in Chapter 3.

Obtaining a causal model

Most traditional simulation languages, like ACSL [Mit, 1986], Simnon [Elmqvist *et al.*, 1990], or Simulink [Mat, 1992], require that each sub-model is represented on *state-space form*, where inputs and outputs are defined explicitly. This means that the equations must be written as:

$$\begin{aligned}\frac{dx}{dt} &= f(x, u) \\ y &= g(x, u)\end{aligned}\tag{2.8}$$

where u is a vector of inputs, y is a vector of outputs, and x is a state vector. We will now see how the electric drive model can be represented on this form.

It was previously found that the acausal model with four external variables allows four different input/output configurations. One possibility is to consider the motor voltage u_m and the external torque τ_2 as inputs. The motor current i_m and the load angular velocity ω_2 are then outputs from the model. It is now possible to determine the computational causality for each one of the tree blocks in the decomposition in Figure 2.4. The resulting causalities are indicated in the block diagram in Figure 2.6. With this particular choice of inputs it is possible to manipulate the equations to represent each block on explicit state-space form. The motor model

2.2 Physical modeling of an electric drive system

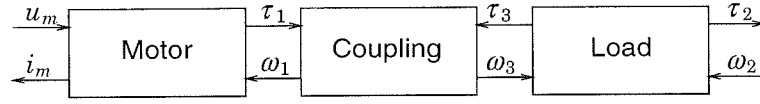


Figure 2.6 Block diagram for one particular choice of input terminals.

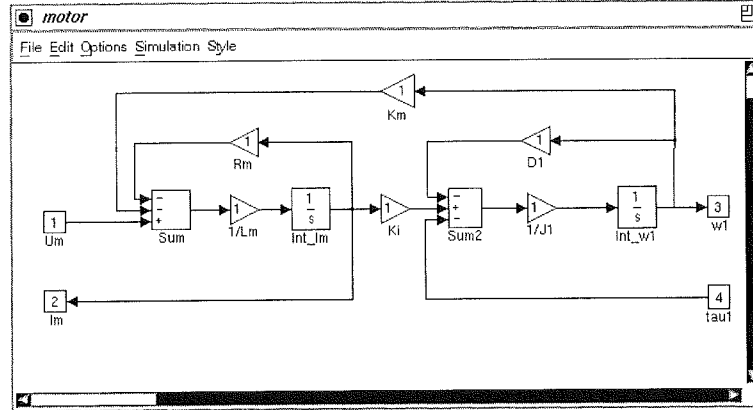


Figure 2.7 A Simulink model of the DC motor.

transformed into this representation results in the following equations:

$$\begin{aligned} \tau_m &= K_i i_m \\ \frac{d\omega_1}{dt} &= \frac{1}{J_1} \tau_m - \frac{D_1}{J_1} \omega_1 - \frac{1}{J_1} \tau_1 \\ \frac{di_m}{dt} &= \frac{1}{L_m} u_m - \frac{k_m}{L_m} \omega_1 - \frac{R_m}{L_m} i_m \end{aligned} \quad (2.9)$$

where u_m and τ_1 are inputs while the states ω_1 and i_m are directly used as outputs from the model. Note that the state-space model is not valid if any of the parameters J_1 and L_m are zero. This restriction did not apply to the original model which was valid also for such idealized cases.

Simulink allows a state-space model to be represented in a graphical form. A diagram of the state-space motor model in Simulink is shown in Figure 2.7. A graphical representation of a state-space model tends to show the *computational structure* of the model rather than the physical or conceptual structure of the system. While the physical structure of a subsystem is independent of the environment, this is not the case for the computational structure. This is demonstrated in the following.

A problem with the state-space representation appears if a different causality is chosen for the total model and the motor. If the power supply for the motor consists of an ideal current source, then i_m must be regarded as an input to the motor model. As a result, the voltage u_m becomes an output. However, it is not possible to represent this model on state-space form since it includes a differentiation of an input. It will then be necessary to simplify the model by removing the third equation in (2.9) and to remove the voltage u_m as an output.

Another problem with the modular state-space representation appears if the coupling unit is changed to a rigid connection between the two shafts. This is done in the original coupling model, consisting of two equations in (2.7), by removing the internal variable δ and replacing the first equation by $\omega_1 = \omega_3$. This modification is *local* to the coupling model object. However, with the new equation it is no longer possible to have the causality required by the other modules as indicated in Figure 2.6. The angular velocities ω_1 and ω_3 cannot be independent inputs, and the torques τ_1 and τ_3 are not determined by the coupling model alone. Nevertheless, the total set of equations are still a valid model. It is easy to realize that with a stiff coupling, the mechanical part of the electric drive consists of a single rotating inertia. Therefore, it should be possible to manipulate the equations and eliminate one state variable, so that the model can be written on state-space form.

It should be noted that assuming a stiff coupling is not a pathological idealization of the real system. The assumption may be just as valid as the assumption that the load subsystem can be modeled by a stiff rotating shaft. The problem with the model occurred only because of the chosen decomposition and because of the causal interfaces required by traditional simulation languages.

A model with changing causality

The electric drive system has so far been modeled as a pure continuous time system. As long as the environment affecting the system are changing continuously, also the variables in the electric drive model is changing continuously. Many real systems however, show behavior that can be regarded as abrupt changes. Such changes are most conveniently modeled as *discrete events*. A discrete event occurs at a certain moment in time and has zero duration. Examples of discrete events are switches in electric circuits, moving bodies that are reaching contact, a tank that becomes full, and a valve that is stuck because of high friction that suddenly starts moving. In these examples, the discrete events can indicate a dramatic change in the behavior of the system. This kind of events are

2.2 Physical modeling of an electric drive system

often referred to as *mode switches*. Models of mode switching systems can be viewed as a set of different continuous time models valid in different modes. The mode switching events are then representing a switch from one continuous model to another.

An example of a mode switching model occurs if a backlash is introduced in the coupling between the DC motor and the load in the electric drive model. It means that the shafts can move freely, independently of each other, within a small angular difference. It is natural to model the play as a separate submodel inserted between the motor and the coupling in the structured model in Figure 2.4. The interface of the backlash model must be similar to the coupling model, i.e., it connects to two rotating shafts each one represented by an angular speed, ω_a and ω_b , and a torque, τ_a and τ_b .

The backlash is modeled with three discrete modes:

Slack mode when the angular difference between the shafts is less than the constant amount Δ , and zero torque is transmitted.

Forward mode when the motor shaft is ahead with the constant amount Δ , $\omega_a = \omega_b$, and the torque is positive.

Backward mode when the load side is ahead with the constant amount Δ , $\omega_a = \omega_b$, and the torque is negative.

The internal variable δ is introduced to represent the angular difference between the shafts. It is assumed that the backlash is an idealized modeling element with zero inertia. The model is defined by two fixed and one mode dependent equation:

$$\begin{aligned} \frac{d\delta}{dt} &= \omega_a - \omega_b \\ \tau_a &= \tau_b \\ 0 &= \begin{cases} \tau_a & \text{in slack mode} \\ \omega_a - \omega_b & \text{in forward mode and backward mode} \end{cases} \end{aligned} \quad (2.10)$$

The continuous time variables of the model changes discontinuously with the mode. It is also important to note that the computational causalities depend on the current mode, and therefore, may change over time. In slack mode, ω_a and ω_b are inputs while the torques are outputs. In forward and backward mode, one of the angular velocities and one of the torques are inputs, depending on the environment model. This means that even if the environment of the model is known, it is not possible to represent this model on explicit input output form. An Omola model for the backlash will be given in Section 6.7.

Chapter 2. Modeling of Dynamic Systems

It should be noted that the backlash model only works as long as it is connected in series with a flexible structure like in the original coupling model. If the coupling was rigid, a switch from slack mode to forward or backward mode would physically mean that the speed of the rotating inertias must change discontinuously. This is non-physical and in addition, the model equations does not provide an initial value for the new common angular velocity. Still, the abrupt change in speed may be a reasonable modeling approximation for a rapid state change. These kinds of discrete state changes are discussed further in Chapter 5 and Chapter 6.

Summary

This section has demonstrated a methodology for physical modeling. First, the relevant variables of the system were identified. Secondly, a set of equations was determined from the physical laws governing the behavior of the system. Thirdly, an interface between the system and the environment was defined. Fourthly, the system was divided into components, each one with an abstract interface. This step consisted of dividing the equations into groups, based on which physical part of the system they belonged to. In this presentation, the model equations were introduced at an early stage but it was done only to demonstrate that the fundamental physical equations survive a model decomposition almost without changes. Normally, a system is first decomposed into subsystems and abstract interfaces are defined in several levels. Equations may be introduced at the lowest level of the decomposition, but it is often possible to use standard library components, thus eliminating the need for writing equations directly.

It was demonstrated by the example that a modularization and a decomposition must be done without any a priori assumptions about the computational causality of the model. Only acausal models are modular in the sense that they separate a subsystem from its environment and allow abstract interfaces to be defined at an early stage in the modeling process. A model of a subsystem on state-space form, with defined inputs and outputs, is just one particular representation of the model. A minor change in the assumptions about the environment or about a submodel may cause a state-space decomposition to collapse.

2.3 Behavioral systems

There is a growing interest in automatic control and systems theory to use a more general definition of a dynamic system than has been

2.3 Behavioral systems

used traditionally, i.e., transfer functions, matrix-fraction descriptions, and state space realizations [Kailath, 1980]. A more general definition of dynamic system is given in [Willems, 1986]:

DEFINITION 2.1

A *dynamic system* Σ is given by the triple $\Sigma := (T, W, \mathcal{B})$ where $T \subset \mathbb{R}$ is the *Time set*, W is the *space of external variables*, and $\mathcal{B} \subset W^T$ is the *behavior* of the system. \square

Usually the time set T is real for continuous time systems and integer for discrete time systems. The space of external variables W is often \mathbb{R}^q . The behavior \mathcal{B} is the set of all time trajectories, $w : T \rightarrow W$, obeying the behavioral laws of the system.

Definition 2.1 makes a clear distinction between the system as such and the representation of the behavior. Dynamic systems according to this definition are often called *behavioral systems* since they focus on the behavior, manifested by the external variables, rather than on the equations representing the system. An important property of behavioral systems is that the causality is not given a priori, i.e., the external variables are not separated into *input* and *output* variables. Rather, causality may be a property of a particular behavior representation.

The behavior of a dynamic system is most often defined by constraint equations. For a rather general class of dynamic systems the constraint equations are general equations in the external variables and their time derivatives, for continuous time systems, or time shifted variables for discrete time systems. Some behavior representations include *internal* variables (called *auxiliary* variables in [Willems, 1986] and *latent* variables in [Willems, 1991]). Internal variables are introduced for convenience and they often appear naturally when the behavior equations are written down from first principles. For example, internal forces in a mechanical system and the voltages across and the currents through each branch in an electric circuit are natural internal variables. A state space representation is a particular representation for a class of dynamic systems. The state variables are internal variables of the representation while the inputs and outputs constitute the external variables.

Interconnected subsystems are common in modeling, analysis, and synthesis of dynamic systems. In the behavioral approach, interconnected systems means that some of the external variables are common to several subsystems. This is equivalent to adding additional constraints to the system, so that attributes of different subsystems are forced to be equal. The external variables involved are called *interconnection variables*. They correspond to what in Omola is called *terminals*.

Linear model representations

An important subclass of dynamic systems consists of linear and time-invariant systems. Their behavior can be represented by the equation:

$$R(\sigma)w = 0 \quad (2.11)$$

where $R(\sigma)$ is a polynomial matrix in σ which denotes either the backward shift operator or the differential operator. The vector w is a vector of time trajectories representing the external variables. This representation is called an *autoregressive system* (AR-system) [Willems, 1986].

Another generic representation is the *pencil representation* which can be written as:

$$G\dot{\xi} = F\xi \quad (2.12)$$

$$w = H\xi \quad (2.13)$$

This representation is of first order and contains internal variables represented by the vector ξ . The vector w represents the external variables. F , G , and H are constant real matrices. Note that this behavior representation is acausal and does not distinguish between input and output variables. The pencil representation and other equivalent first order linear representations are discussed in [Kuijper, 1994]. Results concerning equivalence, minimality, and structural invariants of the first order linear representations are presented in that work.

As an example, the pencil representation of the motor submodel of the electric drive system is given. With the external variable vector as $w = \begin{pmatrix} u_m & i_m & \tau_1 & \omega_1 \end{pmatrix}^T$ a pencil representation of the motor becomes:

$$\begin{pmatrix} 0 & 0 & 0 & J_1 & 0 \\ 0 & L_m & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \dot{\xi} = \begin{pmatrix} 0 & 0 & -1 & -D_1 & 1 \\ 1 & -R_m & 0 & -K_m & 0 \\ 0 & K_i & 0 & 0 & -1 \end{pmatrix} \xi$$

$$w = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \xi$$

The pencil representation is obtained by trivial manipulations of the original equations. There is no loss of generality in this transformation. It is also trivial to construct a pencil representations for an interconnected system when each subsystem is represented by on pencil form.

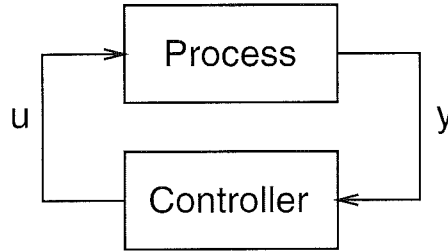


Figure 2.8 A causal model structure of a feedback control system.

Behavioral control systems

Input/Output representations are commonly used in control systems. A control system is normally viewed as a feedback system according to Figure 2.8. The causalities defined a priori in the interaction between the compensator and the controlled system are convenient because they simplify the analysis of the system. It means that the controller can observe the controlled system without affecting it and that it is free to affect the controlled process without being directly affected itself. However, from a physical point of view, it is impossible to affect a system without being affected oneself, and it is impossible to measure a quantity without affecting the observed system. The causal feedback structure is possible only because actuator and the sensor subsystems are regarded as part of the controlled system and that the interface between the controller and controlled system consists of *information channels*. In terminology of behavioral systems, this kind of interaction between dynamic systems is called *compliance free* [Willems, 1991]. Control design methods consider the controlled process as given. The problem is to construct the controller so that the specifications for the total system are fulfilled. However, it is not always a good idea to regard the sensors and the actuators as parts of controlled process. Better performance, or the same performance at a lower cost, can sometimes be achieved if they are considered as design parameters, and thus parts of the controller. Figure 2.9 shows two possibilities of structuring the control system. If it is desirable to include actuators and sensors in the design procedure, then the lower configuration is the natural choice. However, this interconnection between the controlled system and the controller need not be compliance free and the interaction causalities cannot be determined a priori. An approach to control design in a behavioral setting is presented in [Willems, 1993].

As an example of an acausal interconnection between a process and a controller, let us return to the electric drive system, discussed previously

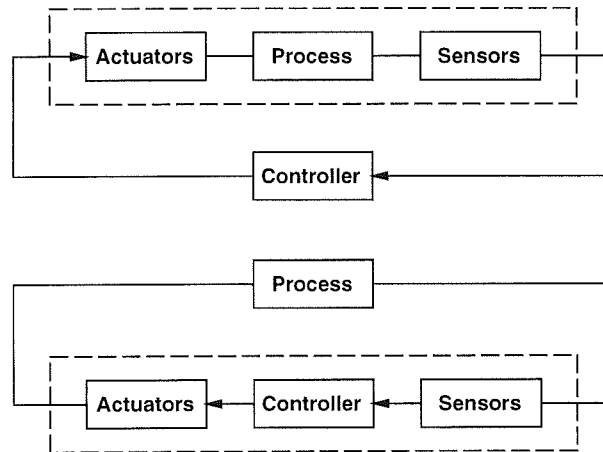


Figure 2.9 Examples of causal (upper) and acausal (lower) interconnections between the controlled and the controlling subsystems.

in this chapter. It is natural to regard the DC-motor and the coupling as a part of the controller subsystem and to design them together with the actual regulator. The sensor subsystem is not considered in the original example but it can likewise be regarded as a part of the controller. It was previously found, by analysis of the model equations, that if the coupling device is flexible, then the causality in the interaction between the coupling and the load is determined. This interaction is compliance free. On the other hand, if the coupling is rigid, the interaction is not compliance free and the causality cannot be established a priori.

Summary

The behavioral approach is motivated by the way physical systems are conceptualized and modeled from first principles. The same ideas motivate the object-oriented model representation which is the main topic of this thesis. The common basic principles are summarized in the following points:

- Systems are represented as interconnected subsystems.
- It is the behavior of a system, as it appears through the external variables, that is important, not the internal representation.
- Interaction causality is a property of the total system and cannot, in general, be defined for the components.

2.4 The essence of object-oriented modeling

The notion of *object-oriented modeling* as a method for structuring models of continuous time dynamic systems was introduced in [Nilsson, 1989, Nilsson, 1993]. The method is based on *object-oriented representation* of models. Omola is designed to provide such a representation and the language was used by Nilsson.

It is difficult to give a clear definition of what is meant by an object-oriented model representation but it is possible to characterize it by a list of properties. The main characteristics of object-oriented model representation are discussed in the following.

Declarative models

Models are *declarative* in the sense that they define facts and relations, rather than being procedures for computing data. For example, the following three models of a resistor are equivalent: $R \times I = U$, $R = U/I$, and $U/R = I$. The equation operator '=' is a symmetric equivalence operator; not an assignment as in some programming languages. Acausal physical models, discussed in the previous section, are examples of declarative models.

The declarative style is of course not suitable for all kinds of models. For example, models of information processing, like digital controllers, may be more convenient to represent as sequential procedures.

Modularity

Modularity is very important in all kinds of engineering. A module is a group of entities that are in some way related. A submodel in the electric drive system is an example of a module containing external variables, internal variables, parameters, and equations defining the relations between the variables.

A model representation must support modularity on multiple levels. A model may have submodels which have submodels themselves. This gives a hierarchical decomposition which can be arbitrarily deep. Figure 2.10 illustrates the concept. A model which has submodels is called a *composite model* or a *structured model*. Models that are not structured are called *primitive models*, since they are defined in terms of primitives like variables and equations.

A model (or submodel) encapsulates state and behavior in the same way as an *object* in object-oriented programming encapsulates data and procedures. For that reason we can regard models (and submodels) as objects.

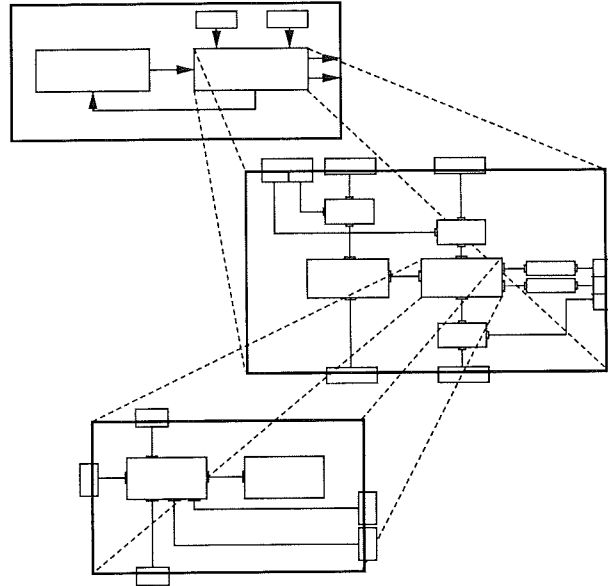


Figure 2.10 A multilevel hierarchy of submodels.

Models are not the only kind of modules that are of interest in a modeling and simulation environment. In this thesis, we are also considering a *library*, which is a set of model definitions, as a module.

Abstract interfaces

Abstraction is closely related to modularity. A module can be regarded as an abstraction if it can be used without knowing all its details. An abstract module has an *interface* and an internal description. The interface isolates the internal description from the environment, so that they can be considered separately.

Abstraction is the way to conquer the problem of complexity in large systems. It makes it possible to focus attention to one part or function of the system at a time. It also makes it possible for individuals in a group of engineers to become experts on different parts of the system.

Abstraction is an important concept in programming [Abelson and Sussman, 1985] and in the development of high-level programming languages. For example, FORTRAN supports only procedural abstraction while Pascal also supports data abstraction. Modula-2 has in addition the *module* as an important abstraction. Object-oriented languages like Simula and C++ use *objects* as the main abstraction units.

2.4 The essence of object-oriented modeling

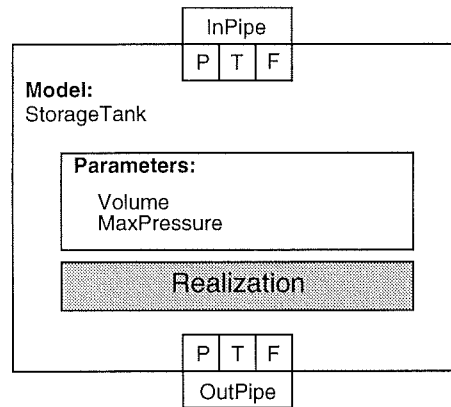


Figure 2.11 Example of an abstract model interface where the realization is hidden.

Abstract model representation makes it possible to use a model, for example, as a component in a larger model, without knowing all details about its definition. The interface of the model consists of terminals (external variables) and parameters. The rest of the model includes the internal representation of behavior, sometimes called the *realization* of the model. Figure 2.11 shows an example of an abstract representation of a tank model.

An important property of an abstract interface is the possibility to group variables that naturally belong together. For example, a pipe connection in a real system may be represented by three variables: the pressure, the temperature, and the flow. This idea is indicated in Figure 2.11.

Abstraction and modularity are closely related to *encapsulation* and *information hiding* [Booch, 1983, Rumbaugh *et al.*, 1991]. By encapsulation is meant that only module attributes belonging to the interface can be accessed outside the module. Different levels of encapsulation can be conceived in a model representation. For example, an internal variable of a submodel may not be accessed from the environment model but it may be available for observation when the model is simulated. Information hiding is a stronger form of encapsulation which means that the internal representation of a module is not visible to the user. Information hiding prevents the user to make assumptions about the internal representation. Such assumptions create interdependencies between modules which make it difficult to modify the system. Information hiding may also be desirable for proprietary reasons.

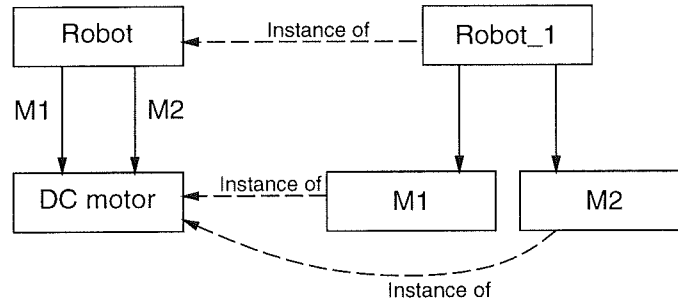


Figure 2.12 Relations in a model (class) and a model instance. The solid arrows represent component relations.

Classes

The *class* is an important concept of object-oriented programming. A class is a description of a group of objects with similar properties [Rumbaugh *et al.*, 1991]. Every object in an object-oriented program is an *instantiation* of a class. The word ‘*instance*’ is often used instead of ‘object’ to indicate that it is an object generated from a class. For example, *Person* is a class describing objects representing individuals. The class defines that all persons have attributes like name, age, address, etc. Every instance of *Person* have their own set of attribute values.

There are two possibilities of object-oriented model representation: to regard a model as a class or as an instance. The former one is chosen since a model and a class are conceptually similar. In other words, *models are represented as classes*, rather than instances since a model is normally regarded as a *description* of a system type, rather than a representation of a particular system. For example, a model of a DC motor describes the properties of a group of DC motor instances. A model defines attributes like variables and equations while an instance has values for each variable, representing the system at a particular moment in time. A model instance is needed when the system is simulated.

The difference between a structured model (class) and a model instance is illustrated in Figure 2.12. A robot has two DC motors. In the robot model, this is represented by two component links to the same DC motor model. However, an instance of the robot model has components which are separate instances of the DC motor model.

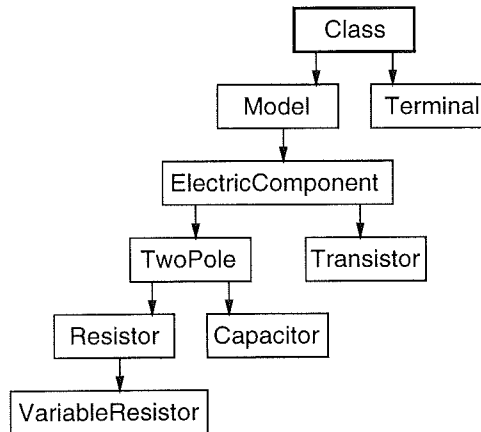


Figure 2.13 Example of a model representation inheritance tree. Each box is a class. The arrows are pointing at the subclasses.

Inheritance

Inheritance is a mechanism for sharing information between classes. A class can be defined as a *subclass* of another class, which is called the *super class*. The subclass *inherits* all attributes defined by the superclass. The subclass can define local additional attributes. This means that the subclass can be viewed as a *refinement* of a general concept defined by the super class. Sometimes we say that a subclass is *derived* from the super class which is also called the *base class*. An important property is the possibility for a class to *override* inherited attributes, and replace them with local definitions. This may be used as an advanced kind of model parameterization, making it possible to reuse model structure and replace single components [Nilsson, 1993].

A class can have any number of subclasses. If a class can have only one super class, the mechanism is called *single inheritance*. Otherwise it is called *multiple inheritance*. If all classes, except one *root class*, have exactly one super class, then the classes form an *inheritance tree*. Figure 2.13 shows an example of a model representation inheritance tree. The root of the inheritance tree is the class called *Class*.

The class nodes in the model representation tree play different roles. The leaf nodes are classes that can be directly used as components in a model. This also applies to some non-leaf nodes such as *Resistor*. Other nodes represent *generalizations*. For example, *ElectricTwoPole* generalizes the fact that resistors and capacitors are electric two-poles. *Resistor* and

Capacitor share a common definition of the terminal interface. Some class definitions may be empty in the sense that they do not define any local attributes. They are used to group other class definitions which conceptually belong together. For example, the class *ElectricComponent* may be empty but included in the tree to show that all classes derived from it are electric component models.

2.5 Object-oriented modeling environments

Omola is a language for object-oriented model representation that supports the essential properties of object-orientation, discussed in the previous section. A detailed description of Omola is given in Chapter 4. OmSim is an implementation of a modeling and simulation environment which is built around Omola representations. OmSim is presented in Chapter 7. A brief presentation of some modeling and simulation environments related to Omola and OmSim is given in the following.

Dymola

An important predecessor to Omola is Dymola [Elmqvist, 1978] which was an early general modeling language that recognized the importance of acausal equations and hierarchical submodels. Equation based modeling was first introduced in Speed-Up, a package for chemical engineering and design [Sargent and Westerberg, 1964].

Dymola is a symbol manipulating program that accepts an acausal, structured model and manipulates it into explicit state-space form for simulation in a standard simulation language like Simnon [Elmqvist *et al.*, 1990] or ACSL [Mit, 1986].

Originally Dymola was not object-oriented in the sense that was characterized above. However, it certainly supported hierarchical modularization and abstract interfaces. It also supported the concept of *model type* making it possible to reuse model definitions. A later, commercial version of Dymola has been extended in an object-oriented direction with a *model class* concept and inheritance [Elmqvist, 1994]. It is not possible to override an inherited definition in Dymola.

ASCEND

ASCEND (Advanced Systems for Computations in Engineering Design) is an object-oriented representation language and an environment for modeling and analysis [Piela, 1989]. The environment is mainly developed

2.5 Object-oriented modeling environments

to support the need for process modeling in chemical engineering and process design, but it has a general representation of models.

The language is based on type definitions (classes) with single inheritance. Both classes and instances can be directly manipulated in the language. Hierarchical decomposition and acausal submodels are supported but there are no special provisions for defining abstract interfaces.

Not only models but also problems can be specified in ASCEND. The environment includes tools for defining and browsing models. The main problem solver focuses on static simulation but also dynamic simulation is provided.

DYMON

DYMON is an object-oriented environment for modeling and simulation [Lund, 1992]. DYMON has a language (DYLAN) with properties similar to Omola. However, DYLAN supports a kind of multiple inheritance.

The environment is designed to support flowsheet modeling in chemical engineering but the aim is more towards dynamic simulation when compared to ASCEND.

DYMON include tools for interactive manipulation of model equations. An interesting facility is that versions of manipulated equation sets can be introduced in a model as new *facets*. Different facets may contain versions of the model of different complexity. For example, one facet may contain a representation manipulated for dynamic simulation, while other facets have representations appropriate for static calculations or design optimizations.

DIVA and VeDa

VeDa (Verfahrenstechnisches Datenmodell) is a frame based data representation for structured process models [Marquardt, 1992]. VeDa supports the general object-oriented concepts but also several application specific concepts. It allows models to be structured according to two aspects: a structural aspect and a *phenomenological* aspect [Räumschüssel *et al.*, 1993]. The VeDa model representation is intended to be implemented on top of an object-oriented database systems or a frame base knowledge representation system.

Models represented in VeDa can be transformed into simulation modules, represented by FORTRAN subroutines, for simulation in DIVA [Kröner *et al.*, 1990]. DIVA is an open architecture for interactive dynamic simulation of DAE (Differential and Algebraic Equation) systems.

3

A Guided Tour Through Omola and OmSim

OmSim is a tool for modeling and simulation of dynamic systems. It is based on the modeling language Omola which is the main topic of this thesis. This chapter serves as an introduction to Omola and OmSim and illustrates their main facilities by simple examples. It is specially directed to readers with no previous experience of Omola.

OmSim can be viewed as a collection — an environment — of tools which can be used more or less independently to define new models, to inspect and edit existing models, to simulate models, and to display simulation results. OmSim is window based and mostly controlled by mouse and menu selections. It is outside the scope of this thesis to describe all parts in detail but most of them are mentioned here in order to illustrate different aspects of Omola.

The main advantage of the object-oriented modeling methodology supported by Omola is that it supports reuse of models in several different ways. The concepts of reuse are best illustrated for models that are partly built from library components and for models that are made in several versions with minor differences. For this reason, the guided tour will start with an existing model, based on library components. This model is first examined and then modified and extended in different directions.

The electric drive system introduced in Chapter 2 is used to illustrate the use of Omola and OmSim. A controller is added to the electric drive, so that a servo system is created where the rotational speed of the load is regulated.

3.1 An Omola model of the servo

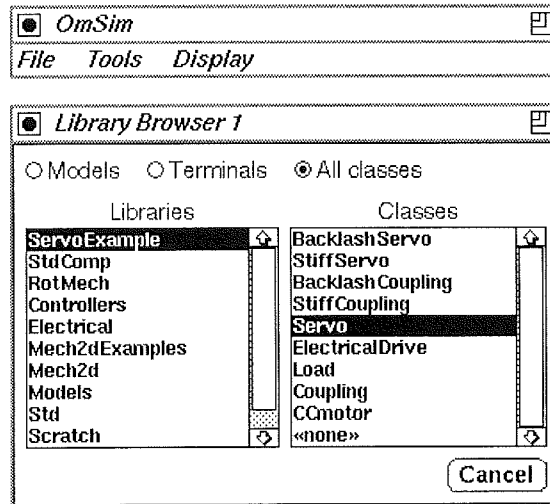


Figure 3.1 The OmSim library browser showing libraries and their contents.

3.1 An Omola model of the servo

An Omola model of the electric drive servo systems has already been prepared for this guided tour. We will start by examining this model to see how it is structured and how the components are defined.

Starting OmSim

OmSim is started with the Unix command:

```
omsim servo.om
```

The command argument is the name of an Omola file containing the servo model definition. This file is parsed so that the model is initially loaded into the internal model store of OmSim. When OmSim is started, three windows appear on the screen. The main OmSim window with the top-level menu bar and a *library browser* window are shown in Figure 3.1. The third window is a log window used by OmSim for printing various messages to the user.

The library browser shows a list of all model libraries currently known to OmSim. Each library may contain a number of Omola *class definitions*. An Omola class defines a complete model or a component intended to be used as a building block in a another class. When a library is selected

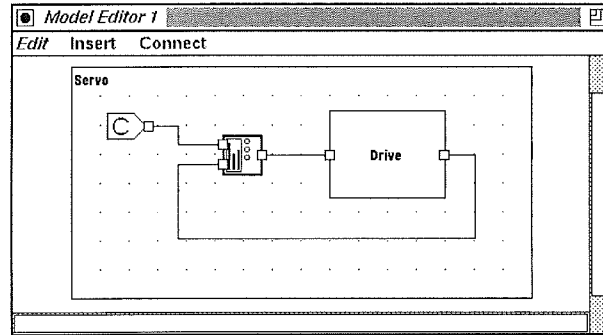


Figure 3.2 The top level of the servo model displayed in the graphical editor.

the contained classes are listed in the right part of the browser. It is then possible to select a particular class definition and to do various operations on it. Operations are activated from the pull-down menus called *Tools* and *Display*, located in the main OmSim window. Examples of tools are the graphical model editor and the simulator. The former one is going to be used next.

In order to examine the servo model it is selected in the library browser and a tool call *MED*, which is a graphical model editor, is activated from the *Tools* menu. The resulting window is shown in Figure 3.2. The servo model consists of three components: the drive unit, the controller, and a block representing a constant reference value for the controller. The constant reference value is included in this example only to make the model immediately ready for step response simulations.

The drive unit model is first investigated. The submodel icons in the MED diagram are mouse sensitive and have associated pop-up menus. Another MED window is opened from the menu of the drive unit. The new window, shown in Figure 3.3, displays the internal structure of the drive unit model. We can see that the servo contains three components: a motor, a coupling, and a load. Going one step further down in the component hierarchy the internal structure of the coupling model is shown in Figure 3.4. The hierarchical decomposition of the servo model in several levels can be displayed as a tree with the top-level servo as the root. The servo is selected in the library browser and the appropriate tool is chosen from the *Display* menu. The result is shown in Figure 3.5.

Object diagrams

The graphical model representations displayed and manipulated by MED are called *object diagrams* which are similar to ordinary block diagrams.

3.1 An Omola model of the servo

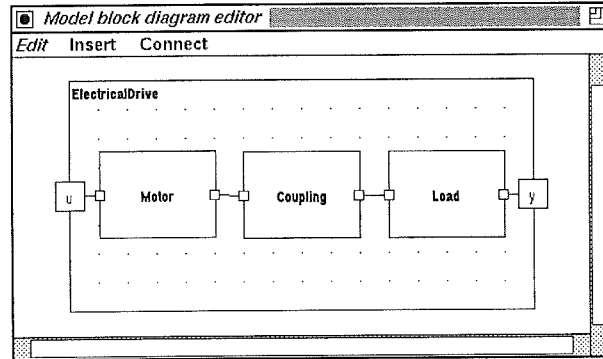


Figure 3.3 Object diagram of the drive unit

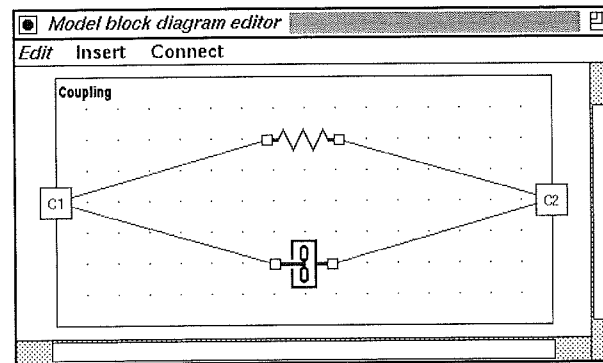


Figure 3.4 Object diagram of the coupling model.

Submodels appear as annotated boxes, like the drive unit in Figure 3.2, or with special icons indicating the type of the submodel like, for example, the controller in Figure 3.2 or the components of the coupling model in Figure 3.4.

Lines between objects in the diagram represent *connections* which define interaction between submodels. A connection is a relation between two *terminals*. Terminals represent the interaction variables (external variables) corresponding to inputs and outputs in ordinary block diagram models. However, a terminal is more general than an ordinary input or output variable, for example, it may consist of several quantities and the *causalities* (input or output) do not have to be defined. Terminals are in most cases indicated by small squares located at the rim of the icon or the object diagram. Figure 3.3 shows terminals at two hierarchical levels:

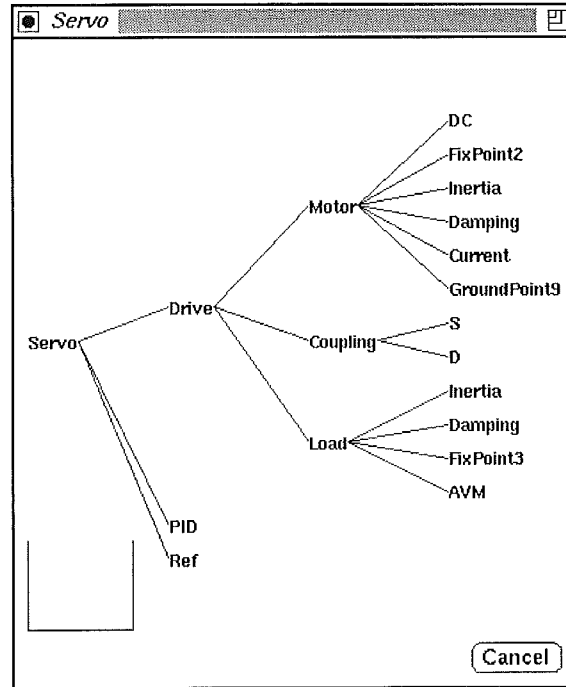


Figure 3.5 Submodel composition tree of the servo model.

the squares marked 'u' and 'y' are terminals of the drive model, while the smaller squares at the submodel boxes are terminals belonging to the components.

All parts of the object diagram displayed in MED are mouse sensitive, i.e., different menus are associated with submodels, terminals, and connections. The menus contain object specific operations like *move*, *delete*, and *show information*.

Library of rotational mechanical components

Let us return to the servo example. The components of the coupling model shown Figure 3.4 are derived from a general library of mechanical components called *RotMech*. It is developed as an example of a general library for a particular application domain and it contains idealized rotational components. The coupling model consists of an ideal torsional spring and a linear damper (a dash-pot).

All the model components defined in *RotMech* are organized in an *inheritance hierarchy* displayed in Figure 3.6. The root of the tree is

3.1 An Omola model of the servo

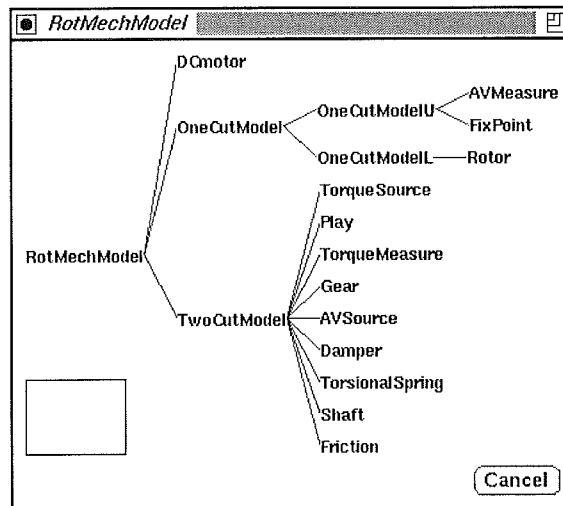


Figure 3.6 Inheritance tree of model classes in the RotMech library.

RotMechModel which in fact is an empty model definition included only to provide a common root for all model definitions in the library. The rest of the tree contains more or less specialized model definitions. The structure of the tree shows common properties of models. For example, Damper and TorsionalSpring, which appear as components of the coupling model, are both descendents of a definition called TwoCutModel. A *two-cut model* in this case is a component with two terminals, as it can be seen in Figure 3.4.

Some models in *RotMech* will be examined in detail below, after the terminals have been studied.

Terminal definitions

An important property of a library is that the components of the library have compatible interfaces that can be connected in a uniform way. This is achieved by defining a set of standardized terminal types used by all the components. The rotational mechanics library uses one basic type of terminal called a *cut*, defined by the Omola class Cut. The definition can be located in the library browser, selected, and displayed as Omola code. The result is shown in Figure 3.7. An Omola class definition often contains many names referring to other class definitions. These names are printed in bold-face in the display window and they are mouse sensitive, so they can be selected and displayed in another window. Consider

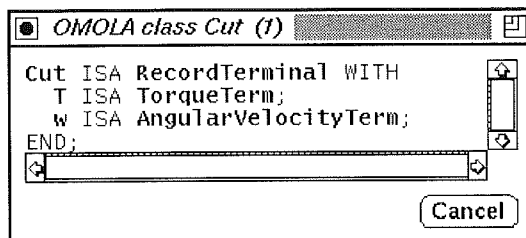


Figure 3.7 Omola definition of the cut terminal.

```
TorqueTerm ISA ZeroSumTerminal WITH
    default := 0.0;
    direction := 'in;
    quantity := "torque";
    unit := "N.m";
END;

AngularVelocityTerm ISA SimpleTerminal WITH
    quantity := "angular.velocity";
    unit := "rad/sec";
END;
```

Listing 3.1 Definitions of basic terminals in the RotMech library.

the definition of Cut, shown in Figure 3.7. The first line says that Cut is derived from the Omola class RecordTerminal, which is predefined in Omola. This means that it is known by OmSim and has a particular meaning. An object that is a record terminal is first of all a terminal, i.e., a part of the interface of the model where it is defined. It is also a kind of terminal that has several component terminals. The cut terminal has two components called T and w derived from the library definitions TorqueTerm and AngularVelocityTerm respectively. These definitions are listed in Listing 3.1. They represent single physical quantities and they are derived from ZeroSumTerminal or SimpleTerminal which are predefined classes in Omola. The meaning of a *zero-sum terminal* is that when two or more of these terminals are connected the individual quantities will sum up to zero. The corresponding meaning of a connection between two *simple terminals* is that their values are equal. The torque terminal has an attribute, default, which is used as a value if the terminal is not connected. Another attribute, direction, bound to in, defines that the terminal represents a torque from the environment acting upon the com-

3.1 An Omola model of the servo

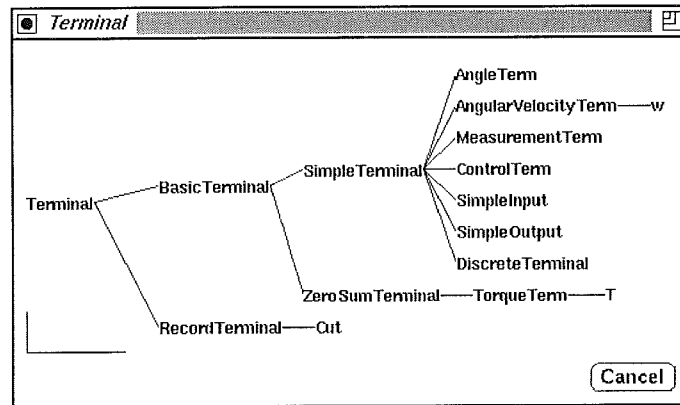


Figure 3.8 Inheritance tree of terminal definitions. The tree includes classes that are predefined in Omola as well as terminals defined the rotational mechanics library.

ponent. Note that this has nothing to do with causality, it merely defines how the sign of the torque must be interpreted.

The terminal definitions in Listing 3.1 also have attributes defining the physical quantities and the units of measure associated with the terminal interfaces. These attributes are checked by OmSim to assure that terminals of different physical quantities are not connected. The units of measure are used for automatic unit conversion in connections.

The physical interpretation of a cut terminal in the context of our particular library, is the following. A cut represents an attachment point of a mechanical object. When cuts of several objects are connected, the angles are forced to be equal at the attachment points, and the sum of the torques acting on each object is zero. The idea of a cut has the same meaning as in mechanical systems analysis. It can easily be generalized to libraries for 2-dimensional and 3-dimensional mechanical systems.

Since terminals are defined as Omola classes, they also form an inheritance hierarchy in the same way as the models. The inheritance tree with the predefined Omola class `Terminal` as a root is shown in Figure 3.8.

Basic model components

We are now ready to take a closer look at some of the model definitions in the library. Several of the model components in the library have two cut terminals. For that reason, a definition has been included that only defines the terminal interface of these models. The class is called `TwoCutModel` and

Chapter 3. A Guided Tour Through Omola and OmSim

```
TwoCutModel ISA RotMechModel WITH
terminals:
    C1 ISA Cut;
    C2 ISA Cut;
END;

TorsionalSpring ISA TwoCutModel WITH
parameters:
    K ISA Parameter WITH default := 1.0; END;
    % Spring constant in [Nm/rad]
variable:
    a TYPE Real;
equations:
    a' = C1.w - C2.w;
    C1.T = K*a;
    C1.T + C2.T = 0.0;
END;
```

Listing 3.2 Definition of a general interface class with two cuts and a specialization into a model of a torsional spring.

it is defined in Listing 3.2.

One of the components of the coupling model is the library model of a torsional spring defined in Listing 3.2. Since the spring has two attachment points it is defined as a *subclass* of `TwoCutModel`. This means that the spring model *inherits* the terminal interface definition from `TwoCutModel`. The definition of the torsional spring includes a parameter, K , with a default value, an internal variable, a , and three equations defining the behavior of the model. Variable a represents the twist angle. This is defined by the first equation saying that the time derivative of a is equal to the difference in speed between the cuts. The second equation defines the spring torque to be proportional to the twist. Since the spring is free of mass and inertia, a torque applied at one cut must be balanced by an equally large torque of opposite sign at the other cut. This is expressed by the last equation.

The *tags* written as ‘terminals:’, ‘parameters:’ etc. in the model definitions are not significant. They can be viewed as comments, added to the model to indicate the different groups of attributes. Normal textual comments start with a percent sign and last to the end of the line.

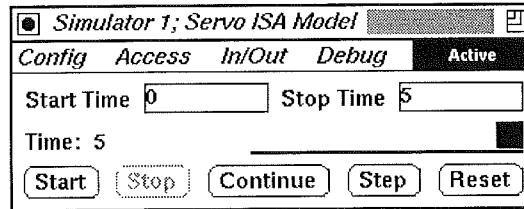


Figure 3.9 The top-level control panel of the OmSim simulator.

3.2 Simulating the servo

Having examined the servo model in some detail, we are now ready to simulate it. An OmSim simulator for the servo is created by selecting the model in the library browser and then choosing *Simulator* from the *Tools* menu. A control panel for the simulator appears as a new window shown in Figure 3.9. When a simulator is created, the selected model is first checked for consistency and then *instantiated*. Instantiation means that a special copy, an *instance*, of the model is created to be used for the actual simulation. All terminal connections of the model are translated into equations in the instance. The complete set of equations are then analyzed and manipulated in order to create efficient simulation code that can be executed by the numerical simulation algorithms. Errors are reported if the model contains too few or too many equations or if any other structural deficiency is detected.

The simulator panel has a menu bar with various options and subtools. From the *Config* menu it is possible to open a panel with several options and parameters that can be used for controlling the simulation. For example, it is possible to choose from a set of different integration methods, change the required accuracy, etc. The simulator has a set of different subtools for inspecting the model instance, changing model parameters and initial values, plotting results, connecting to external files, etc. From the *Access* menu it is possible to open a panel for displaying and changing all parameters of the model. The panel in Figure 3.10 is shown with all parameters having the default setting defined in the model.

Plotter windows are opened from the *In/Out* menu. Variables can be connected to plotters by means of menu selections in a *Variable Access* window. Two plotters are shown in Figure 3.11. They show the control signal and the load speed resulting from a simulation of the servo model with default parameter settings.

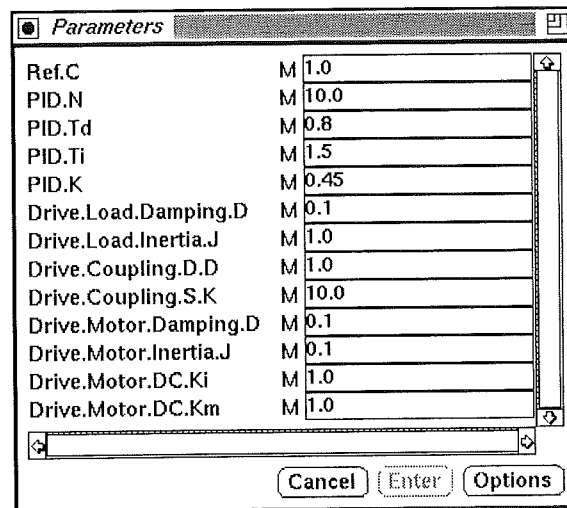


Figure 3.10 A parameter access tool for the servo model.

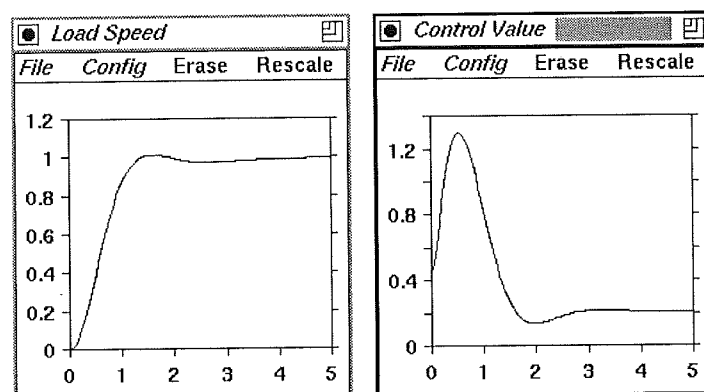


Figure 3.11 Plotter windows showing results from a simulation of the servo.

3.3 Creating new model variants

Assume we would like to simulate the servo system with a different model for the coupling of the drive unit in Figure 3.3. A coupling model with a mechanical play, or backlash, is defined using the graphical model editor. The result is shown in Figure 3.12. The backlash component is available from the library. It is different from the other components of the servo

3.3 Creating new model variants

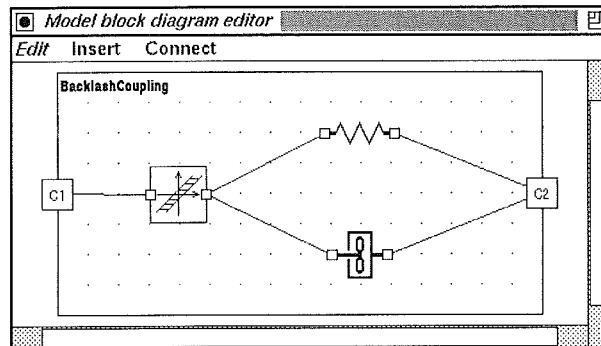


Figure 3.12 A coupling model a backlash component.

```
BacklashDrive ISA Drive WITH
  Coupling ISA BacklashCoupling;
END;
```

```
BacklashServo ISA Servo WITH
  Drive ISA BacklashDrive;
END;
```

Listing 3.3 Definitions of new versions of the drive and servo models.

model in the sense that it is not a pure continuous time model. The backlash uses discrete events to model the switches between different modes. The component is either in a slack mode with no transmitted torque when the angle difference between the cuts is less than a certain constant value, or in a torque transmission mode with a constant angular difference between the cuts. The definition of the model is given in Listing 6.12.

Since the new coupling model has the same interface as the old one, it may replace the old one in the drive model in Figure 3.3. Instead of actually redefining the drive, a subclass can be defined which inherits the structure from the original drive model but replaces the coupling component with the new version. The same thing is done to define a new version of the servo with the new drive model. The new definitions are shown in Listing 3.3. The definitions are very simple since most of the model structure is inherited from the previously defined models. This is an example of reuse of model structure.

3.4 Summary

The following points of interest have been encountered during the guided tour through Omola and OmSim:

- The same uniform schema, based on class definitions, is used for representing terminals, submodels, and other model components.
- Structured, acausal terminals make it possible to define model components representing idealized physical objects.
- A graphical editor is used for defining and displaying composite models.
- Discrete events are used for defining models with switching modes.
- Two kinds of hierarchies appear in models and model libraries: composition trees and inheritance trees.

Reuse of models and model components saves time and prevents errors when new models are developed. In addition to the traditional *copy-and-modify* methodology, Omola supports the object-oriented *inherit-and-specialize* methodology. The object-oriented method has several advantages:

- A library can grow organically, like a tree, where the origins of each new definition can be traced back to its root.
- Modification of the original model are automatically available to derived objects.
- Some of the structure intended by the library designer becomes explicit in the inheritance hierarchy. This makes it easier for the library user to find what is needed for a particular application.

4

The Modeling Language Omola

Omola is a language for describing dynamic models. Its main purpose is to provide a user oriented, high-level, and textual representation of models. Omola models can be used for simulation and as input to different analysis and design tools. Omola is designed to be a general *modeling language* rather than a more specialized simulation language. It is designed to serve as a general model representation in an integrated environment of tools supporting model development, system analysis, and design.

This chapter describes the fundamental concepts of Omola in detail. The syntax and the semantics of the language is defined. The presentation includes fundamental concepts and continuous time behavior only; concepts concerning discrete event behavior are discussed in Chapter 5 and Chapter 6.

4.1 Introduction

To design a simulation language means to choose a set of high-level modeling concepts, to give them a suitable syntactic form, and to give them a precise meaning in terms of some underlying formalism. A language design is influenced by several factors, for example

- the kind of models to be represented,
- the size and complexity of the models,
- the capabilities and requirements of the assumed underlying simulation or analysis machinery,
- the background and expertise of the presumed users,
- the traditions established by existing simulation languages and modeling formalisms,

- the stylistic preferences of the designer.

Omola is designed to represent models whose behavior can be described by ordinary differential and algebraic equations, difference equations, and discrete events. One of the main goals have been to include facilities for structuring large and very complex systems in such a way that they can easily be understood and maintained. The assumed simulation machinery consists of a state-of-the-art Differential and Algebraic Equation (DAE) solver in combination with a discrete event handler. The presumed users are engineers and researchers, skilled in their own application areas but not necessarily experts in simulation as such. Omola uses a mathematical notation common to many existing simulation languages and to matrix calculation systems. Notation for representing model structures is influenced by object-oriented programming and by *frames* and similar knowledge representation schemes developed in artificial intelligence (AI), e.g., see [Winston, 1984].

To describe a modeling language

The problem of describing a modeling or simulation language is similar to describing a general programming language. It is difficult to give a complete and rigorous description. A programming language is often described in terms of *syntax*, *semantics*, and *pragmatics* [Waite and Goos, 1984]. The syntax of the language defines which character strings constitute well-formed sentences. There are formal ways of describing syntax. For example, BNF (Backus-Naur form) and various extensions can be used for representing a context free grammar [Aho and Ullman, 1977]. A variant of BNF is used in Appendix A to define the syntax of Omola. The semantics of a language describes the meaning of a program or model in terms of basic concepts in the language. The pragmatics relates language concepts to concepts outside the language like mathematics or the operation of the executing computer.

A simulation language has many properties in common with programming languages. Syntax and semantics are in large parts identical. For simulation languages designed for discrete event simulation, also the pragmatics concerning the execution of the model is very similar to a programming language. A good example of that is the development of Simula [Birtwistle *et al.*, 1973], a language that was originally designed as a simulation language for discrete systems and was later developed into a general object-oriented programming language. However, the executional model of a modeling language mainly designed for continuous time simulation is considerably different from a programming language. Equations of a continuous time model are virtually executed in parallel

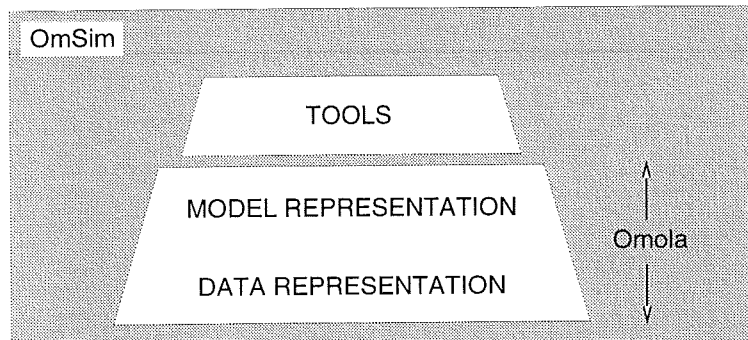


Figure 4.1 A layered view on Omola and OmSim.

while the assignments of a programming language are executed in strict sequence.

The syntax and meaning of basic Omola concepts are presented in the following sections. A formal description of the Omola syntax is given in a BNF like form in Appendix A. The semantic rules and interpretations of Omola concepts are summarized in special *Semantic Rule* paragraphs.

How Omola is structured and presented

Omola can be viewed as consisting of two separate layers: the *data representation layer* and the *model representation layer*, see Figure 4.1. The data representation layer defines a set of syntactic, semantic, and pragmatic rules for representing general data. Definitions of model variables and equations, as well as the basic structuring concept, the *class*, belong to this layer. The model representation layer consists of a set of predefined classes with a well-defined meaning. This layer defines the semantic and pragmatic rules governing how dynamic models are represented in the language. The first half of this chapter, up to Section 4.7, is mainly concerned with the data representation layer, while the remaining part is devoted to the model representation layer.

Figure 4.1 also shows a third layer, the *tool layer*, on top of the model representation layer. This layer is not actually a part of the Omola language but it is a natural extension of the two basic layers and it is a part of the modeling and simulation environment OmSim, described in Chapter 7.

The advantage of viewing Omola as consisting of two separate levels is that a degree of modularity is achieved and it becomes possible to replace any of the layers separately. For example, the data representation

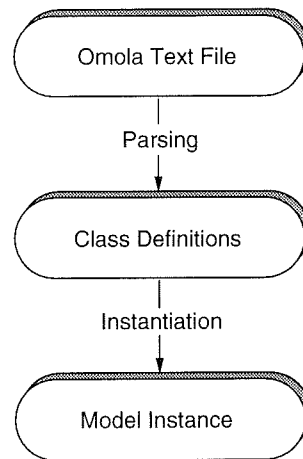


Figure 4.2 Representations and transformations of Omola models.

layer can be replaced by an object-oriented database or a standardized data modeling language like EXPRESS [Hope *et al.*, 1991, Spiby, 1992]. It is also easier to extend the model representation layer in order to represent totally different kinds of models, such as PDE models or qualitative models.

Omola was originally designed to work as an experimental framework where different model representations could be tested. It has been a main goal to design the language in such a way that different modeling concepts can be tested while the data representation layer is kept conceptually clean and relatively stable. For that reason it was important to have flexibility in the language so that new modeling concepts could be added along the way. In the chosen language structure it is relatively easy to add new concepts at the model representation level while the data representation level is kept unchanged.

Another view on Omola focuses on the different forms in which models are represented in the OmSim environment, see Figure 4.2. Omola models are permanently stored as text. A parser scans an Omola text file and builds the internal data structures in the OmSim model database. Each class definition is represented as an object in the model database. Model instances can be created from model classes. A model instance is a data structure where each single part of a model, such as a variable, an equation, or a submodel, is represented by a separate object. Model instances are used during model analysis and simulation.

4.2 The class concept in Omola

The key concept in Omola is the *class*. A class is a general data aggregation which is the basis for representing different modeling concepts like model, terminal, parameter, etc. Just as in object-oriented programming languages, an Omola class is a description of an object and it can be used for constructing any number of instances. Model instances are created from Omola classes by an instantiation procedure which is discussed in more detail below. A model instance is a data structure that is needed for further analysis and simulation of a model.

A class has a name and a *super class*. The name must be a valid identifier according to the standard rules used in most programming languages. The super class is a reference to another defined class. The meaning of a super class is discussed in Section 4.6 below. For the moment it is enough to view it as a special kind of relation to another class. In addition, a class may have any number of *attributes*. The syntactic form of a class definition is the following.

```
<name> ISA <name of super class> WITH
    <body with attribute definitions>
END;
```

If the class has no attribute definitions a shorter form can be used:

```
<name> ISA <name of super class>;
```

The words ISA, WITH, and END are key-words in Omola and cannot be used as names. Omola is case insensitive but key-words are written in capital letters in this thesis. The key-word ISA should be understood as 'is a' and it has the synonym key-word ISAN.

As an example of a class definition, regard the definition of Model which is a built-in class in Omola:

```
Model ISA Class;
```

In this case, 'Class' refers to another predefined class.

The properties of a class are defined by its super class and its local attributes. There are six kinds of attributes; they are:

- other class definitions, called *component classes* or just *components*,
- variable definitions,
- variable bindings,
- equations,
- connections, and
- events.

Chapter 4. The Modeling Language Omola

These attributes are discussed in this chapter except for events which are discussed in Chapter 6.

Component classes

Much expressive power of a modeling language comes from the ability to create hierarchies of structural abstractions. A model may be represented as a structure of submodels which are models themselves.

Hierarchical structures are represented in Omola as classes with components, i.e., local class definitions inside other class definitions. The outer class definition is the owner of the component class. Component classes of a model are used for representing submodels, parameters, terminals, etc.

The meaning of a class with components is given by the following rule:

SEMANTIC RULE 1—Composition

1. A class *C* with a component *D* means that every instance of *C* includes a part which is an instance of *D*.
2. Given an instance of *C*, this component can be accessed using the name of *D* as a symbolic reference.

□

For example, the class definition

```
C ISA Class WITH
  D ISA Class;
END;
```

and one instance of *C* are shown as an object diagram in Figure 4.3. Classes are displayed with double framed boxes while instances are displayed with single framed boxes. Relations between objects are represented by arrows. The *has* relations are indicated by dashed annotated arrows while the *instance-of* relations are indicated by solid arrows. From Semantic Rule 1 we can conclude that a component in Omola works in two ways. It is a class definition and it is a declaration saying that an instance of the owner class has a component instance with the same name as the component class. Because of this, local class definitions in Omola have a different meaning than local class definitions in object oriented programming languages like Simula [Birtwistle *et al.*, 1973]. A local class definition in Simula does not automatically mean that the owner class also has a data attribute of the locally defined class. Data attributes must be declared explicitly. A motivation for the chosen construct is given in the discussion at the end of this chapter.

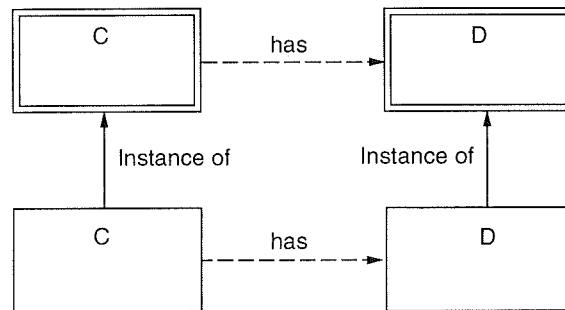


Figure 4.3 An object diagram showing a class with one component and one instance.

4.3 Variables

Variables are the primitive elements used as attributes of classes. Every variable has a name, a type, and possibly a *binding* which is an associated symbolic expression. Variable definitions with and without a binding have the following syntactical forms:

```
<name> TYPE <type>;
<name> TYPE <type> := <expression>;
```

The following is an example of a class definition with two variable attributes, *x* and *y*, where the latter has a binding expression.

```
C ISA Class WITH
  x TYPE Integer;
  y TYPE Real := 2*x;
END;
```

Variables represent numeric or symbolic quantities in an instantiated model. The meaning of a variable attribute can be defined in a similar way as the meaning of component attributes:

SEMANTIC RULES 2—Variable attributes

1. A class *C* with a variable attribute *x* means that every instance of *C* has a part which is a variable instance of *x*.
2. Given an instance of *C*, the variable instance can be accessed using the variable name as a symbolic reference.
3. A variable instance of a variable attribute is an object capable of storing a value of the data type defined for the variable attribute.

□

Chapter 4. The Modeling Language Omola

A binding is an expression defining how the value of a variable should be computed. If the value of the binding expression is defined, then the value of the variable is also defined to that value. A variable binding may also appear as a separate attribute of a class. In this case it has the form of an assignment equation, i.e., the variable name followed by the assignment symbol `:=` followed by the binding expression. Equation attributes are further discussed below. An equivalent definition of the previous Omola example with a separate binding attribute for `y` is

```
C ISA Class WITH
  x TYPE Integer;
  y TYPE Real;
  y := 2*x;
END;
```

A variable may have bindings defined outside the class where the variable itself is defined. However, at most one binding is valid and takes precedence over the others. For example, regard the following class definition which includes `C` as a component:

```
D ISA Class WITH
  C1 ISA C;
  C1.y := 1.0;
END;
```

The binding attribute refers to variable `y` defined in class `C`. This binding *overrides* the local binding for the same variable defined in `C`. Overriding is controlled by scope rules that are defined in Section 4.7.

A variable that has no binding is called a *free variable*. A variable with a binding that can be evaluated to a constant value is regarded as a constant.

Variable types

Omola has five basic data types. These are *real*, *integer*, *string*, *symbol*, and *enumeration*. In addition, one- and two-dimensional arrays of reals are supported. A *symbol* variable takes a value that is an identifier according to the lexical rules of Omola. A symbol literal is written as an identifier preceded by a single quote. An enumeration variable takes values restricted to a given set of symbols.

Table 1 lists the type specifiers as they are given in a variable definition, and examples of literals, for all Omola types. The type specifier for an enumeration is a list of possible symbolic values. The table shows a specific example of an enumeration type. Arrays are defined using the

Table 1 Omola types and examples of literals. Note that the type specifier for an enumeration is a list of arbitrary symbols; the table gives a specific example.

Type specifier	Example of literal
REAL	3.14
INTEGER	1
STRING	"This is a string"
SYMBOL	'OmSim
(Gas, Water, Steam)	'Gas
MATRIX[m,n]	[1.1, 1.2, 1.3; 2.1, 2.2, 2.3]
COLUMN[m]	[1.1; 2.1; 2.2]
ROW[n]	[1.1, 1.2, 1.3]

type specifiers *Matrix*, *Column*, and *Row*, which also include size declarations. *Column* and *Row* are one-dimensional arrays. The size declarations, indicated by *m* and *n* in the table, are general integer expressions that must evaluate constant values larger than zero. In the table, *m* indicates the number of rows in a matrix or a column while *n* indicates the number of columns in a matrix or a row. The notation for matrix literals is similar to Matlab [Mat, 1993].

Discrete and static variables

Special keywords can be added to a variable definition to define the variable as being *discrete* or *static*. A discrete variable represents a piecewise constant function of time in a dynamic model. The time instances where a discrete variable changes its value are called events which are further discussed in Chapter 6. A variable that is declared as *static* is not instantiated. It defines a property of the class itself and it is shared by all instances of that class. Static variables are similar to static class members in C++ [Stroustrup, 1986]. In Omola, static variables are normally used to define time independent model properties like the size of an array. Static and discrete variables are declared as:

```
<name> TYPE STATIC <type>;
<name> TYPE DISCRETE <type>;
```

4.4 Expressions

Expressions do not appear alone as class attributes but they are parts of equations, variable bindings, and in matrix size declarations. An expres-

sion is a combination of variable references, literals, mathematical operators, and invocations of predefined functions. Variable references are names of variables which are visible in the context of the expression. Which variables are visible and how they are referenced are discussed in Section 4.7. Operators and variables are combined to legal expressions according to a set of association and precedence rules which are similar to most programming languages and languages for numeric and symbolic mathematics. Every expression has a type that depends on the involved variables, functions, and operators.

Omola includes a basic set of mathematical and logical operators. Three special operators for matrix indexing, time derivatives, and conditional expressions are discussed below.

Matrix indexing

Variables of type Matrix, Row, or Column can appear in expressions with element or submatrix indices. The index operator consists of a pair of square brackets with one or two integer expressions separated by a comma. A variable of type Matrix must have two index expressions while a variable of type Row or Column must have one index expression. An index expression is either an integer expression or an integer interval. An interval is indicated by two integer expressions with the range operator consisting of two periods: “..”, in between. For example, assume there is a class with the following variable attributes:

```
X TYPE Matrix [5, 5];  
Y TYPE Row [10];  
i, j TYPE Integer;
```

The variable reference ‘X[1,1]’ refers to the scalar element of the first column and the first row of X, while the reference ‘Y[i]’ refers to any element of the single dimensional array Y. A matrix, row, or column which has interval indices (with intervals larger than one element) results in a submatrix. To continue the example, the reference ‘X[1,1..3]’ refers to the submatrix consisting of the first three elements of the first row in X.

It must always be possible to determine the size of a submatrix. This means that if the interval expressions contain variables, it must be possible to eliminate these variables from the difference between the upper and lower bound expressions. For example, ‘Y[i..i+2]’ is correct since the size of the submatrix does not depend on i while ‘Y[i..j]’ is not correct unless i and j are known constants.

Constant indices are checked in Omola classes. Variable indices are checked when the instantiated model is simulated. A row, a column, or a

matrix with a single element is accepted as a scalar variable everywhere in expressions.

Derivative operators

There are two equivalent ways of referring to the derivative of a variable. These are to use the function operator `dot` or to use single quotes as a postfix operator. For example, `"x' "` and `"dot(x)"` are equivalent and refer to the time derivative of `x` which must be a real or array expression. Second and higher order derivatives are indicated by several quotes or an additional integer argument to the `dot` operator. For example, `"x'' "` and `"dot(x,2)"` refer both to the second time derivative of `x`.

When the derivative operator is applied to an expression containing several real variables, Omola and OmSim may simplify the expression according to standard algebraic rules, or introduce an auxiliary variable equal to the expression. For example, the expression `"(x*y)'"` may be expanded to `"x'*y+x*y'"`. Algebraic manipulations of the equations are not a property of the language itself. They depend on the tools using the models and they are discussed in Chapter 7.

When the derivative operator is used together with matrix indexing, the correct syntax is to write, for example, `"dot(x[i])"` or `"x[i]'"`. Though, the meaning of the expression is `"x'[i]"` since the derivative operator is not applied to the integer index.

Conditional expressions

A conditional expression has the following syntactic form:

```
IF <condition> THEN <expression> ELSE <expression>
```

where the condition part should be a logical expression. If the condition evaluates to true, the conditional expression is equivalent to the first expression, otherwise it is equivalent to the second expression.

In general, the condition of a conditional expression is time dependent and cannot be evaluated in the model (before simulation). In this case the two alternative expressions must be of the same type which is regarded as the type of the conditional expression as a whole. However, if the value of the condition is a known constant, then one of the expressions is disregarded and not type checked. This is an advantage when a matrix has a parameterized size and a conditional expression is used for selecting a correct indexing depending on the actual size. The disregarded expression may in this case have illegal indices. For example, regard the following attribute definitions:

```
N TYPE Integer;
```

```
X TYPE Row [N];  
Y TYPE Real := IF N>1 THEN X[N-1] ELSE 0.0;
```

If the definitions appear in a class where N is bound to 1, then the THEN part of the conditional expression is illegal since the index is out of range, but this part is ignored since the condition is known to be false.

4.5 Equations

An *equation* is a class attribute for representing constraints between variables and it is one of the basic ways of representing model behavior in Omola. The syntactic form of an equation is a left and a right hand side expression with a separating equal sign:

```
<expression> = <expression>;
```

The equation has the following precise meaning which is consistent with an equation in the usual mathematical sense.

SEMANTIC RULE 3—Equations

An equation is an equality relation between two symbolic expressions declaring that a possible model behavior is such that the left and the right hand side expressions are evaluating to the same value. □

Note that the left and the right hand side expressions have equal status and can be switched without changing the meaning of the model.

Variable bindings, introduced above, can be viewed as a special kind of equations with the syntactic form

```
<variable name> := <expression>;
```

The difference is that a binding defines how a particular variable should be computed if all variables appearing in the right hand side expression are known. An ordinary equation defines no particular computational order and can therefore be solved for virtually any variable that appears in any of the expressions.

4.6 Class inheritance

Inheritance is a mechanism for propagating attributes of one class, called the super class, to the definition of a new class, called the derived class. Every class in Omola, except for the predefined class Class, is derived

from a super class. Because of this, all classes, predefined as well as user defined ones, form a tree structure where Class is the root.

Inheritance works as follows: all attributes of the super class are also attributes of the derived class, except for those overridden by local definitions in the derived class. Attributes that are defined in the body of a class definition are called *local* attributes with respect to the class. Attributes that are not local are called *inherited* attributes. Inheritance is transitive so that a class inherits all attributes defined in the hierarchy of super classes starting from the root of the class tree. A local attribute overrides an inherited attribute with the same name. This makes it possible for a class to redefine inherited attributes. Attributes without names, like equations, are also inherited but they cannot be redefined. Inheritance and the meaning of the super class concept can be defined by the following statement:

SEMANTIC RULE 4—Inheritance and attribute ordering

The attributes of a class is the list formed by the list of attributes of the super class, excluding those attributes of the super class having the same name as a local attribute, followed by the list of local attributes in order of definition. □

The rule states that a class is an (ordered) list of attributes. In most cases the order in which the attributes are defined is not significant but for record terminals, which are discussed below, order is important.

The meaning of a class is not dependent on whether a particular attribute is inherited or local. The inheritance concept is used to structure class definitions so that more specific classes can be derived from more general ones. A class representing some general concept can be reused as a super class and specialized in several different ways. A derived class can only redefine inherited attributes and add new ones — it cannot remove inherited attributes. As an example of inheritance, regard the following definitions:

```
A ISA Class WITH
  x TYPE Real;
  y TYPE Real;
END;
```

```
B ISA A WITH
  y TYPE Integer;
  z TYPE Real;
END;
```

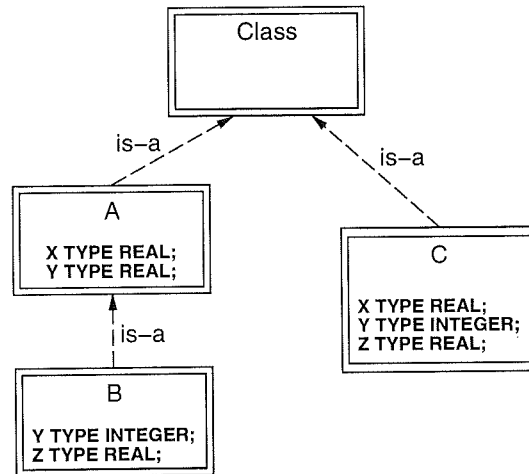


Figure 4.4 Diagram showing relationships between classes and attributes for the example given in the text.

```
C ISA Class WITH
  x TYPE Real;
  y TYPE Integer;
  z TYPE Real;
END;
```

Class C is equivalent to class B in terms of the attributes. The definition of B includes a local attribute, y, which overrides the attribute with the same name defined in the super class A. The classes are displayed graphically in Figure 4.4.

Omola only supports *single inheritance* which means that every class has at most one direct super class. *Multiple inheritance* means that a class may inherit from more than one direct super class (see for example [Rumbaugh *et al.*, 1991]). Single inheritance is conceptually more clear and easier to implement. This is the main reason why it is adopted in Omola. Multiple inheritance may cause conflicts between attributes inherited from different paths. Many modeling cases where multiple inheritance may seem appropriate, can be solved using single inheritance and composition.

Resolving the super class

Every class definition has a reference to a super class. The reference is usually a single name like 'Class' and 'A' in the examples above. In these

```

M1 ISA Class WITH
  X ISA Class WITH ... END;
  W ISA Class WITH ... END;
END;

M2 ISA Class WITH
  X ISA Class WITH ... END;
  M21 ISA M1 WITH
    X ISA Class WITH ... END;    % overrides inherited X
    Y ISA THIS::X WITH ... END; % specializes M2.M21.X
    Z ISA SUPER::X;              % specializes M1.X
    W ISA OUTER::X;              % specializes M2.X
  END;
END;

```

Listing 4.1 Example of usage of special super class qualifiers.

cases, the names refer to *global* class definitions, either predefined classes or defined in the same *library*. A *global* class is a class defined at the top level, i.e., not as a component of some other class. A class defined as a component of another class is called a *local class*. A *library* is a named group of class definitions. Each library makes a separate name space for global definitions. It is possible to refer to a super class defined in another library using qualified reference. For example, the following class has a super class defined in a library called `Controllers`:

```
Regulator ISA Controllers::PID;
```

Libraries are further discussed in Chapter 7.

Specializing local classes

It is possible to specialize a local class, that is, to use a local or inherited component as a super class of another component. To indicate local inheritance three special super class qualifiers are used: ‘THIS’, ‘SUPER’, and ‘OUTER’. Listing 4.1 illustrates the use of these qualifiers. All three qualifiers indicate that the super class is localized by a search in the local context instead of globally in libraries. The search works upwards in the inheritance hierarchy and outwards in the component hierarchy. The qualifiers differ only in the place where the search is started. They work as follows.

THIS The super class is first searched in the owner of the class being defined.

SUPER The super class is first searched in the owner's super class. This makes it possible to define class attributes like 'X ISA SUPER::X ...' where the inherited component X is redefined and specialized.

OUTER The super class is first searched in the owner of the owner of the class being defined. This qualifier is similar to THIS but the search starts one step further out in the composition hierarchy. It is an error to use this qualifier for components of global class definitions.

Even though local inheritance is not used for most simple model definitions, there are cases when it can be useful and there are good reasons for allowing it in Omola. Omola is heavily based on local class definitions. The reasons for this are discussed in the end of this chapter. If a local class definition should really work as a true class, rather than just an invocation of a global class as a component, it must be possible to specialize it. This motivates the introduction of the qualifier THIS. If one component of a model specializes a global class, this qualifier makes it possible for other components (in the same class body or in derived classes) to reuse the specialization. The definition of Y in Listing 4.1 is an example of this.

Another example of a natural use of local inheritance is when it is desirable to specialize an inherited component. This means to replace an inherited component with a specialized version of it. It is not possible to use THIS since "X ISA THIS::X ...;" would create a circular class definition (a class that is its own super class). Instead of "THIS::X" it should be "SUPER::X" which explicitly refers to the inherited X and the circularity is avoided.

The following is a motivation for the OUTER qualifier. Components can sometimes be used as model parameters. A model can be reused so that some of its components are redefined and replaced by definitions adapted to a particular usage. This kind of model parameterization is discussed in detail in [Nilsson, 1993]. Class M1 in the example in Listing 4.1 is a model that is reused as a component, called M21, in class M2. M21 redefines the inherited component called Y as a subclass of the component X defined in M2. Without the OUTER qualifier it would not have been possible to refer to M2.X as a super class in this case, since there is a another component called X defined locally in M21.

4.7 Variable scope and binding rules

Mathematical and symbolic expressions appear in equations and bindings. Expressions include symbolic references to class attributes and mathematical and logical operators. For an expression to be valid, each symbolic

4.7 Variable scope and binding rules

reference must uniquely determine a variable attribute. When a model is instantiated, every symbolic reference is resolved into the corresponding variable instance. For a class to be semantically correct, it is required that all its expressions are correctly resolvable.

Omola is a block oriented language in a similar way as Algol and Pascal. A block in Omola is a class definition. The *scope rules* define where in the block structure a particular name is a valid reference to an attribute. The scope rules of a block structured language [Aho and Ullman, 1977] can be rephrased into Omola terminology:

SEMANTIC RULES 5—Scope rules

1. *A named attribute declared within class C is directly accessible within C and in classes derived from C.*
2. *If class D is a local component of C, then any named attribute directly accessible in C is also directly accessible in the body of D, unless D has an attribute with the same name.*

□

The scope rules are adapted to take into account the inheritance. The phrase “directly accessible” means that the attribute can be referred to simply by its name. Some attributes that are not directly accessible can still be reached using name concatenations explained below. In the second scope rule, it can be observed that if the component class *D* has an inherited attribute with the same name as an attribute of the outer class *C*, it is the inherited attribute that will be directly accessible. In other words, an attribute inherited from the super class takes precedence over an attribute defined in a surrounding (owner) class.

Attributes of components can be accessed using name concatenation with a period between each name. For example, if *A* in some context refers to a component which, in its turn, has a component called *B* which has a variable called *X*, then that variable can be referred to as ‘*A.B.X*’. This is illustrated by the example in Listing 4.2 where the binding in class *M2* refers to the variable defined in class *M0*.

Value semantics

References in a mathematical or logical expression is assumed to refer to variable attributes. It is sometimes useful (for reasons that are explained in Section 4.8) to use classes for representing variables. For example, assume there are the following definitions:

```
State ISA Class WITH value TYPE Real; END;  
X ISA State;
```

Chapter 4. The Modeling Language Omola

```
M0 ISA Class WITH
  X TYPE Real := 0.0;
END;

M1 ISA Class WITH
  B ISA M0 WITH X := 1.0; END;
END;

M2 ISA Class WITH
  A ISA M1;
  A.B.X := 2.0;
END;
```

Listing 4.2 Example illustrating reference notation and variable bindings.

Normally, in an expression referring to the value of X , the reference is written as “ $X.value$ ”. However, it is also possible to refer to the value of X using only the symbol “ X ”. The concept is called *value semantics* and it is introduced to simplify the notation. Value semantics applies to all classes having a variable attribute called *value*. The concept is summarized by the following rule.

SEMANTIC RULE 6—Value semantics

If r is a valid reference to a class attribute in a context where a variable is expected and if $r.value$ is a valid reference to a variable attribute, then r is also a valid reference to that variable attribute. □

Variable bindings

A *variable binding* is an assignment attribute associating a variable with an expression. Bindings are inherited and can be overridden in subclasses as well as in owner classes. Listing 4.2 shows an example of this. The variable is first given an original binding to zero when it is defined in class $M0$. Class B , which is a component of $M1$, inherits the variable but overrides the binding. Hence, in $M1$, the value of $B.X$ is one. Component A in $M2$ inherits from $M1$. The binding attribute in $M2$ again overrides the value of X in the component of the component.

The following rules summarize the use of variable bindings:

SEMANTIC RULES 7—Variable bindings

1. A variable has at most one valid binding.
2. A local binding overrides an inherited one.
3. A local redefinition of a variable inhibits an inherited binding to that variable.

4.7 Variable scope and binding rules

4. *If r is a valid reference to a component c , and if $r.x$ is a valid reference to a variable attribute of c , then a binding to $r.x$ overrides a local or inherited binding to x in c .*
5. *It is an error if there is more than one local binding to the same variable.*
6. *It is not allowed to bind a submatrix or a single element in a matrix or a vector variable.*

□

Rule 3 is illustrated by the following example where the variable X is made unbound (free) by the redefinition of the variable in class B .

```
A ISA Class WITH
  X TYPE Real := 1.0;
END;

B ISA A WITH
  X TYPE Real;
END;
```

The difference between bindings and equations

A variable that has a binding to a constant expression is called a constant. A constant expression is an expression that contains no non-constant variables. The value of a constant can be determined from the class definition alone by evaluating a number of binding expressions in a recursive order. The difference between the binding “ $x := 1$ ” and the equation “ $x = 1$ ” is that the binding defines x to be a known constant while this is not the case for the equation. The equation is not used for deriving a value for x in the class definition. In the general case, free variables and equations make up a non-linear equation system, which cannot be solved by a simple recursive algorithm. However, when a class definition is instantiated into a simulation model, general non-linear equation systems are analyzed and the constant values are inferred from general equations. This is discussed in Chapter 8. The reason why trivial equations like “ $x = 1$ ” are not used at an early stage to derive constant variables is to keep the difference between variable bindings and equations conceptually clear. Variables with constant bindings are used as model parameters. This is discussed in detail in Section 4.12.

A binding where the expression cannot be evaluated to a known constant is similar to an ordinary equation. The difference is that the computational causality is determined by the binding but not by the equation. For example, compare the equation and the binding:

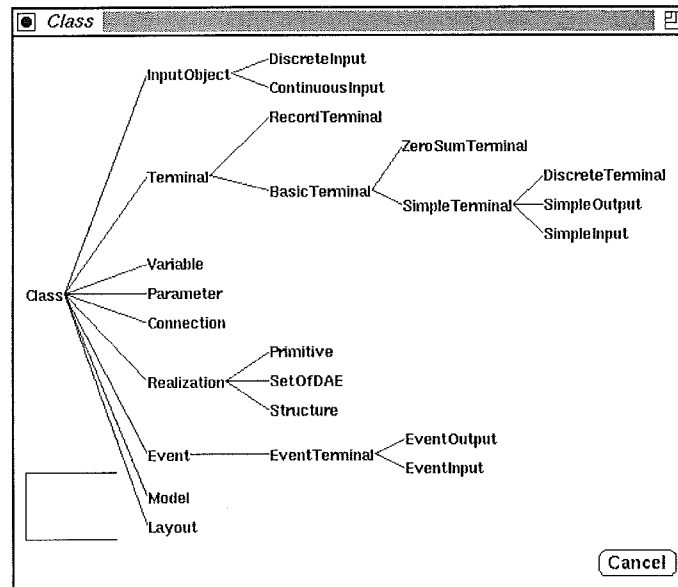


Figure 4.5 The inheritance tree of all predefined Omola classes.

$x = y+z;$
 $x := y+z;$

The equation may be used to determine either one of the tree variables, while the binding must only be used to compute x when y and z are known. Normally a structured modeling methodology requires acausal models which means that behavior is represented by ordinary equations. However, assignments (bindings) are motivated when the computational causality is clear from the real system, for example, the output signal from a controller. If the causality defined by a binding is not consistent with the other equations of the model, this is detected and reported as an error when the model is instantiated and analyzed. This is discussed in detail in Chapter 8.

4.8 Model representation in Omola

Models are represented in Omola based on a set of predefined classes with predefined interpretations as model components. These classes are displayed as an inheritance tree in Figure 4.5. The complete definitions of the built-in Omola classes are found in Appendix B. In this section we will

in some cases give simplified definitions where details, irrelevant for the discussion, are left out. In most cases the omitted parts of the definitions concern the graphical representation of models.

Models

The predefined class `Model` is the root class of all user defined models. A model is basically a class that encapsulates behavior. For example, it can describe a complete autonomous dynamic system or it can describe a simple static relation between two interface variables. Any class that descends from `Model` is considered a model. It is possible to distinguish between three basic types of models depending on for what purpose they are defined:

1. A complete model that can be instantiated into a well-defined simulation problem.
2. A component model that can be used as a submodel in an aggregate model. It is characterized by having an interface of terminals that can be connected to other submodels. If it is instantiated solely, it normally leads to an underdetermined simulation problem.
3. An abstract model that can only be used as a super class in other model definitions. It generalizes properties common to a set of models.

The definition of the abstract model `Model` can, for the discussion in this chapter, be viewed as the trivial specialization of `Class`:

```
Model ISA Class;
```

The true definition of `Model`, found in Appendix B, contains an attribute with graphical information determining the model's graphical presentation.

4.9 Terminals

A terminal is an object or a class descending from the predefined Omola class `Terminal`. Terminals are used for defining the interface of a model, especially for defining how a component model interacts with other component models by means of connections. The class `Terminal` has number of subclasses, shown in Figure 4.5, that add attributes and represent different kinds of terminals.

Basic terminals

The class `BasicTerminal` is a specialization of `Terminal`. A basic terminal represents a single interface variable. Its definition is given in Listing 4.3.

Chapter 4. The Modeling Language Omola

```
BasicTerminal ISA Terminal WITH
  value      TYPE Real;
  quantity   TYPE STATIC String := "number";
  unit       TYPE STATIC String := "1";
  variability TYPE STATIC (TimeVarying, Parameter) :=
    'TimeVarying;
  default    TYPE STATIC Real;
END;

SimpleTerminal ISA BasicTerminal WITH
  causality  TYPE STATIC (Undefined, Input, Output) :=
    'Undefined;
END;

ZeroSumTerminal ISA BasicTerminal WITH
  direction  TYPE STATIC (In, Out) := 'In;
END;
```

Listing 4.3 The definition of BasicTerminal and the specializations SimpleTerminal and ZeroSumTerminal

The most important attribute in BasicTerminal is value which defines the actual interface variable. The attributes quantity and unit refer to a database of quantities and units of measure. They can be used for checking consistency of connections and they are discussed in more detail in the following section. The attribute variability is normally bound to the symbolic value TimeVarying indicating that the terminal represents a time varying variable. It can be rebound to Parameter to indicate that the terminal represents a model parameter, i.e., a time invariant value specified by the user. Parameters are discussed in detail in Section 4.12 and in Section 4.14. The attribute default may be bound to an expression which will be used as an equation defining the value of the terminal when it is not connected. This is also discussed in detail in the following section.

BasicTerminal is further specialized into SimpleTerminal and ZeroSumTerminal, also defined in Listing 4.3. A simple terminal results in an ordinary symmetric equation when it is connected to another simple terminal. A set of two or many connected zero-sum terminals result in a single equation where the sum of the terminal variables equates to zero. In physical modeling, simple terminals are used for representing *across variables* while zero-sum variables represent *through variables*. Across and through variables are for example discussed in [Koenig *et al.*, 1967].

In bond graph modeling the corresponding terms *effort variables* and *flow variables* are used [Karnop and Rosenberg, 1975]. Typical examples of across variables are variables representing voltage, pressure, and position. Examples of through variables are electric current, flow of matter or energy, force, and torque.

`SimpleTerminal` defines an attribute called *causality* which is bound to `Undefined` but can be rebound to either `Input` or `Output`. This attribute makes it possible to define the computational causality for the terminal. A causality of `Input` means that the value of the terminal is computed by the environment of the model owning the terminal while a causality of `Output` means that the value is computed by the model owning the terminal. Zero-Sum terminals cannot be given a defined causality. On the other hand, they have an attribute called *direction*, which can be given the symbolic values `In` or `Out`. This attribute determines if the variable should appear in the resulting zero-sum equation with a positive or negative sign. When a zero-sum variable is representing, for example, a flow, its direction should be interpreted as the direction of positive flow: into the model or out from the model which owns the terminal. The following section defines exactly how the terminal attributes affect the consistency and meaning of connections.

The class `SimpleTerminal` is further specialized in three additional classes: `SimpleInput`, `SimpleOutput` and `DiscreteTerminal`. The former two classes simply rebound the causality attribute of `SimpleTerminal` to `Input` and `Output`. The class `DiscreteTerminal` represents a discrete interaction variable, i.e., a variable that only changes its value as a result of a discrete event. This is discussed more in Chapter 6.

Structured terminals

The class `RecordTerminal` is used as the super class of terminals that consists of several quantities, i.e., terminals representing an aggregate of interaction variables. It is defined as an empty specialization of `Terminal`. A record terminal is supposed to have component attributes which are terminals as well. It is used for grouping a set of terminals which naturally belong together. This is useful for terminals representing physical interaction like pipe connections containing components of flow, pressure, and temperature. An example of a record terminal is the *cut* defined in Figure 3.7.

4.10 Connections

A *connection* is a basic concept with a special syntax in Omola. It is a

Chapter 4. The Modeling Language Omola

class attribute that defines a symmetric relation between two terminals. It is written as:

`<terminal> AT <terminal> ;`

Each `<terminal>` is a symbolic reference to a component which must be a descendant of the classes `Terminal` or `EventTerminal` (event terminals are discussed in Chapter 6). Since the connection is symmetric the order of the terminals is not significant.

A connection is semantically correct only if the involved terminals have identical structures and if the basic terminal components are connectable. Basic terminals are connectable if their quantity and causality attributes are consistent. When two record terminals are connected, it means the same as if each terminal component is pair-wise connected. Because of this, the order in which the components appear in a record terminal definition is important. This is the only case in Omola where the ordering of class attributes is significant. The order of the attributes in a class is defined by the inheritance rule: Semantic Rule 4.

The rules of connection consistency are summarized by the following statements:

SEMANTIC RULES 8—Connection consistency

1. *Structure: A basic terminal (representing a single quantity) must be connected to another basic terminal. Record terminals must have the same number of components and the components must be pair-wise connectable.*
2. *Quantity: Basic terminals must have consistent quantities. Quantities are consistent if they are known by the database and are equal.*
3. *Type: Basic terminals must be data type consistent according to the same rules that apply for equations.*
4. *Value: If connected terminals have constant values they must be equal.*
5. *Causality:*
 - (a) *Two simple terminals (across terminals) with defined causality (input or output) must be consistent.*
 - (b) *Two zero-sum terminals (through terminals) have no defined causality and are always correct.*
 - (c) *If a simple terminal is connected to a zero-sum terminal the former must have a defined causality (input or output).*

□

Rules 5a and 5c are defined more precisely in the following. A motivation for connection rule 5c is that the only sensible interpretation of a connection between a simple terminal and a zero-sum terminal is to regard it

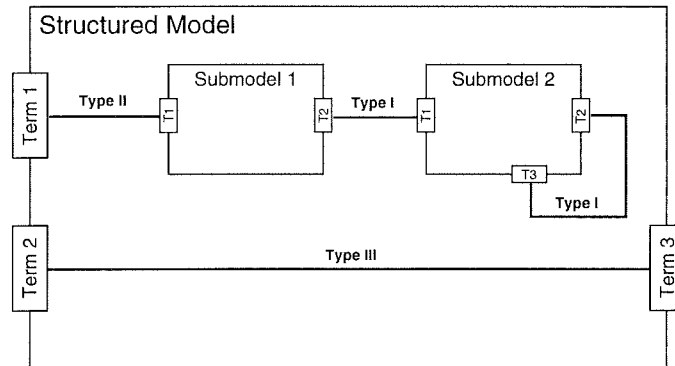


Figure 4.6 A connection diagram showing the three possible connection schemes.

either as measurement or as an “injection” of a value at the connected zero-sum terminal.

Equations deduced from connections

Connections are equivalent to equations or assignments between terminal variables. A connection between two basic terminals corresponds to a single equation. Actual equations are generated from connections during model instantiation. The resulting equation depends on the classes of the terminals and their attributes. A terminal derived from `SimpleTerminal` has the attribute `causality` affecting the type of equation, while a `ZeroSumTerminal` has the attribute `direction` affecting the generated equation. Since causality can take three different values and direction can take two different values, terminals can be connected in fifteen different combinations. Not all combinations are valid connections. The equation generated from a connection depends also on the connection scheme. There are three kinds of connection schemes, illustrated in Figure 4.6: (I) a connection between terminals belonging to one or two submodels, (II) a connection between a submodel terminal and a terminal of the model owning the connection, and (III) a connection between two terminals, both belonging to the model owning the connection.

Table 2 shows the types of generated equations from different combinations of terminals and connection schemes. The table lists all possible combinations of simple terminals with different causalities and zero-sum terminals with different directions. A dash instead of an equation means that the combination is illegal and is violating either rule 5a or 5c in Semantic Rules 8.

Table 2 Equations generated from connection 'A AT B' with different combinations of terminals A and B and with different connection schemes. In the two left hand columns a SimpleTerminal with different values of the causality is indicated by 'S/...' while a ZeroSumTerminal with different values of the direction is indicated by 'Z/...'. For type II connections, note that terminal A belongs to the outer model and the reverse order is not included in the table.

Terminal class and attributes		Equations resulting from different connection schemes		
A	B	I	II	III
S/Undefined	S/Undefined	$A = B$	$A = B$	$A = B$
	S/Input	$B := A$	$B := A$	$A := B$
	S/Output	$A := B$	$A := B$	$B := A$
	Z/In	—	—	—
	Z/Out	—	—	—
S/Input	S/Input	—	$B := A$	—
	S/Output	$A := B$	—	$B := A$
	Z/In	$A := B$	$B := A$	$B := A$
	Z/Out	$A := B$	$B := A$	$B := A$
S/Output	S/Output	—	$A := B$	—
	Z/In	$B := A$	$A := B$	$A := B$
	Z/Out	$B := A$	$A := B$	$A := B$
Z/In	Z/In	$A+B = 0$	$A-B = 0$	$A+B = 0$
	Z/Out	$A-B = 0$	$A+B = 0$	$B-A = 0$
Z/Out	Z/Out	$A+B = 0$	$B-A = 0$	$A+B = 0$

Multiple connections to a single terminal

A terminal can be involved in several connections in the same model. As an example, regard the situation in Figure 4.7 which shows a model with one terminal, two submodels, and two connections: 'T1 AT S1.T' and 'T1 AT S2.T'. If the involved terminals are all derived from SimpleTerminal, the configuration represents no special case and two equations are deduced according to the rules above. However, if the terminals are derived from ZeroSumTerminal, they represent flows and the semantics of the connections is different. In this case the two connections together represent a single node in a circuit graph, see Figure 4.7, which corresponds to a single equation according to Kirchhoff's law: $T_1 + T_2 + T_3 = 0$.

In order to define the general rule for deducing equations from sets of connected zero-sum terminals some definitions are introduced. First assume that the connections between the zero-sum terminals of a model are represented as a graph where terminals are vertices and connections are edges. In general, the graph is not connected but consists of a set of

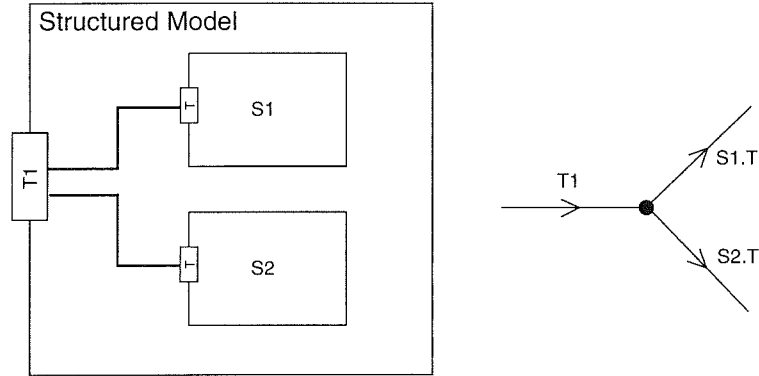


Figure 4.7 The left diagram shows a connection diagram with two connections to the same terminal. The right diagram shows a corresponding node in a circuit diagram resulting when the involved terminals are zero-sum terminals.

component graphs which consists of the largest connected subgraphs.

DEFINITION 4.1

A *connection set* of model M is a set of terminals belonging to the same component graph. \square

For example, assume the following connections are defined in the same model:

T1 AT T2;
T2 AT T3;
T4 AT T5;

The model has two connection sets: $\{T_1, T_2, T_3\}$ and $\{T_4, T_5\}$.

DEFINITION 4.2

A terminal is an *outer terminal* with respect to a model M if it is defined as a component of M . It is an *inner terminal* with respect to M if it belongs to a submodel of M . \square

For example, in Figure 4.7, T1 is an outer terminal while S1.T and S2.T are inner terminals of the structured model.

One equation is deduced from each connection set of a model. This is done according to the following rule which generalizes Table 2 for connection sets with more than two zero-sum terminals.

Chapter 4. The Modeling Language Omola

SEMANTIC RULE 9—Zero-sum connections

Each connection set T of a model generates the equation:

$$\sum_{T_i \in T} d_i s_i v_i = 0 \quad (4.1)$$

where

$$d_i = \begin{cases} 1 & \text{if the direction of } T_i \text{ is In} \\ -1 & \text{if the direction is } T_i \text{ is Out} \end{cases}$$
$$s_i = \begin{cases} 1 & \text{if } T_i \text{ is an inner terminal} \\ -1 & \text{if } T_i \text{ is an outer terminal} \end{cases}$$

and v_i is the value variable of T_i . □

Units and quantities

Basic terminals have attributes for specifying quantity and unit of measure. The quantity attribute is used for checking consistency of connections according to Semantic Rules 8. A connection between two terminals is only valid if they have the same quantity. The concept is useful for preventing accidental connection errors when models are representing physical systems.

The quantity attribute defined in `BasicTerminal` must be bound to a string which must be a valid name of a quantity according to the standard international standard ISO31. The unit attribute, also defined in `BasicTerminal`, must be bound to a string which is a valid unit of measure specification according to the built-in quantity and unit data base [Andersson, 1993]. The unit of measure specification for a terminal may involve multiplicity prefixes which will be used as conversion factors in the generated equations. For example, if terminal `T1` has unit bound to "m" (meters) and terminal `T2` has unit bound to "mm" (millimeters), then a connection between the terminals, assuming they are simple terminals, will generate an equation equivalent to

$$T_1 = 10^{-3} \times T_2.$$

If the terminals are derived from `ZeroSumTerminal`, the equation generated according to (4.1) is modified in a similar way.

4.11 Variable components

The predefined class `Variable` should be used for representing model variables when it is desirable to define an initial value in the model.

Initial values apply to state variables and are used by the simulator. The definition of `Variable` looks like:

```
Variable ISA Class WITH
  value TYPE Real;
  initial TYPE Real;
END;
```

Except for the possibility to bind the `initial` attribute, a variable component, for example defined by “`x ISA Variable`”, is equivalent to an ordinary variable defined by “`x TYPE Real`”. Because of value semantics, the variables can be referred to by the name `x` in an expression, independently of the type of definition used.

An advantage of using a class to represent a variable is that the class can be specialized and extended with more attributes. For example, it may be useful to associate minimum and maximum values with a model variable.

4.12 Parameters

The predefined class `Parameter` should be used as a super class for all user parameters. A *free parameter* is a variable that is regarded as a known constant as far as the dynamic model behavior is concerned, but where the value can be changed, for example between different simulation experiments, in the instantiated model by the user. In other words, a free parameter typically affects the model behavior but changes value only due to events external to the instantiated model. The definition of `Parameter` is:

```
Parameter ISA Class WITH
  value TYPE Real;
  default TYPE Real := 0.0;
END;
```

The parameter has two variable attributes: `value` and `default`. The latter is used as the value of the parameter in case no explicit value is given.

A parameter which has a binding is not a free parameter and cannot be changed directly in the instantiated model. A parameter with a binding is either a constant or its value depends on other free parameters. It is an error to bind a parameter to an expression that depends on time varying variables.

A *parameter equation* is an equation that only include parameters. Parameter bindings and parameter equations can be defined explicitly

or by connections between terminals. They can be used for propagating parameter values between different parts of a model. Parameter propagation occurs in the instantiated model and it is performed by the simulator. This is discussed in detail in Chapter 7. It should be noted that most of the semantics concerning free parameters and parameter bindings and equations is defined by the tools superior to the model definition level of Omola. At the model definition level, the meaning of a parameter can be summarized by the following definition and semantic rule.

DEFINITION 4.3

A *parameter variable* is the value attribute of a component derived from the class `Parameter`.

SEMANTIC RULE 10—Parameter bindings

A *binding to a parameter variable may only depend on other parameter variables or constants.* □

4.13 Model instantiation

Model instantiation is the procedure of transforming a model, represented as Omola classes, into a data structure suitable for further analysis and simulation. The resulting data structure is called a model instance. The instantiation procedure is discussed in detail in Chapter 7. This section is intended to clarify the difference between a model represented as a class structure in Omola and a model instance.

The main difference between an instance and its class is that the instance contains separate objects for each single component, variable, and equation. The class representation of a model can be viewed as a description (a recipe or a blue print) on how to construct the actual model, i.e., the model instance. The class representation is economic in the sense that a component that appears in several similar copies may be represented by a single class. In the model instance however, each component must have its own separate representation. For that reason, the model instance is a structure with a one-to-one mapping from model component objects to objects in the modeled reality.

The following example may help to clarify the differences between the different model representations used in an Omola based modeling environment. As shown in Figure 4.2 there are three main representations to consider: the Omola text representation, the class representation, and the instance representation. As an example, regard the following Omola definitions.

A ISA Model WITH

4.14 Model parameterization

```
C1 ISA B WITH y TYPE Real; END;  
C2 ISA B WITH z TYPE Real; END;  
END;  
  
B ISA Model WITH  
  x TYPE Real;  
END;
```

The definitions are derived from the predefined class Model. Figure 4.8 shows an object diagram with the corresponding class representation of the textual definitions and one instantiation of class A. The class representation is created by parsing the textual representation while the instance representation is constructed by applying the instantiation procedure once to class A. Variables are actually represented as objects as well, but here they are only indicated in the diagram by box annotations in order not to complicate the picture too much. Relations between objects are represented by arrows. The inheritance (is-a) relations and the component (has) relations are indicated by dashed annotated arrows. The instance – class relation is indicated by solid arrows.

An important observation from Figure 4.8 is that the inheritance structure is removed while the component structure is preserved in the model instance. These are design decisions based on the following arguments. Inheritance is a way of structuring model descriptions in order to facilitate model reuse. Inheritance is an artifact added to the model representation and it has usually no correspondence in the reality represented by the model. The model instance is a direct representation of the modeled reality and there is no reason to maintain the inheritance structure. Viewed simply as a representation of a mathematical problem to be solved by the simulator, there is no reason to maintain the component structure in the model instance. However, simulation is an interactive activity and the user needs to get access to variables and parameters. This is most easily done if the component structure, as defined in the class representation, is kept intact. Since the component structure of a model often reflects the structural properties of the real system there is also a reason to maintain the component structure.

4.14 Model parameterization

Model parameters are variables introduced in the model to make it more reusable and adaptable for different purposes. There are two basic kinds of parameters appearing in Omola models. These are

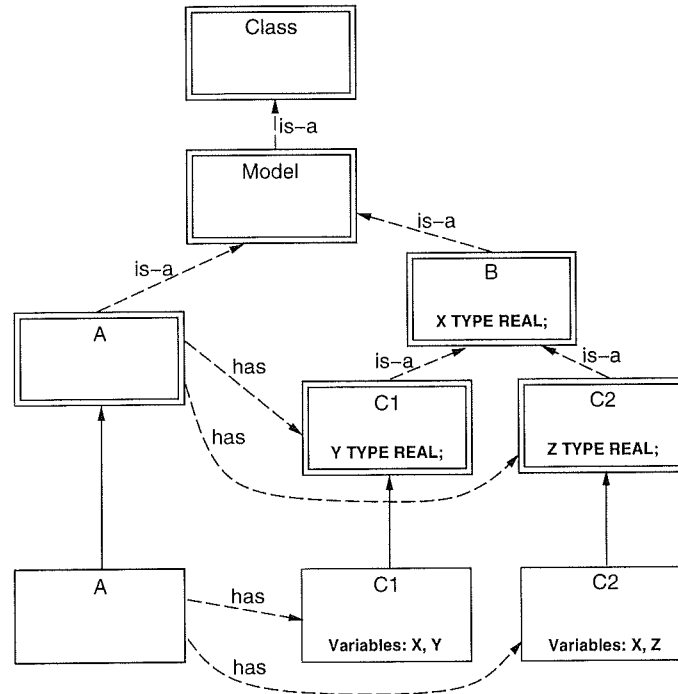


Figure 4.8 An object diagram showing classes and instances of a simple example given in the text. Class objects are displayed with double framed boxes while instance objects are displayed with single framed boxes. Dashed arrows are indicating inheritance and component relations. Solid arrows are indicating instance – class relations.

- instance parameters or just ordinary parameters and
- class parameters or structure parameters.

Ordinary parameters

Ordinary parameters are used for *time constant* physical attributes that the user may want to change in the instantiated model. For example, a model of a control system with a process and a PID controller may have the controller attributes like *gain* and *integral time* defined as parameters. Important properties of the process may also be defined as parameters. These kind of parameters are not required to have known values in the model definition. They can be changed in the instantiated model instead, to specify different simulation experiments.

The predefined class *Parameter*, presented above, must be used to de-

```

TrayModel ISA Model WITH
  TrayArea ISA Parameter;
...
END;

ColumnModel ISA Model WITH
submodels:
  Tray1, Tray2, ..., Tray10 ISA TrayModel;
parameters:
  TrayArea ISA Parameter;
parameterEquations:
  Tray1.TrayArea := TrayArea;
  Tray2.TrayArea := TrayArea;
...
  Tray10.TrayArea := TrayArea;
END;

```

Listing 4.4 A simplified model definitions to illustrate the concept of parameter propagation.

fine ordinary parameters. An attribute of this class tells that the variable (the value attribute) is not time varying and it must get a user defined value in the instantiated model. The parameter class has an attribute called *default* which may be defined in the model and used as a default value for the parameter.

An ordinary parameter with an unbound value is called a *free parameter*. It is common that a library model with many free parameters is specialized when it is used as a component, and some of the free parameters are bound to specific values. A bound parameter can no longer be changed directly in the instantiated model. In fact, if it is bound to a constant, it cannot be changed at all. A common situation when models are reused as components is that parameters are bound to expressions depending on other parameters. In this way the value of single parameter of a top level model can propagate to parameters of several submodels. Parameter propagation is defined by ordinary binding attributes.

Listing 4.4 shows an example, adopted from [Nilsson, 1993], of a distillation column model with a set of equally sized tray models as components. The TrayArea parameter of the column model is propagated to all its tray components. The parameters of the tray components can no longer be changed directly. Parameter propagation helps to create abstract interfaces for structured models and relieves the user from the trouble of setting many individual parameters.

Chapter 4. The Modeling Language Omola

```
Linear ISA Model WITH
  N TYPE Integer;
  X TYPE Column [N];
  A TYPE Matrix [N, N] := -eye(N);
  X' = A*X;
END;
```

```
Linear2 ISA Linear WITH
  N := 2;
  A := [-1.0, 0.5; 0.3; -0.8];
END;
```

Listing 4.5 Example of a general linear model of parameterized size and a specialization into a second-order model.

Structure parameters

Structure parameters are only affecting a model at the class level. Most of the attributes of the predefined class `BasicTerminal` (refer to Listing 4.3) are class parameters. For example, `quantity` is used for checking connection consistency and *causality* affects the way a connection is translated into an equation. Class parameters concern the way a model definition is interpreted but are not related to actual behavior of the represented system. In this sense, class parameters are a kind of *meta model* parameters.

Another way of using class parameters are as array sizes. Constants and constant expressions may be used in array size declarations. It is possible to define classes with undefined array sizes. Such a class cannot be instantiated alone but it can be used as a super class or a component in another class which adds a binding to the matrix size parameter. This is a common way of defining reusable models. An example is given in Listing 4.5. The integer variable `N` determines the size of the matrices. The class `Linear` is specialized into `Linear2` which binds the value of `N` to 2. Since `N` affects the size of the model it is called a *structure parameter*. A class cannot be instantiated into a valid model unless all structure parameters can be determined as known constants.

4.15 Discussion

Omola is totally based on class definitions. Most object-oriented programming languages make a distinction between a class definition and the invocation of a class as a component of another class. This distinction is not made in Omola where every local class definition is also an implicit

invocation of that class as a component. The reasons for this are that the notation becomes simpler in most practical cases and that fewer names must be invented by the modeler. The following example illustrates the idea.

Assume a process is going to be represented as an Omola model. The process contains, among other components, two storage tanks for liquids. An appropriate model for the tanks is available from a library of process model components. The tank model has a parameter, called *capacity*, defining the maximum storage capacity of the tank. The tanks of the modeled process have maximum capacities of 8 m² and 10 m² respectively. Since the tanks have fixed sizes it is natural to specialize the general tank model and to bind the capacity parameters. This is done by the following definitions:

```
SmallTank ISA ProcessLibrary::Tank WITH
  capacity := 8.0;
END;
```

```
LargeTank ISA ProcessLibrary::Tank WITH
  capacity := 10.0;
END;
```

The two definitions are clearly specializations of the general tank model since a free parameter is bound to constant values. However, the specializations are defined to be used as components in the process model only, and therefore, it is natural to define them as local classes of the process model. In Omola, this looks like:

```
Process ISA Model WITH
  SmallTank ISA ProcessLibrary::Tank WITH
    capacity := 8.0;
  END;
  LargeTank ISA ProcessLibrary::Tank WITH
    capacity := 10.0;
  END;
  ...
END;
```

The local class definitions mean also that the process have two components, called *SmallTank* and *LargeTank*. This can be referred to as *implicit invocation* of local class definitions. If Omola did not have implicit invocation the process model would have to be defined as:

```
Process ISA Model WITH
  SmallTank ISA ProcessLibrary::Tank WITH
```

Chapter 4. The Modeling Language Omola

```
        capacity := 8.0;
    END;
    LargeTank ISA ProcessLibrary::Tank WITH
        capacity := 10.0;
    END;
    Tank1 TYPE SmallTank; % not correct Omola
    Tank2 TYPE LargeTank; % not correct Omola
    ...
END;
```

A drawback with implicit invocation is that it is sometimes desirable to define local classes that are not instantiated as model components but only used as super classes. Examples where this is a powerful way of parameterizing model are discussed in [Nilsson, 1993]. A class definition that is not indented for direct instantiation is called an *abstract class*. Nilsson suggests that Omola is extended with the possibility to define abstract local classes which are not automatically invoked as components. This is probably a good idea and requires only a minor extension of Omola and the associated manipulation tools. One possibility is to introduce the optional keyword `ABSTRACT` in a class definition. For example:

```
A ISAN ABSTRACT Class WITH ... END;
```

would mean that A is available for specialization but it is not automatically invoked as a component if it appears in a class body.

4.16 Summary

The modeling language Omola is designed to be a general language for representation of structured dynamic models. Omola consists of two layers: a basic, general data modeling layer and a model representation layer. The basic layer contains the following general concepts:

- Classes with single inheritance are used for data aggregation.
- Classes have attributes that are
 - variables,
 - local class definitions,
 - equations and bindings, and
 - connections.
- Named inherited attributes and bindings can be overridden.
- Variables are of different types including scalar real, array of reals, integer, string, and symbol.

4.16 *Summary*

Model representation is based on concepts at the second layer of Omola. This layer includes a set of class definitions used as base classes for modeling objects such as model, terminal, and parameter. There are different kinds of terminals with predefined attributes. Terminal attributes are used for checking connection consistency and they affect the way connections are translated into model equations. Terminals may be structured to represent multivariable interaction between model components.

Omola provide for abstract model interfaces but has no concept of information hiding. Every attribute of a class is accessible from the outside. Every named attribute of a class can also be redefined in a subclass.

5

Modeling of Discrete Event and Hybrid Systems

Not all dynamic phenomena can be represented by continuous equations. In particular man-made, non-physical systems involving information processing are often modeled as *Discrete Event Dynamic Systems* (DEDS). A difficulty with DEDS is that there does not exist any laws of conservation similar to physical systems. Models cannot be derived from first principles based on conservation of energy, matter, or momentum. The state space of a *Continuous Variable Dynamic Systems* (CVDS) is a dense subset of R^n while the state space of a DEDS can be any set, represented by variables of different types. This makes DEDS systems hard to analyze compared to CVDS systems. Hybrid systems, which contain continuous variable parts as well as discrete events, are even harder to analyze. Simulation is often the only way to investigate the behavior of such systems. This makes a simulator an extremely important engineering tool for DEDS and hybrid systems.

The aim of this chapter is to develop a general mathematical and logical framework for representing hybrid systems. This framework should serve as the basis for extending Omola with discrete event concepts.

5.1 Introduction

A computer controlled system is an example of a *hybrid system*. A general hybrid system consists of continuous processes interacting with a sequential automata in a network [Nerode and Kohn, 1993]. A dualistic model of a hybrid control system is shown in Figure 5.1. The physical plant can be viewed as a set of physical quantities, such as masses, energies, positions, velocities, etc., that change continuously in time. The behavior of

the plant, i.e. the evolution of the quantities, is governed by the general laws of physics. The control system is often a real-time, reactive computer system. It responds to discrete events from the environment by issuing other discrete events which affect the environment. A discrete event occurs at a certain point in time, it belongs to a predefined set of possible event types, and it may be associated with other data. The physical plant and the computer system interact through two kinds of transducers: sensors which measure quantities of the plant and generate discrete events, and actuators which generate physical commands from discrete events.

The problem of automatic control design consists of designing a control system so that the behavior of the overall system stays within certain given boundaries. Since general hybrid systems are hard to analyze formally and few theoretical results exist, control theory is mainly concerned with more specialized system models. An example is “classical” computer control [Åström and Wittenmark, 1990], which models the plant, including the transducers, as a sampled data model represented by difference equations. The controller and the plant are interacting by discrete signals, issued at regular sampling time points. This model is discussed in some more detail below. Another type of model abstraction is to view the process as an asynchronous, discrete event dynamic system (DEDS). A controller for a DEDS is called a supervisor. The supervisor and the plant are interacting by means of discrete events occurring with irregular time intervals. In some cases the supervisor can affect the plant by enabling or disabling certain controllable events in the plant. Control of DEDS has attracted much interest in recent years. A control theory with formal analysis and synthesis methods for DEDS is emerging [Ramadge and Wonham, 1989, Balemi, 1992]. DEDS and supervisors are often represented as finite automata with various extensions involving the notion of time. Finite automata models are discussed in more detail below. Also more general hybrid system models are in the focus of theoretical research, see for example [Nerode and Kohn, 1993].

A simulation environment for discrete event and hybrid models is a

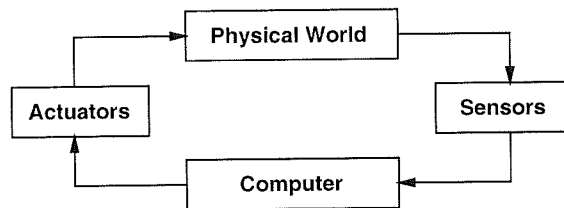


Figure 5.1 A view of a general hybrid system

very important tool for control systems engineering. Control systems are often designed using idealized models and formal methods. The designs are then verified by simulation with high-fidelity models. This requires a model representation which is able to represent models with different levels of idealization and accuracy.

There are several approaches to structure computer controllers. Some controller structures are suitable for formal design methods while others are useful for more intuitive design and tuning. Controllers are of very different complexity, ranging from single loop PID controllers to plant-wide control systems with layers of sequential and supervisory control. A lot of interest has recently been attracted to fuzzy controllers [Driankov *et al.*, 1993], which can be seen as an attempt to simplify the design process as well as the implementation of controllers with comparably little *a priori* knowledge about the plant. The design process of a fuzzy controller involves much trial-and-error, which makes simulations even more important. Adaptive and self tuning controllers often include complex logics for supervising the behavior of the controller and of the overall system. Knowledge based controllers [Årzén, 1987, Årzén, 1989] often include aspects of fault detection, fault diagnosis, on-line optimization, and strategic planning. Simulation of these types of control schemas within a coherent environment requires a powerful and general representation of hybrid models.

The dualistic view of a hybrid system, shown in Figure 5.1, is a simplified structure. Very often in process design, the process as well as the control system are modeled in a hierarchical fashion, and the control system is often distributed and local to the particular process units [Nilsson, 1993]. In this case, components at every hierarchical level consist of discrete events and continuous elements. Flexible structuring mechanisms are required in the modeling framework used for representing these kind of systems.

There exists a number of different formalisms for representing DEDS and hybrid systems and many of the formalisms appear in several different versions, generalizations, and specializations. A few of the formalisms often used in modeling and control of DEDS and hybrid systems, are presented in this chapter. The purpose is to give a brief overview of what kind of models are used in some of the theoretical work being done in the area of discrete event and hybrid systems. Some of the formalisms have influenced the design of the discrete event concepts in Omola. Among the formalisms discussed are sampled data models and Grafcet models often used in control engineering as way of specifying sequential control systems. The chapter ends by defining a hybrid model formalism developed for serving as mathematical and logical framework of Omola. The prob-

lems associated with simulation of the hybrid model are also discussed briefly.

5.2 The sampled data model

A simple type of discrete model is the sampled data model used extensively in automatic control [Åström and Wittenmark, 1990], system identification and signal processing. A model can be given on state-space form as

$$\begin{aligned}x_{k+1} &= f(x_k, u_k, t_k) \\ y_k &= g(x_k, u_k, t_k),\end{aligned}\tag{5.1}$$

where $x_k \in R^n, k = 0, 1, \dots$, is a sequence of state vectors, u_k is a sequence of input vectors which may be random variables, and y_k is a sequence of output vectors. In a real-time environment, periodic sampling is assumed and x_k, u_k , and y_k represent the values at times $t_k = kh$, where h is the constant sample interval. If the functions f and g are linear and time invariant, the analytical properties of this representation are vast compared to the other formalisms discussed in this chapter.

5.3 Finite-State Automata

The finite automaton is a simple and often used formalism for representing discrete behavior [Heymann, 1990]. It is often used for modeling and specification of digital sequential circuits, manufacturing systems, and supervisory control algorithms.

A finite state automaton, FSA [Hopcroft and Ullman, 1979, Balemi, 1992], is the tuple

$$M = \langle Q, \Sigma, \delta, I, F \rangle,\tag{5.2}$$

where the five components have the following meanings:

- $Q = \{q_0, q_1, \dots\}$ is a finite set of states,
- $\Sigma = \{\sigma_1, \sigma_2, \dots\}$ is an alphabet of symbols denoting state transitions,
- δ is the next state function mapping a state and a transition symbol to a set of states,
- $I \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of final states.

The automaton can move from a state q to a state q' accepting a symbol σ if the next state map $\delta(q, \sigma)$ is defined and $q' \in \delta(q, \sigma)$. The FSA is

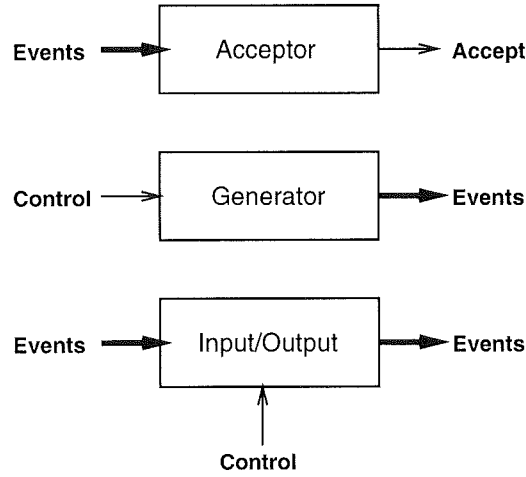


Figure 5.2 Three possibilities to use FSA:s as model components

deterministic if δ is a singleton, $\delta(q, \sigma) = \{q'\}$, for every q and σ , and if the initial state I is a singleton.

It can be noted that the symbols in Σ are neither regarded as inputs nor outputs to the automaton. This depends on how the automaton is used and – if it is used as a model component – on how it interacts with other parts of the system. When an FSA is used for representing discrete behavior, it can be viewed as a submodel interacting with a dynamic environment. The FSA itself has no notion of time but it can be a part of a system where time is relevant. In a continuous time dynamic environment the state transition symbols of an FSA correspond to discrete events. Consider an FSA submodel in three basic configurations shown in Figure 5.2.

1. The FSA works as a string acceptor. The state transitions are caused by input events generated by the environment. A sequence of input events corresponds to an accepted string if the reached state is a final state. The FSA must be deterministic in order to give a well defined behavior.
2. The FSA works as a generator, generating a sequence of events as the output. The FSA is controlled by the environment and it may be non-deterministic. The selection from the set of possible transitions, and the time when a transition should occur, are controlled by mechanisms outside the FSA itself.
3. The FSA works as an input/output process. Some symbols of the al-

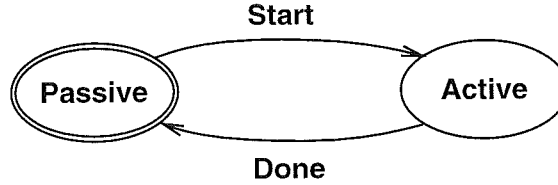


Figure 5.3 An FSA representation of a simple server model.

phabet are considered to be inputs and the corresponding events are generated by the environment. The remaining symbols are considered to be outputs and the events are generated internally, possibly controlled by some other mechanism outside the FSA.

The input/output configuration of FSAs are used for representing the behavior of plants and supervisors in a control system configuration. Control theory and synthesis methods based on an input/output interpretation of the Ramadge and Wonham supervisory control theory [Ramadge and Wonham, 1989] are being developed [Balemi *et al.*, 1993].

Example: Simple server model. As a very simple example of an FSA, consider the model of a server in Figure 5.3. The figure shows a graphical representation of an FSA with two states labeled *Active* and *Passive*, and with the alphabet $\Sigma = \{Start, Done\}$. The state transition function is defined for two cases: $\delta(Passive, Start) = \{Active\}$ and $\delta(Active, Done) = \{Passive\}$. *Passive* is the initial state and the final state. The server model can be used in an input/output configuration with *Start* as an input event and *Done* as an output event. \square

There are several extensions of finite state automata. One extension, called *Timed Automata* [Alur and Dill, 1992] introduces time in the formalism. A timed automata can have a set of clocks which can be restarted as a result of a transition. At any time, a clock represents the time since it was last restarted. A condition can be associated with a transition, so that the transition can occur only when certain timing constraints, based on the clock readings, are fulfilled.

5.4 Generalized Semi-Markov Processes

Generalized Semi-Markov Processes, GSMP, is a formalism for representing discrete event behavior [Glynn, 1989, Ho and Cao, 1991]. A GSMP can be viewed as an extension of an FSA, which adds stochastics and

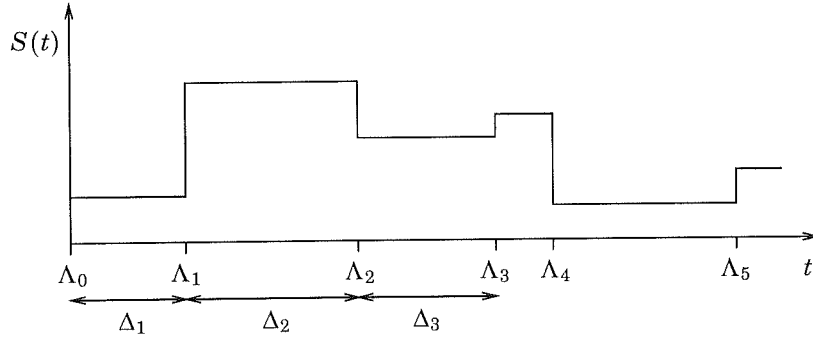


Figure 5.4 An example of an output path generated by a GSMP.

the notion of a continuous time. GSMP models are often used for simulation and analysis of discrete manufacturing systems with queuing models. The definition presented here is based on [Glynn, 1989] with some minor simplifications.

A basic idea behind a GSMP is to give a stochastic discrete event model for a process generating a piecewise constant output signal. The process is assumed to be driven by a sequence of independent random variables. A sample path generated by the process is shown in Figure 5.4. The output process jumps from a discrete level to another at distinct time points. Each jump is a discrete event. The output levels and the event time points depend on random variables. The output signal $S(t)$ can be characterized by the sequence of levels $(S_n : n = 0, 1, \dots)$ and the event times $(\Lambda_n : n = 0, 1, \dots)$ or the time intervals between two consecutive events $\Delta_n = \Lambda_n - \Lambda_{n-1}$. Consider Λ_0 to be the starting time of the process and Λ_1 to be the time of the first event. The output levels S_n belong to an enumerable set of discrete outputs.

We regard the output process $S(t)$ as being generated from an internal state sequence X_n according to

$$\begin{aligned} X_{n+1} &= f(\eta_{n+1}, X_n) \\ S_n &= h_1(X_n) \\ \Delta_n &= h_2(X_n) \\ X_0 &= x_0(\eta_0) \end{aligned} \tag{5.3}$$

where η is a sequence of independent stochastic variables.

Example: Server with queue. As an example of a simple GSMP model assume a system with a server and a queue of waiting customers. The

5.4 Generalized Semi-Markov Processes

time to serve a customer is random with a probability function depending on the number of waiting customers when the service is started. The time between two arriving customers is random with a known probability function. \square

A GSMP can be viewed as “language” for representing the state process (5.3). It can be defined by the tuple

$$M = \langle Q, \Sigma, E, C, \delta, F \rangle, \quad (5.4)$$

where the six components have the following meanings.

- Q is an enumerable set of discrete states. These states are often called *physical* states to distinguish them from the total state X .
- $\Sigma = \{\sigma_1, \sigma_2, \dots\}$ is an enumerable set of event types.
- $E : Q \rightarrow \Sigma'$, where $\Sigma' \subseteq \Sigma$, is a mapping from states to subsets of events. $E(q)$ denotes the set of event types that are active in a state q . An active event means that it may occur as the next event.
- $C = \{c_1, c_2, \dots\}$ is a set of clocks, $c_i \in R^+$, each one associated with an event type. The clock c_i represents the time since event type σ_i last occurred or became active. The total state X consists of the physical state in Q together with all the current clock readings.
- δ is a family of next state probability functions. $\delta(q'; x, \sigma)$ is the probability that next discrete state q_{n+1} is q' , given that the current total state is x and that the next event is σ .
- F is a family of probability distribution functions, F_i , for the lifetime of each active event σ_i . $F_i(T)$ is the probability $P\{t_i \leq T\}$ where t_i is the lifetime random variable for event σ_i , i.e., the time between two consecutive firings of the event.

The evolution of a GSMP can be described by the following algorithm. Assume that the n 'th event has just occurred, the current time is $\Lambda(n)$, and the current physical state is q_n .

1. For each active event, $\sigma_i \in E(q_n)$, generate the residual lifetime random variable from the distribution function $F_i(\cdot; q_n, c_i)$.
2. Let the next event, the trigger event σ^* , be the event with the smallest residual lifetime, and let the holding time in current state, Δ_{n+1} , be the residual lifetime of σ^* .
3. Generate the next physical state q_{n+1} from the probability function $\delta(\cdot; q_n, \sigma^*)$.
4. Update the clocks for every event $\sigma_i \in \Sigma$ so that

$$c_{n+1,i} = \begin{cases} c_{n,i} + \Delta_{n+1} & \text{if } \sigma_i \in E(q_{n+1}), \sigma_i \neq \sigma^* \\ 0 & \text{otherwise} \end{cases}$$

Example: Server with queue (continued). Let the discrete state of the process be the natural number q representing the number of customers waiting in line plus the one currently being served. Let the set of event types be $\Sigma = \{\sigma_1, \sigma_2\}$ where σ_1 represents the arrival of a new customer and σ_2 represents the finished serving of a customer. If there are no customers in the system ($q = 0$) only σ_1 may occur; otherwise both events are active. This gives the active event function:

$$E(q) = \begin{cases} \{\sigma_1\} & \text{if } q = 0 \\ \{\sigma_1, \sigma_2\} & \text{if } q \geq 1 \end{cases}$$

The next state probability functions δ are simple since the state transition is deterministic when the event type is given. The state q increases by one when σ_1 occurs and it decreases by one when σ_2 occurs. Assume that the times between two σ_1 events are normally distributed with the mean m and the standard deviation d . Finally, assume that the service time is equally distributed between 0 and $t_1 + t_2/(1 + q)$, where t_1 and t_2 are constant times and q is the discrete state when the customer starts getting service. \square

A more general form of GSMP is called *time inhomogeneous* GSMP [Glynn, 1989]. In this case, the state transition and residual lifetime probability functions may depend on the complete history of states. In most practical modeling cases, time homogeneousness is achieved by a proper selection of physical states and event types.

The state sequence of a time homogeneous GSMP is a time homogeneous Markov chain. If the residual lifetime probability distribution functions F are all exponential, then the GSMP is a continuous time Markov process. If the clocks are omitted and the residual lifetimes are determined anew for each active event after a transition, based on a probability distribution function $F_e(\cdot; q)$, then the GSMP is a semi-Markov process [Howard, 1971].

5.5 The DEVS formalism

The Discrete Event System Specification (DEVS) formalism was introduced by Zeigler [Zeigler, 1976, Zeigler, 1990, Kim and Zeigler, 1991]. DEVS was constructed to be a convenient representation of models for discrete event simulation. However, DEVS is also a formal representation that can be subject to analysis and mathematical manipulations. In a formal mathematical sense, a DEVS is similar to a generalized semi-Markov process. A difference between DEVS and the other formalisms

presented here is that the DEVS formalism includes structuring concepts which make it possible to structure models in a hierarchical fashion. This is indeed an attractive feature if a formalism is to be used for representing large systems.

A *basic DEVS* model is a primitive DEVS model that is not further decomposed into submodels. It is represented by the tuple

$$M = \langle Q, \Sigma_i, \Sigma_o, \delta_{int}, \delta_{ext}, \delta_t, \lambda \rangle,$$

where the items have the following definitions.

- Q is the internal state set. In general the state space is represented by real-valued, integer-valued, and symbolic variables. The total state includes an additional variable, $c \in R^+$, which is the *local clock*. Let the total state set be denoted by X , $X = Q \times R^+$.
- Σ_i is an enumerable, finite set of input event types.
- Σ_o is an enumerable, finite set of output event types.
- δ_{int} is the internal transition function, $\delta_{int} : X \rightarrow Q$, defining the next state after an *internal* event. An internal event occurs when the local clock has reached the time defined by δ_t .
- δ_{ext} is the external transition function, $\delta_{ext} : X \times \Sigma_i \rightarrow Q$, defining the next state after the occurrence of an input event.
- δ_t is the residual time function, $\delta_t : X \rightarrow \{R^+, \text{inf}\}$, defining the time remaining until the next occurrence of the internal event. If $\delta_t(q) = \text{inf}$, then the internal event is not scheduled in state q .
- λ is the output event function, $\lambda : X \rightarrow \Sigma_o$, used for generating an output event when the internal event occurs. The output event $\lambda(q)$ is emitted just before the internal event occurs in state q .

A DEVS model is interpreted as follows. At any time when an input event occurs, the internal state will jump to the new state defined by $\delta_{ext}(q, c, e)$ where q is the current state, c is the local clock, and e is the input event. Directly after the state transition the local clock is reset to zero. Between events only the clock is changing and increasing at a constant rate. If the residual time function, $\delta_t(q, c)$ with q and c as before, reaches zero, then the internal event occurs. In this case, an output event according to $\lambda(q, c)$ is issued and the state is updated according to $\delta_{int}(q, c)$.

Example: Simple server (continued) Reconsider the simple server example introduced in the FSA section. Assume that the service times are normally distributed with mean value m and variance v . Represent the server as a basic DEVS model with the state $q = \langle s, t_r \rangle$ with $s \in$

$\{Active, Passive\}$, and t_r is the local time when the current job is going to finish. The model has one input event called *Start* and one output event called *Ready*. The internal transition event occurs when the current job is finished and the state changes from *Active* to *Passive*. This gives the internal transition function:

$$\delta_{int}(q, c) = \langle Passive, 0 \rangle.$$

If the input event occurs when the state is *Passive* the state changes to *Active* and the finishing time is generated from the normal distribution. If the input event occurs when the state is *Active*, the system remains active with current job but the local clock is reset and t_r must be updated with the remaining service time. Thus, the internal transition function becomes:

$$\delta_{ext}(\langle q, t_r \rangle, c) = \begin{cases} \langle Active, \eta \rangle & \text{if } q = Passive \\ \langle q, t_r - c \rangle & \text{if } q = Active \end{cases}$$

where η is a random variable generated from the distribution function of the service times. Remember that the local clock c is reset to zero by the external event. The residual time function becomes:

$$\delta_t(\langle q, t_r \rangle, c) = \begin{cases} \inf & \text{if } q = Passive \\ t_r - c & \text{if } q = Active \end{cases}$$

Finally, the output event function becomes:

$$\lambda(\langle q, t_r \rangle, c) = Done$$

□

Several basic DEVS models may be connected to form a multi-component, coupled DEVS model. A coupled DEVS model is equivalent to a basic DEVS model and can be used as a component in another coupled DEVS model. Since the DEVS formalism is closed under composition it is possible to define arbitrarily deep hierarchical models. A coupled model is defined by

- a set of component DEVS models,
- sets of input and output events as in a basic model,
- a set of connections, called *influences*, which are pairs of input events and output events of this model and of the components, and
- a tie-breaking selector used when several components have internal events scheduled at current time. The selector determines which one of the component models should be allowed to carry out its transition first.

In a coupled DEVS model, it is also possible to propagate data that is dependent on the current state of the event issuing component, to the receiving components. An influence is conceptually similar to a connection between structured terminals in Omola.

The DEVS formalism can be compared to the GSMP formalism defined previously. In a GSMP model the total state is represented by the physical, countable state and the clocks associated with each state. In a DEVS model, the total state can belong to any uncountable and infinite set. In this sense, the modeling power of DEVS is greater than the modeling power of GSMP. However, for the special case that the state of a DEVS model is only represented by discrete variables, the model is equivalent to a GSMP.

Software for object-oriented modeling and simulation of DEVS models has been implemented [Zeigler, 1990, Kim and Zeigler, 1991].

5.6 Hybrid Automata

Hybrid Automata, HA, is a formalism for representing systems with behavior involving discrete events as well as continuous time elements. A typical example is a model of a physical process controlled by a computer algorithm. The hybrid automata model presented here is introduced in [Alur *et al.*, 1993] and is based on [Maler *et al.*, 1992]. It can be seen as an extension of the FSA formalism, adding the notion of time and continuous state evolution.

A *hybrid system* is the tuple

$$A = \langle Q, V_D, \mu_1, \mu_2, \mu_3 \rangle,$$

where the five components have the following definitions and meanings.

- Q is a finite set of discrete states called *locations*. They correspond to what previously have been called physical states.
- V_D is a finite set of real-valued data variables. They represent the continuous state of the system. The set of all states is denoted by $\Sigma_D \subseteq R^n$, where n is the number of variables in V_D .
- μ_1 is a labeling function that assigns to each location a set of possible *activities*. An activity is the behavior of the system with respect to the evolution of the continuous state within a particular location. It is a smooth function from R (time) to Σ_D .
- μ_2 is a labeling function that assigns to each location $l \in Q$ a *location invariant* set $\mu_2(l) \subseteq \Sigma_D$. If the system is in the physical state l , then

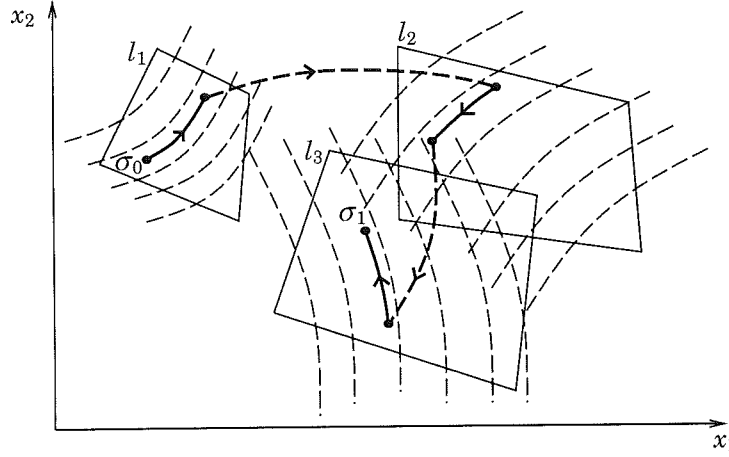


Figure 5.5 An example of a trace of a hybrid system. The initial state of the system is σ_0 in location l_1 . The continuous state changes according to the behavior defined for location l_1 . After some time a transition occurs and the state is changed discontinuously to a new state in location l_2 , etc.

the continuous state has to be in $\mu_2(l)$, otherwise an exception is triggered.

- μ_3 is a labeling function that assigns to each pair of locations a transition relation $\mu_3(l_1, l_2) \subseteq \Sigma_D^2$. $\mu_3(l_1, l_2)$ is a mapping from a continuous state in location l_1 to a continuous state in location l_2 .

The total state of the hybrid system is represented by the location and the continuous state. The state can change instantaneously, at discrete events, according to the transition relations μ_3 or continuously, by elapse of time, according to the activities defined by μ_1 . The invariants μ_2 must be fulfilled at all times. A *trace* of a hybrid system is a state sequence that fulfills all the restrictions. An example of a trace of a hybrid system is displayed graphically as a phase plot in Figure 5.5. The hybrid system has two continuous state variables, named x_1 and x_2 . Three locations are indicated with their corresponding invariant sets and possible activities. The trace starts in state σ_0 and ends in state σ_1 .

The definition of a hybrid system does not define when the transition events occur. A hybrid system can be viewed as an input-output model of a controlled process. The input to the process consists of discrete events causing state transitions in the HA. The output from the process is a function of time defined by some mapping from the discrete and the continuous states. The objective of the controller is to monitor the process and to issue control events so that the HA fulfills its invariants and

additional specific control objectives.

A *linear* hybrid system is an important special case of a hybrid system. A hybrid system is linear if its activities, invariants, and state transitions can be defined by linear expressions over the set V_D . In a linear hybrid system, the continuous state changes linearly in time. This means that all the activities can be represented in the form $\dot{x}_i = k_{i,l}$ where $x_i \in V_d$ and $k_{i,l}$ is a constant depending on the current location. Note that the continuous behavior of a linear hybrid system is more restricted than the behavior of an ordinary linear system in the form $\dot{x} = Ax + Bu$. It is possible to do some analysis on linear hybrid systems. For example, there are algorithms for verifying that all possible traces of a linear hybrid system fulfill a given invariant [Alur *et al.*, 1993]. In [Tittus and Egardt, 1993] algorithms for analyzing controllability and for synthesizing control laws for linear hybrid systems are presented.

5.7 Grafcet

Grafcet is a graphical language for modeling and specification of sequential processes. Contrary to Finite State Machines, Grafcet can represent concurrent activities. It was developed by the French organization AFCET [AFCET, 1977]. The goal was to standardize a well-defined, unambiguous, and practically useful language for describing sequential and parallel activities in automation. The Grafcet standard is now adopted by IEC [IEC, 1988] under the notion of *sequential function charts* or SFC.

Grafcet can be seen as a special kind of Petri net, introduced by C. A. Petri in his dissertation in 1962. A lot of publications exist on Petri nets, see for example [Reisig, 1982, Murata, 1989, David and Alla, 1992]. A Petri net is a graphical representation and a mathematical formalism used as a modeling tool. It is suitable for formal analysis and as a specification and simulation language for many kinds of discrete event systems, like communication systems, control systems, and information processing systems. Petri nets exist in many different versions.

The name “Grafcet” is used in this thesis for the formalism as such, while “a grafcet” denotes a particular model defined in Grafcet.

The basic elements of Grafcet

An example grafcet is shown in Figure 5.6. The two most basic elements are *step*, drawn as a square box and representing the state of the system, and *transition* drawn as a horizontal bar. Other fundamental elements are *parallel split* and *synchronization*, both drawn as double horizontal

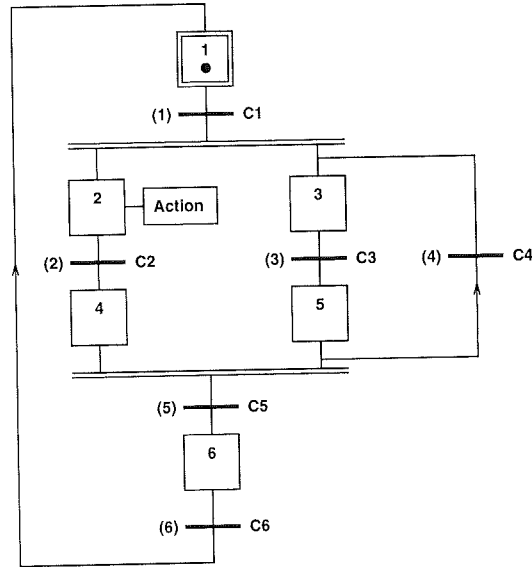


Figure 5.6 Example of a simple grafset in its initial state.

lines. Steps and transitions are labeled by unique numbers. In Figure 5.6 there are six steps labeled 1 to 6, and six transitions labeled (1) to (6). The graph also includes a parallel split (the upper one of the two double bar symbols) and a synchronization (the lower one). The basic elements of a grafset are connected in a net by *directed links* (or *arcs*). An arc always runs from a step to a transition or from a transition to a step. Arcs drawn without an arrow are assumed to be directed downwards in the diagram.

The steps of a grafset represent the state of the modeled system. Each step can be either *active* or *passive*. The current state of the system may be indicated by filled dots in the active steps. Steps drawn by double squares are *initial* steps. They are active in the initial state of the system. In Figure 5.6 the initial state is shown with Step 1 active and all the other steps passive.

The state of a grafset evolves by firing of transitions. Each transition is associated with a firing condition, also called a *receptivity*, which is a logical function of input variables and internal states. A transition is said to be *enabled* if all preceding steps are active. An enabled transition is *fireable* if its condition is true. Firing a fireable transition means that all preceding steps are deactivated while all the following steps are activated.

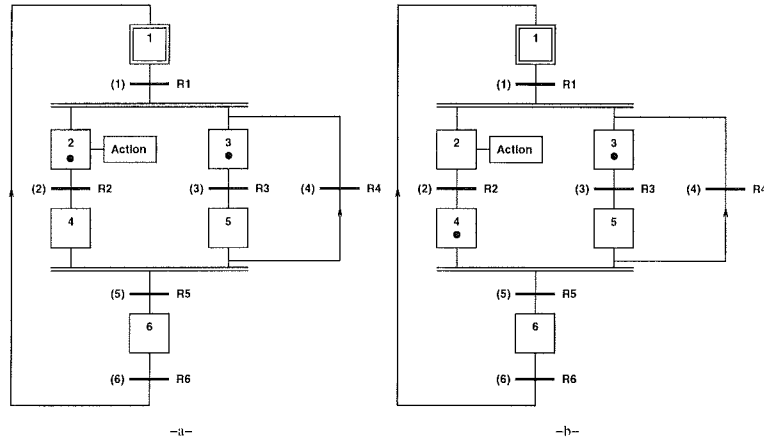


Figure 5.7 The grafcet shown after transition (1) has fired (a) and then after transition (2) has fired (b).

For example, in Figure 5.6 Transition (1) is enabled since Step 1 is active. When the condition C1 becomes true, the transition is fired and the result is that Step 1 becomes passive while Step 2 and Step 3 both become active. The new situation is shown in Figure 5.7.a. Now, Transition (2) and Transition (3) are enabled. If we assume that the condition of Transition (2) becomes true, the transition is fired and the situation shown in Figure 5.7.b is reached.

Inputs and Actions

A grafcet interacts with the environment which, for example, consists of another grafcet, a continuous time model, or a real physical plant. The inputs to the grafcet consist of the transition conditions. The conditions may depend on external variables which are then considered as input variables. The outputs from the grafcet are defined by the actions associated with the steps.

A transition condition may be a purely logical function of input variables and the state of steps in the same grafcet. Such a transition is fired when the transition is enabled and the condition changes to true, as well as when the condition is true when the transition becomes enabled by the preceding steps. It is also possible to have *edge-triggered* firing conditions. In this case the transition is only fired if it is already enabled when the condition turns from false to true.

Outputs are associated by the steps of the grafcet. For example, an

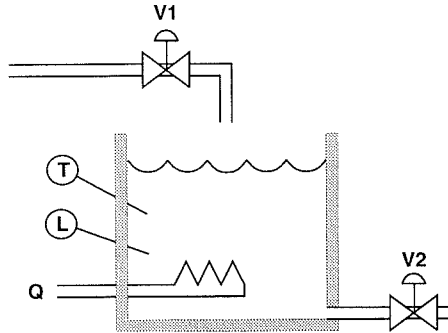


Figure 5.8 A tank with a heater.

action may consist of a boolean output variable that is true when the step is active. It is also possible to have *impulse actions* which take place when the step switches from passive to active.

Figure 5.9 shows the previously discussed grafset augmented with conditions and actions in order to control a plant consisting of a storage tank with a heater shown in Figure 5.8. The plant provides measurements of the temperature T , and the level L . The control variables are the valves $V1$ and $V2$, and the heating Q . The control variables are all boolean variables for opening and closing the valves and switching the heat on or off. The Grafset controller has two input conditions, *Start* and *Empty*, which are commands from the operator or higher level controllers. When the controller is in its initial state, it is waiting for the *Start* condition of transition (1). When the operator pushes the start button, the transition fires and Step 2 and Step 3 are activated. This means that $V1$ is opened and heat is switched on. The valve remains open until the condition of Transition (2) becomes true when the tank level becomes larger than the fixed value L_{max} . The heating is on until the condition of Transition (3) becomes true when the temperature reaches the fixed value T_{max} . The operator can empty the tank by pushing the *Empty* button when Step 4 and Step 5 are active. In this case, Step 6 becomes active and the valve $V2$ is opened. When the level reaches below L_{min} , Transition (6) can fire and the process is ready for a new cycle. Steps 3, Step 5, Transition (3), and Transition (4) constitute a temperature control cycle. If the temperature goes below T_{min} when Step 5 is active, Transition (4) fires, Step 3 becomes active again, and the heating is restarted.

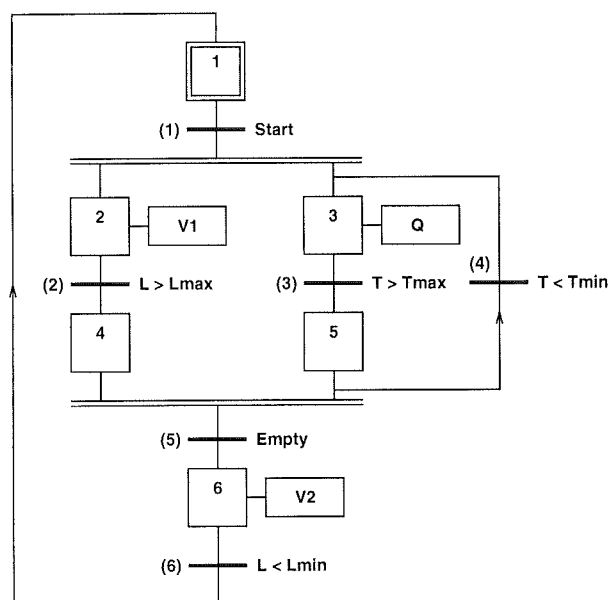


Figure 5.9 A grafset for controlling the tank with the heater.

5.8 Other approaches

A modeling and simulation package called *gPROMS* (general PROcess Modeling System) has been developed at Imperial College [Barton and Pantelides, 1991]. The language and the simulation environment are general-purpose but they are developed with a special aim to support modeling of chemical batch processes in combination with continuous time processes.

Similar to Omola, *gPROMS* uses general differential and algebraic equations to represent continuous time behavior. The concepts for describing discrete behavior are based on a set of *elementary tasks*. These tasks include *RESET* which replaces an input time function with another, and *REPLACE* which replaces one or more equations with an equal number of new equations. An elementary task called *REINITIAL* is used for defining discontinuous state changes.

Another approach to hybrid system modeling is based on bond graphs (see Section 2.1) and is presented in [Strömberg, 1994]. The approach focuses on physical models with local mode changes such as switches in electric circuits, dry friction with stick slip motion, etc. An ideal switch

is introduced as a new bond graph element and is used for representing mode changes in different domains. The model representation is suitable for formal analysis and simulation.

5.9 A hybrid model for Omola

A hybrid model formalism based on the formalisms discussed earlier in this chapter and on the DAE representation for continuous time models, is introduced. The formalism serves as a basis for extending Omola in order to represent discrete event and hybrid systems. The formalism is referred to as the OHM (Omola Hybrid Model).

A solid mathematical formalism, with a well-defined interpretation, is essential as a foundation for a modeling language like Omola. First, it makes it easier to translate other formalisms into Omola and to integrate model components developed in different formalisms, into a composite model. Secondly, a well-defined formalism makes Omola models portable and easier to translate into other simulation languages and into executable, real-time control programs. The translation of models between Omola and other formalisms can often be done mechanically. Thirdly, it facilitates the implementation of the simulation algorithm and provides a way of verifying its correctness.

Let a hybrid model be represented by the tuple

$$M = \langle Q, X, E, g, h, \Phi, \Delta \rangle,$$

where the components have the following definitions.

- $Q = (q_1 \dots q_{n_q})$ an array of variables representing the discrete state of the system. The variables may be of real, integer, or symbolic type.
- $X = (x_1 \dots x_{n_x})$ is an array of real variables representing the continuous state of the system.
- $E = \{e_1, \dots, e_{n_e}\}$ is a set of event types.
- $g = \{g_1, \dots, g_n\}$ is a set of functions defining the continuous behavior by the equations:

$$g_i(X(t), \dot{X}(t), Q_t) = 0, \quad g_i \in g \quad (5.5)$$

The notation ' Q_t ' is used to indicate that the discrete state Q is updated at discrete time points.

- $h = \{h_1, \dots, h_{n_h}\}$ is a set of Boolean functions such that

$$h_i(X(t), \dot{X}(t), Q_t) = \text{true}, \quad h_i \in h \quad (5.6)$$

define the *invariants* of the model. The invariant functions divide the state space in two parts: the set of *admissible* states where all h_i are true and the remaining states where at least one of the constraint functions are false.

- Φ is the map $h \rightarrow E$ associating each invariant function with an event type.
- $\Delta = \{\delta_1, \dots, \delta_{n_c}\}$ is a set of vector valued functions, each one associated with an event type, $e_i \in E$, such that when e_i occurs then the equation

$$\delta_i(X_a, \dot{X}_a, Q_a, X_b, \dot{X}_b, Q_b) = 0 \quad (5.7)$$

is fulfilled. X_a and Q_a refer to the state immediately after the event, while X_b and Q_b refer to the state just before the event.

The framework describes a system where the state evolves in two ways: continuously in time by changing the values of X , and by instantaneous changes in the total state represented by the variables Q and X . Instantaneous state changes are called events. They occur when the system crosses the border of the set of admissible states, defined by the invariant functions.

Continuous time behavior

The constraint equations defined by g define the continuous behavior of the model. This means that continuous behavior is represented by a differential and algebraic equation (DAE) system. Since the discrete state Q is constant in time between events, its time dependency is indicated by a subscript t . This means that the elements of Q appear as time constant, known parameters, as far as the continuous behavior is concerned. The constraint equations (5.5) may also include static relations between discrete variables.

Discrete events

Events occur asynchronously, i.e., they are affecting the system one at a time, causing a sequence of state transitions. The invariants h and the map Φ define an event to occur when an invariant is violated, i.e., when any $h_i(t)$ turns from true to false. We say that an event $e = \Phi(h_i)$ becomes enabled at time t_e if, and only if, $h_i(t_e - \varepsilon)$ is true and $h_i(t + \varepsilon)$ is false for any ε and some δ such that $0 < \varepsilon \leq \delta$, without considering the effect of the event.

Firing of an event may cause a discontinuous state transition. The transition is defined by the set of functions in Δ associated with the detected event type. It is the solution of the equation system defined by

(5.5) and (5.7) that defines the state of the system immediately after the event, when the current state (X_b , \dot{X}_b , and Q_b) is known. Firing of an event may cause new invariants to be violated. The OHM formalism proclaims that events are fired until a stable situation is reached when all invariants are fulfilled.

Event sequences

It may occur that several events become enabled at the same time point. In this case the events must be fired in sequence, according to some deterministic or non-deterministic ordering. In many cases, the order in which simultaneously enabled events are fired is not significant because different events may affect different parts of the model without any direct interaction. However, it is easy to find examples where the firing order is significant. From a model formalistic viewpoint, it is natural to define that simultaneously enabled events are fired in a non-deterministic order. On the other hand, there are advantages of having a deterministic model, with a well-defined firing order of simultaneously enabled events. A deterministic approach, where the ordering of events in E determines the firing order, is chosen for OHM. However, it is easy to extend a simulator to use a random firing order as an alternative.

Comparing OHM with HA

The suggested model is inspired by Hybrid Automata [Alur *et al.*, 1993]. A difference is that OHM is complete in the sense that a model also defines when events should occur. A Hybrid Automata model can be viewed as a model of a process where the events are control actions issued by a mechanism outside the formalism. In OHM it is also possible to represent the controller, i.e., to define when the control events should occur. This is done by associating event types with invariant expressions.

Another difference between HA and OHM is that OHM allows a continuous state space for the discrete time state, represented by Q . This is not a formal difference but a technical convenience. We can view the discrete state Q in OHM, as being represented by a set of real-valued variables Q_r , and a set of discrete-valued variables Q_d . Q_d represents the enumerable set of physical states corresponding to the locations in a HA. Since Q_r represents an uncountable state, the member variables belong to the set V_D if the OHM is interpreted as a HA. The variables in Q_r are special in the sense that they are constant between events, i.e., they have zero time derivatives in every location.

Limitations of the OHM

A limitation of the Omola hybrid model is that the set of continuous state variables X and the set of behavior functions g are time invariant. This means that models where objects with local state and behavior, are created and destroyed, cannot be represented. Such models often appear in pure discrete event models of, for example, production systems and job shops. This kind of models are less frequent in process modeling and in automatic control.

Models of batch processes have properties similar to discrete event production systems as well as continuous processes. The batch process contains several processing units which can be activated or deactivated. If the batch process is represented in Omola, the deactivated units are also parts of the total model at all times. The problem is mainly a matter of efficiency.

An advantage of the OHM is that the structural properties of the model can be analyzed in advance, before the actual behavior is determined by simulation. Consider a formalism with the more general approach where a separate continuous time model, with a different set of state variables and equations, is associated with every discrete mode. Unless the model has very few discrete modes, every discrete state cannot be analyzed in advance. Instead, every new mode is analyzed when it occurs in a simulation. In the more general representation, the problem also arises how to represent the mode transitions in a general way in order to avoid defining all possible transitions explicitly. This is a topic for future research.

5.10 Simulation of hybrid models

A *simulation problem* is a hybrid model and a set of initial conditions. The initial conditions are additional constraints on the model variables and derivatives at the initial time t_0 . The simulation problem is well-defined if the initial conditions are consistent and determine initial values for all variables and if the model is specific enough to define a unique solution up to some final time t_1 .

The initial conditions of a simulation problem are often specified by the user in the simulation tool. The model itself may also define default initial values for variables. The user specifies the initial time, t_0 , and the initial values of some of the continuous variables, derivatives, and discrete variables, so that the equation $g(x_0, \dot{x}_0, Q_0) = 0$ can be solved for all remaining unknown initial values. Generally we can define the initial

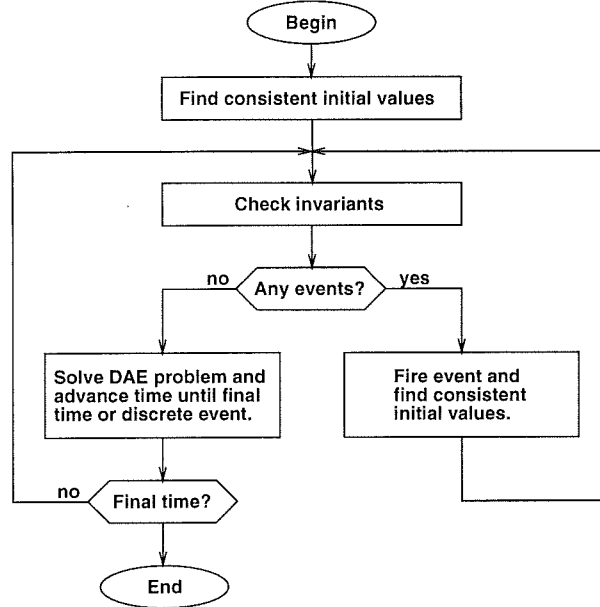


Figure 5.10 The simulation algorithm for Omola Hybrid Models.

value problem by the initial condition function g_I , so that the equation system

$$\begin{aligned} g(x_0, \dot{x}_0, Q_0) &= 0 \\ g_I(x_0, \dot{x}_0, Q_0) &= 0 \end{aligned} \quad (5.8)$$

can be solved. The vector function g_I includes constraints on derivatives, implied by a high index DAE problem, and user defined initial values. High index DAE problems are discussed in Chapter 7.

Assuming consistent initial values, the simulation problem can be reduced into three subproblems and can be solved by the simple algorithm displayed in Figure 5.10. The subproblems are:

1. to solve the DAE problem,
2. to detect events, and
3. to execute events.

Solving the DAE problem means to advance the time so that the continuous time behavior equations (5.5) are fulfilled. It is assumed that the initial state is consistent. This problem implies solving non-linear systems of equations and numerical integration. The methods are discussed in more detail in Chapter 8.

The problem of detecting events means to check the invariants h continuously during DAE solving, and to detect when any of the functions become false. Methods for this are also discussed in Chapter 8.

To execute an event means to determine the variables affected by the event and to compute new values for them. The new values must be consistent with the continuous behavior equations (5.5) so that it works as an initial state for a new DAE problem. Therefore, event execution is also referred to as the *restart problem* which is discussed in the following.

Some terminology that is useful in the following discussions will now be introduced. The Boolean complements of the invariant expressions in $h = \{h_1, \dots, h_{n_h}\}$, are called event *conditions*. A particular event e may have several conditions defined by the set $\{h_i : \Phi(h_i) = e\}$. An event condition that depends on at least one continuous variable is called a *continuous* condition. An event condition that only depends on discrete variables is called a *discrete* condition. An event is *enabled* if it has at least one of its conditions equal to true, i.e., an invariant is violated, at some point in the simulation algorithm. An event e_i is *fired* when its corresponding state update equation δ_i is applied and the restart problem (5.9) is solved.

Solving the restart problem

After an event e_i has been detected, the equation system

$$\begin{aligned} g(X_a, \dot{X}_a, Q_a) &= 0 \\ \delta_i(X_a, \dot{X}_a, Q_a, X_b, \dot{X}_b, Q_b) &= 0 \end{aligned} \quad (5.9)$$

is solved for a unique and consistent initial state. This problem is similar to the initial value problem (5.8).

If the restart problem defined by (5.9) can be algebraically manipulated into the following form, it becomes easy to solve:

$$\begin{aligned} Q_a &= \delta_{1,i}(X_b, \dot{X}_b, Q_b) \\ X_a &= \delta_{2,i}(X_b, \dot{X}_b, Q_b, Q_a) \\ \dot{X} &= f(X, Q_a) \end{aligned} \quad (5.10)$$

This means that after the event has been detected, the new discrete state Q_a can be computed from the state immediately before the event. Then the new continuous state X_a can be computed. Finally, since the continuous time problem is on explicit state-space form, the new initial state is trivially consistent and the numerical integration can be resumed.

The general mathematical restart problem defined by the equation system (5.9) is difficult to solve, since it contains continuous as well as discrete unknown variables. However, modeling of physical systems with a conscious methodology can result in restart problems that can be manipulated into a simple form like (5.10) and can be solved with available numerical methods.

The restart problem can be analyzed symbolically, before simulation, in order to determine how hard the problem is to solve. This analysis is similar to the analysis applied to a DAE problem in order to verify that it is structurally well-defined. The steps taken in the analysis include block triangularization of the equation system which is also needed in order to transform the problem into executable and efficient simulation code. Symbolic manipulation of the DAE problem and of the event restart problems are discussed in more detail in Chapter 8.

Event sequences

In a valid trace of an Omola hybrid model all the invariants of the model are satisfied at all times. However, since the solution of the restart problem (5.9) may not fulfill all the invariants, an new event may become enabled as a result of firing one. This means that in some cases a sequence of events is fired, at the same time instance, until all the constraints are fulfilled. The main rule is that two different events never occur simultaneously. If several events are enabled at the same time, then they are fired in sequence according to their index, i.e., in their order of definition. It is necessary to define the firing algorithm more precisely. For example, assume that the continuous event conditions for the events e_1 and e_2 are detected at the same time. Then they are both enabled. First, event e_1 is fired according to the ordering rule. As a result, one of three different things may happen. One possibility is that all the events of the model are disabled and the continuous simulation can be resumed. Another possibility is that one of the events is now enabled so it must be fired. The third possibility is that both events are enabled after the firing of the first one. In this case, we can fire the first event again according to the ordering rule, or we can argue that the second event should be fired first since it was originally enabled at the same time as the first one. There seems to be at least two reasonable algorithms for event firing. The first alternative can be defined as:

1. Fire the first one of all currently enabled events.
2. Check for enabled events. If no event is enabled, resume continuous simulation, otherwise go to 1.

The second alternative is defined as:

1. Determine the ordered set $E' \subseteq E$ of enabled events.
2. If E' is empty, resume continuous simulation.
3. Fire the first event in E' .
4. For each of the remaining events in E' , check if it is still enabled and if so fire it.
5. Go to 1.

The second algorithm is chosen as being the correct way of interpreting an Omola model. This algorithm is more fair, since it considers all events that are enabled at the same time, before it re-evaluates the condition for the first one. It is more complicated to implement but it may in some cases simplify modeling. An example of that is given when Grafcet simulation is discussed in Chapter 6.

It is possible to formulate models where the chosen event firing algorithm never halts. In this case the model is not well-defined. There is no way to detect this problem analytically without running the simulation, except for very simple cases. In practice, since most models are defined so that the same event type rarely fires more than a few times in a sequence, a simulator can detect a suspected infinite loop and suspend the simulation with a warning message.

The suggested execution schemes for event sequences assume that it is possible to compute a consistent state after each separate event firing. It could be possible to relax this requirement and allow a sequence of events to be fired where only the last event must result in a consistent initial state. It is not clear whether this is a desirable feature or not and it is not tested in OmSim.

Scheduled events

In practical discrete event and combined modeling it is common that the firing times for some events are known in advance, at some earlier time in the simulation. Such events are represented in the formalism as having associated invariants defined as $h_i \equiv t < t_i$, where t is the current time and t_i is a discrete variable representing the future time when the event should be fired. Events with this kind of conditions are called *time events*. Events that are not scheduled are called *state events*.

Time events can be scheduled in advance. The simulator runs more efficiently if this type of events are treated specially. The simulator can have a queue of scheduled events, ordered in time. In that case, the continuous time solving is always stopped at the next scheduled event, unless a state event is detected at an earlier time.

5.11 Summary

This chapter has given a brief overview of some commonly used modeling formalisms for discrete event and hybrid systems. These formalisms inspired the definition of a hybrid model formalism to be used as basis for Omola. The Omola Hybrid Model (OHM) formalism is general enough to represent several different classes of discrete event, continuous time, and hybrid models. It is based on differential and algebraic equations for representation of continuous time behavior, and on boolean invariants for defining discrete event conditions. The effect of each discrete event type is defined as a set of general equations, relating the model states immediately before and after the event.

The purpose of the OHM is to serve as an intermediate representation between high-level Omola models and more special-purpose representations used for numerical simulation or design. A simulation algorithm for OHM representations was outlined and discussed in general terms.

OHM cannot represent models with variable dimension. That is, systems where the total number of state variables and equations change as a result of discrete events. The formalism may be extended to cover also these kinds of systems but it requires more effort in the implementation of an efficient simulator.

6

Discrete Events in Omola

There are many different ways to represent discrete event and hybrid models. A common property is their capability of representing discontinuous state changes. The aim is to provide Omola with basic facilities for representing discrete events, so that most high level discrete event and hybrid model formalisms can be supported. The discrete event can be regarded as a fundamental concept, common to all hybrid model formalisms. This chapter extends Omola with language constructions for defining discrete events, and for defining event interaction between submodels. The additional basic concepts together with the ordinary model structuring facilities make it possible to support different higher level hybrid model formalisms in Omola.

6.1 Introduction

The *event* is a fundamental concept in hybrid models. Every discrete state change in a simulated model is caused by an event. Events occur because the time or the state of the system has reached a certain point. A hybrid model defines discrete behavior in terms of *event types*. The concepts can be illustrated by the following examples.

- Assume a model of a digital controller operating on a continuous time process. At regular time instances, the controller samples the continuous measurements from the process and computes a new control value. Each sample instance, occurring when the model is simulated, is an event of the sampling event type defined in the controller model.
- Assume a model of the friction force between two moving bodies in physical contact. The friction model is concerned with two kinds of discrete events defined by two event types in the model. One type of event occurs when the relative velocity of the two bodies reaches zero

and they may get stuck to each other. The other type of event occurs when the static friction force between the bodies exceeds a certain maximum and they start sliding again.

- Assume a model of a processing cell in a discrete manufacturing system. The arrival of a new item to the input buffer is modeled by an event type. The completed processing of an item is modeled as another event type.

An event type defines how a particular event affects the system. The event type definition can also include the conditions for the event to occur.

The event type is the basic language concept in Omola for defining hybrid models. The concept of an event precedes the concept of a discontinuous state change in the sense that any discontinuous state change is always caused by the occurrence of an event. This is different from Dymola which does not recognize events as a basic language concept and does not allow event types to be defined explicitly [Elmqvist *et al.*, 1993]. However, the concept of an event type is useful when events are propagated between different parts of a model. An event type is abstract in the sense that it does not have to be associated with the discontinuity of a particular variable. The concept is also useful in the definition of other discrete modeling facilities.

A reason for having events as a basic language concept is the desire to define clean and abstract interfaces between model components. If a submodel needs to communicate to the environment the fact that some internal discontinuity has occurred, it defines a terminal that is an *output event*. This terminal can then be connected to an *input event* terminal of another submodel, that can react on the event and take appropriate internal actions. A modeling language where events are not part of the language must represent the same type of submodel interaction using, for example, Boolean variables in the terminals. In this case the interacting submodels must agree on how to interpret a change in the interface variable: Is the important event occurring when the interface variable changes from false to true, from true to false, or both? If one of the two former conventions are adopted, then one of the interacting processes must take care of resetting the interaction variable before the next event occurs. This kind of implicit conventions, and possible confusions, between communicating processes are avoided if the more abstract concept of a pure event is used.

To summarize, the concept of an event, defined in a model as an event type, makes a clear distinction between

- the event *as such*, i.e., the idea that something occurs at a specific instance in time,

- the condition for the event to occur, and
- the effect on the system when the event occurs.

This chapter describes how events are defined in Omola. It introduces the syntactic constructions and defines them in terms of the Omola Hybrid Model (OHM) formalism, defined in the previous chapter. This is followed by a description of the event propagation and synchronization facilities. Finally, some examples are given, relating to the traditional hybrid model formalisms discussed in the previous chapter.

6.2 Defining discrete events

In this section, the fundamental concepts for defining discrete event models in Omola are introduced.

Discrete variables

Discrete variables are declared to change value only at discrete time points as the result of events. Ordinary (continuous time) variables change in general continuously by time and discontinuously by events. The optional key-word `DISCRETE` is used to declare a discrete variable. For example:

```
X TYPE DISCRETE Real;
```

The effect of declaring a variable as discrete is the same as defining the time derivative of a continuous variable to be equal to zero. Only real variables and matrices can change continuously. Therefore, variables of discrete types, such as integer and boolean, may be considered as discrete variables even if this is not explicitly declared.

A continuous variable may behave as a discrete variable as a result of the total model behavior. This is the case for a variable that can be computed from an equation that otherwise only depends on discrete variables or parameters, or a variable with the time derivative set equal to zero. Such variables can be found in the model analysis and eliminated from the continuous time simulation problem. Parameters are similar to discrete variables in the way they affect the continuous time behavior of the model.

Event type declarations

Events occur in hybrid systems and when a hybrid model is simulated. Events are defined in Omola models as *event type* declarations. Event

Chapter 6. Discrete Events in Omola

```
M ISA Model WITH
  x TYPE Real;
  x' = 1-x;
  E ISAN Event WITH
    condition := x > 0.5;
    NEW(x) := 0;
  END;
END;
```

Listing 6.1 An example of a model with a discrete event.

types are model components defined as subclasses of the predefined class Event, which is defined as:

```
Event ISA Class WITH
  condition TYPE Boolean := false;
END;
```

The attribute condition defines a Boolean condition for the event to occur. It is the logical complement of an invariant in the hybrid model formalism. The condition is bound to the constant false in Event but it is typically rebound to a Boolean condition, depending on continuous time variables, when a particular event type is defined as a subclass of Event. An event of the defined type occurs when its condition changes from false to true. The effect of the event, called the event *action*, is defined as a set of equations defining a state transition. Listing 6.1 shows an example of a complete model with an event. The model has a continuous time variable x and a differential equation. It defines the event E to occur as soon as x becomes larger than 0.5. As a result of the event, the value of x will change discontinuously to zero. The definition of E specializes Event by redefining the condition and adding an equation attribute. The equation uses the operator new to refer to the value of x immediately after the event.

As illustrated by the example in Listing 6.1, an event definition appears as a component in a model class and it has the structure:

```
<event name> ISAN Event WITH
  condition := <logical expression>;
  <body of actions>
END;
```

There is also an alternative, more compact syntax for defining events without names. This syntax is presented in Section 6.4. Attributes of an

event class are indicated as the “body of actions”. They define the effect of the event on the system. Not all kinds of attributes have well-defined meanings in an event definition. The following attributes are accepted:

- *Variable definitions with bindings* are always regarded as discrete variables and they are computed as a result of the event. The binding expression may depend on any variable, accessible in the scope, with or without the new-operator.
- *Equations* (or assignments) involving at least one new-operated variable.
- *Side effect actions* like, for example, calls to the event scheduler.

The different parts of an event definition will now be discussed in more detail.

Event Conditions

An event condition is the logical expression appearing as the binding to the condition attribute. The event condition is the logical complement to an invariant in the Omola Hybrid Model formalism. For example, an event condition defined as:

condition := $x > 0$;

defines the invariant $x \leq 0$.

Event conditions that involve continuous time variables are called *continuous* event conditions. The simulator has to treat continuous event conditions specially, since they have to be evaluated along the continuous time solution by the numerical integration routine. Non-continuous conditions only have to be evaluated immediately after discrete state transition caused by an event.

Continuous event conditions involve relational operators on continuous time expressions. The relational operators available in Omola are the inequality operators $<$, $<=$, $>$, $>=$, and the equality operator $=$.

The standard boolean operators AND, OR, and NOT can be used in event conditions to logically combine several continuous and discrete conditions. It is possible to activate and deactivate continuous conditions selectively in different discrete states. For example, assume i is a discrete integer variable and x and y are continuous variables in the following event condition.

condition := $(i == 0 \text{ AND } x > 0) \text{ OR } (i == 1 \text{ AND } y < 0)$;

In this case, we can say that the invariant $x \leq 0$ is active only in the discrete states characterized by $i = 0$, while the invariant $y \geq 0$ is active only in the discrete states characterized by $i = 1$.

Edge triggered events

Event conditions may include a special operator called the *transition operator*. It is a prefix operator on Boolean expressions and it is written in Omola as '^' (the exponential operator). The transition operator is used to indicate that the event occurs when the condition changes from false to true only. In order to fire the event for the same condition again, it must become false and then true again. A condition with the transition operator does not define an invariant in the ordinary OHM meaning. However, the transition condition can be transformed into an ordinary invariant condition by introducing an additional Boolean variable. For example, regard the event definition:

```
E ISAN Event WITH
  condition := ^(x > 0);
END;
```

The event is *edge triggered* which means that it is fired once each time x becomes larger than zero. The same event will not occur again until x has become negative and then reaches zero again. If the condition was defined without the transition operator, the model would prohibit a positive x . The event with the transition operator is equivalent to a model with the following definitions:

```
Econd TYPE Boolean;
E ISAN Event WITH
  condition := NOT Econd AND x > 0;
  NEW(Econd) := true;
END;
Ereset ISAN Event WITH
  condition := Econd AND x <= 0;
  NEW(Econd) := false;
END;
```

Since transition events are rather common, the transition operator is much more convenient to use, compared to defining the same behavior using only invariant conditions. The simulator can also represent the transition conditions for most efficient simulation.

From a formal, model semantic point of view, the event conditions are ordinary logical expressions with continuous and discrete variables. In order to achieve an efficient and numerically sound simulation model, they need to be manipulated and transformed into a more suitable form. This is discussed in more detail in Chapter 8.

Event Actions

Event actions are defined by attributes in the event class definition. There are a few different types of actions that can be defined for an event. A discontinuous state update is a kind of action defined by equations involving at least one new-operated variable. These equations define the discrete transition functions Δ of the OHM formalism. Examples of transition equations are:

```
NEW(x) := 1;
NEW(y) + NEW(x) + y = 0;
```

where x and y may be discrete or continuous variables. The first equation uses the binding operator to emphasize that x is assigned a new value as a result of the event. The binding operator is equivalent to the ordinary equation operator but it requires that the left side expression is a single new-operated variable. The second equation implicitly defines a new value for y as a function of the new value of x and the old value of y .

Equations without any new-operated variables are not allowed in an event. Such equations express constraints that are valid at all times, i.e., ordinary equations. They have no particular meaning within an event definition but they are allowed in a model anywhere outside the body of an event definition.

Note that when an event is fired, not only the equations defined in the event but all the equations of the model are considered in the state transition.

Another type of event actions are local variable definitions with bindings. Such definitions are motivated by the need for defining temporary variables for complicated expression, used mainly within the event definition. The value of a local variable is computed as a result of the event and may be used in other expressions in the event definition. A local variable may not be new-operated. In fact, all occurrences of the variable in the event definition are interpreted as representing the new value computed as a result of this event. Outside the event definition, a local variable is treated as any discrete variable and it maintains the value computed at the last occurrence of the event in which it is defined. The value of a local variable is undefined until the first occurrence of the event. An example of an event with a local variable is the following:

```
E ISAN Event WITH
  x TYPE Real := sin(y);
  NEW(z) := x*x;
END;
```

When the event fires, the local variable x is first computed. This value

is then used for computing the new value of z . In general, the order in which the equations and local variable definitions are defined in an event is not significant. Their meaning and computational order is determined by the occurrence of new-operators and local variables in the expressions. In fact, current version of Omola does not allow the model to specify the computational order explicitly. This may be a disadvantage when, for example, a computer algorithm must be represented as a model component. A solution to this problem could be to allow procedures coded in some ordinary programming language, like C or FORTRAN, to be included in models and executed as discrete event actions.

A third type of event actions are procedure calls. A common type of procedure call is the schedule command which instructs the simulator to fire a certain event at a certain time in the future. A schedule command looks like

```
schedule(EventName, Delay);
```

The Delay argument can be any real-valued expression which is evaluated when the event occurs. It defines the delay from the current time to the firing time of the specified event. A descheduling command used as

```
deschedule(EventName);
```

is useful in certain types of modeling. It removes the next occurrence of the specified event type from the scheduling queue.

Another type of procedure action available in OmSim is `printlog` which does not affect the model, but is used for logging data and for diagnostic messages in respond to events.

Discrete and continuous equations and bindings

Continuous and discrete variables can be mixed in ordinary model equations. If an equation contains at least one, non-constant, continuous variable, it is a continuous time equation. An equation with only discrete variables and constants is a discrete equation. Continuous and discrete equations have the same meaning in a model — they define constraints that must be true at all times. However, a discrete equation does not affect the continuous behavior directly, and can be disregarded by the continuous time simulation procedure.

Assignments to discrete variables are only defined if the assignment expression is discrete, i.e., contains no continuous variables. This is expressed in the following rule:

SEMANTIC RULES 11—Discrete variables and bindings

6.3 Event synchronization and propagation

1. A discrete variable changes its value only as a result of an event.
2. A binding expression for a discrete variable must not include any continuous variables.

□

For example, regard the following definitions:

```
x, y TYPE Real;  
d TYPE DISCRETE Real;  
x := d - 1.0; % correct  
d := 2.0 * y; % error
```

The third line is correct since a continuous variable can be bound to a discrete expression. The fourth line is an error since a discrete variable cannot be assigned with a continuous expression except as an event action.

6.3 Event synchronization and propagation

Events can be synchronized and propagated between submodels. Event synchronization is a way of defining interaction between submodels. Omola allows event interaction to be defined by terminals and connections in a way similar to continuous time interaction. Omola uses three basic synchronization mechanisms:

- Symmetric synchronization
- Directed synchronization, or propagation
- Conditional directed synchronization

The equation and assignment operators in basic Omola, (= and :=), are used for defining event synchronization. The connection operator (AT) is also interpreted as event synchronization when the connected terminals are derived from a set of predefined *event terminals*.

In order to simplify the presentation and exact definitions of the synchronization operators, some additional notation is introduced. Let an event definition be represented by the triple $E = \langle S, C, A \rangle$, where S is a set of unique symbolic names referring to the event type, C is the boolean event condition, and A is the list of action attributes. Let all event types of a model be totally ordered by the depth-first traversal of the instantiated model. If an event e_i comes before another event e_j , we say that e_i has *higher priority* than e_j . Let the *priority concatenation* of two action lists be the action list where the actions of the higher priority event are followed by the actions of the lower priority event with identical actions removed. The priority concatenation of the action lists A_1 and A_2 is denoted by

Chapter 6. Discrete Events in Omola

$A_1 \cdot A_2$. Note that the ordering of actions is only important for certain side-effect actions.

Symmetric synchronization is represented in a model using the equation operator with the left and right operands as event names. For example, with E1 and E2 as event names, they are synchronized by the “equation” attribute

E1 = E2;

The meaning of symmetric synchronization is that whenever one of the synchronized events fires, the other one fires as well, at the same time. The behavior of the synchronized events is not separable. A more precise definition of event synchronization is:

SEMANTIC RULE 12—Symmetric synchronization

Let $e_1 = \langle S_1, C_1, A_1 \rangle$ and $e_2 = \langle S_2, C_2, A_2 \rangle$ be two event types. The synchronization of e_1 and e_2 is the event $\langle S_1 \cup S_2, C_1 \vee C_2, A_1 \cdot A_2 \rangle$ which replaces e_1 and e_2 in the model. \square

In other words, when two events are synchronized, they are replaced by a new event which can be referred to by either of the names of the original events, its condition is the disjunction of the original conditions, and its action is the combination of the actions of the original events.

Directed synchronization is defined using the binding operator. For example, the assignment

E2 := E1;

defines the directed synchronization from E1 to E2. It means that the occurrence of event E1 also causes E2 but E2 may occur independently of E1. This type of synchronization is also called *event propagation*; event E1 is propagated to event E2, and it defines a causal relationship between two events. The exact meaning of directed synchronization is the following.

SEMANTIC RULE 13—Event propagation

Let $e_1 = \langle S_1, C_1, A_1 \rangle$ and $e_2 = \langle S_2, C_2, A_2 \rangle$ be two event types. A directed synchronization from e_1 to e_2 leaves the event e_2 unaffected but replaces e_1 with the event $\langle S_1, C_1, A_1 \cdot A_2 \rangle$. \square

In other words, with this type of synchronization, the actions of the propagated event are augmented by the actions of the event to which it is propagated. An example of a system using event propagation, is a sampled system with several subsystems and common clock, generating sampling events. The sampling events are propagated to every module to achieve synchronous sampling.

6.3 Event synchronization and propagation

A variant of event propagation is conditional directed synchronization. The difference from ordinary (unconditional) event propagation is that the propagation only occurs if a certain boolean expression is true. Conditional propagation is defined as

$$E2 := \text{condition AND } E1;$$

where *condition* is any boolean expression. As for unconditional propagation, only the definition of $E1$ is affected. The actions of $E1$ will conditionally be augmented by the actions of $E2$. Formally this is described as the follows.

SEMANTIC RULE 14—Conditional propagation

Let c be a boolean expression and let e_1 and e_2 be two event types with a conditional directed synchronization from e_1 to e_2 on condition c . When e_1 is enabled and about to fire, then c is evaluated. If c evaluates to false, then no synchronization takes place and e_1 is fired alone. If c evaluates to true, then e_1 is propagated to e_2 as if there was a directed synchronization from e_1 to e_2 . \square

Events with directed and symmetric synchronization operators have algebraic properties that can be used to simplify models and to develop a canonical representation used in the simulator. These properties will now be discussed.

DEFINITION 6.1

Two sets of event specifications (event definitions and synchronizations) are *behavior equivalent* if one set can replace the other without affecting the behavior (the set of all possible traces) of the model. \square

Behavior equivalence is denoted by the equivalence operator \Longleftrightarrow . For example, the behavior equivalence

$$\langle S_1, C_1, A \rangle, \langle S_2, C_2, A \rangle \Longleftrightarrow \langle S_1 \cup S_2, C_1 \vee C_2, A \rangle \quad (6.1)$$

states that two event definitions with identical actions are behavior equivalent to a single event definition with the union of symbolic names and a disjunction of the conditions. The validity of (6.1) can be verified against the OHM formalism. Another basic rule is that unique names, not occurring in any other events, can be added or removed from an event without changing the behavior of the model. This is written as

$$\langle S, C, A \rangle \Longleftrightarrow \langle S \cup S', C, A \rangle, \quad (6.2)$$

where S' is a set of unique symbols. Behavior equivalences can be used as rewriting rules defining model manipulations that are not changing

Chapter 6. Discrete Events in Omola

the behavior of the model. Such manipulations may be done in order to simplify the simulation.

The definitions of the directed and symmetric synchronization operators, given as Semantic Rule 12 and Semantic Rule 13 above, can also be defined as the following behavior equivalences where $e_1 = \langle S_1, C_1, A_1 \rangle$ and $e_2 = \langle S_2, C_2, A_2 \rangle$.

$$e_1, e_2, e_2 = e_1 \iff \langle S_1 \cup S_2, C_1 \vee C_2, A_1 \cdot A_2 \rangle \quad (6.3)$$

$$e_1, e_2, e_2 := e_1 \iff \langle S_1, C_1, A_1 \cdot A_2 \rangle, \langle S_2, C_2, A_2 \rangle \quad (6.4)$$

DEFINITION 6.2

An *empty event*, denoted e_\emptyset , is the event $\langle S, \text{false}, () \rangle$, i.e., the event with an arbitrary set of symbolic names, that never occurs, and has no actions. \square

An empty event can be added to or removed from a model without affecting its behavior. We can write this as

$$e_\emptyset \iff \emptyset \quad (6.5)$$

Provided that its symbolic names are unique, an empty event is a neutral element under synchronization, i.e., using (6.2) – (6.5) it is easy to show that for any event e_1 :

$$e_\emptyset = e_1 \iff e_1 = e_\emptyset \iff e_1 \quad (6.6)$$

$$e_\emptyset := e_1 \iff e_1 := e_\emptyset \iff e_1 \quad (6.7)$$

It can also be shown that symmetric synchronization is commutative and associative, and that directed synchronization is associative.

The following proposition states that if two events have directed synchronization in both directions, this is equivalent to a symmetric synchronization.

PROPOSITION 1

For any events e_1 and e_2 :

$$e_1 := e_2, e_2 := e_1 \iff e_1 = e_2 \quad (6.8)$$

\square

The proposition can be proved using the definition of directed synchronization (6.4) twice on the left hand side of the equivalence, and then reducing it by using (6.1) and (6.3).

6.3 Event synchronization and propagation

Active events

Events defined without any actions of their own can still be useful as links in an event propagation chain. Since event propagation may be conditional and depend on the current state, it may occur in some states that events without actions of their own are not affecting the model state at all. The following definition is useful:

DEFINITION 6.3

An event type is *active* in a particular model state if it has the potential to affect the model state. \square

The following rules determine if an event is currently active or not:

SEMANTIC RULES 15—Active events

1. An event $e = \langle S, C, A \rangle$ is always active (in any model state) if it has a non-empty action list A . Pure side-effect actions like `printlog` are not counted.
2. An event e_1 is active if there is a directed synchronization from e_1 to e_2 and if e_2 is active.
3. An event e_1 is active if there is a conditional directed synchronization from e_1 to e_2 and if the condition is true and e_2 is active. \square

The concept of active events is useful in models for specifying event propagation. Examples are given in Section 6.6. For this reason, a special event operator is available in Omola. The operator, called `ACTIVE`, operates on event names and returns a boolean value. As an example, regard the Omola definitions:

```
E1, E2, E3 ISAN Event;  
E1 := ACTIVE(E2) AND E3;
```

Event `E3` is propagated to event `E1` only if event `E2` is currently active.

Event terminals

One reason for having concepts for event synchronization and propagation is the possibility to have submodels that interact by means of events. An event caused by a condition in one submodel may propagate to another submodel causing its internal state to change. Synchronized events may cause the states of different submodels to be updated consistently and at the same time.

Chapter 6. Discrete Events in Omola

The concepts of directed and symmetric synchronization of events in different submodels are analogous to connections with and without a defined causality. For that reason, it is also natural to allow terminals that are events, rather than ordinary variables. Omola has three predefined event terminal classes:

```
EventTerminal ISAN Event;  
EventInput ISAN EventTerminal;  
EventOutput ISAN EventTerminal;
```

Within a model they all appear as ordinary event classes. In addition, they are regarded as terminals and are accepted by the connection operator AT. Since event terminals are regarded as terminals as well as events, they could have been derived from the predefined class Terminal as well (see Figure 4.5 in Chapter 4). This is an example where multiple inheritance, i.e., classes with more than one direct super class, would have been useful.

Two connected event terminals that are both derived from EventTerminal result in a symmetric synchronization. Connected combinations of event inputs, event outputs, and undirected event terminals result in directed synchronization. For example, if submodel S1 has an event input called Ei, and submodel S2 has an event output called Eo, then the following three definitions are all equivalent.

```
S1.Ei AT S2.Eo;  
S2.Eo AT S1.Ei;  
S1.Ei := S2.Eo;
```

6.4 A compact syntax for events

The generic way to define events is to define subclasses of the predefined class Event, as described above. In many cases, the name of the event is not relevant and not used anywhere else in the model. In this case, it is convenient to use an alternative syntax, specialized for event definitions, that does not require a name to be defined for the event. The syntax also includes constructions for defining event propagation and synchronization. The following constructions may be used for defining events in Omola.

```
WHEN <condition> [ CAUSE <event list> ]  
[ DO <actions> END ] ;
```

6.5 Discontinuities in continuous time equations

The initial keyword WHEN is followed by the event condition. There is an optional list of event names defining propagation to other events. Finally, there is an optional do-end block with the event actions. With this construction it is possible to define condition, propagation, and actions of an event in a compact way.

Event propagation, i.e., directed synchronization, can be defined in either direction. Propagation from the defined event to a named event is defined by including the name in the list after the key-word CAUSE. Propagation from named events can be defined using the OR operator and the event name in the condition expression. For example, regard the following definitions on generic form.

```
E1, E2 ISAN Event;
E ISAN Event WITH
  condition := c1;
...
END;
E2 := E;
E := c2 AND E1;
```

The equivalent definition in compact form, where previous event E is replaced by an anonymous event, is the following.

```
E1, E2 ISAN Event;
WHEN c1 OR (c2 AND E1) CAUSE E2
DO
...
END;
```

Symmetric synchronization cannot be defined directly between anonymous events.

6.5 Discontinuities in continuous time equations

The modeling language contains several functions and relations that depend on real valued arguments in a discontinuous way. A few examples are the round function that returns the closest integer to a real argument, and the comparison operators like '<' and '>' returning a Boolean result. When these functions appear in continuous models they may cause trouble for the numerical integration algorithm which has to resolve the discontinuities by taking very small steps. A numerically better way to handle discontinuities is to use a state event to detect the crossing of a discontinuity. In this case, the discontinuous function or operator should

be replaced by a discrete variable or a function that extends continuously across the original discontinuity. For example, if the original model contains a conditional expression like

```
y = IF x > 0 THEN a(x) ELSE b(x);
```

where x is a real variable and $a(x)$ and $b(x)$ are real functions of x . If $a(0) \neq b(0)$, the compound expression has a discontinuity at $x = 0$. An equivalent model using discrete events is:

```
xpos TYPE Boolean;
WHEN xpos AND NOT (x > 0) DO NEW(xpos) := FALSE; END;
WHEN NOT xpos AND (x > 0) DO NEW(xpos) := TRUE; END;
y = IF xpos THEN a(x) ELSE b(x);
```

The important difference is that the conditional expression now depends on the discrete variable `xpos` and thus, switches between the alternatives only as a result of an event. The original condition, $x > 0$, is no longer part of the continuous time model but appears instead as an event condition. The function $a(x)$ must be defined also for $x \leq 0$ and the function $b(x)$ must be defined also for $x \geq 0$ for the construction to work. Better behavior of the numerical integration can be expected when the two functions are continuous also at $x = 0$.

It is convenient for the model designer to use the former, much simpler construction, and get the more efficient event model generated automatically. This is done in Dymola [Elmqvist *et al.*, 1993], and it is planned to be implemented in OmSim as well.

A similar type of rewriting can be applied to discrete variables that depends directly on continuous time variables. For example:

```
i TYPE Integer := round(x);
```

The integer variable i is a discrete variable but its value depends on the real variable x . Since i can appear in some other continuous time expression, it causes the same problem for the numerical integrator as the previous example. A way of rewriting the discontinuous binding into an event model is given in Listing 6.2. Similar constructions can be introduced for every discontinuous or discrete function of real variables. An advantage of rewriting expressions in this way is that it decouples the discrete time part of the model from the continuous time part. This is a form of *tearing* [Elmqvist and Otter, 1994]. In short, tearing means that a system of equations is divided into two parts where the equations in one part can be solved easily if only the variables solved from the other part are known.

6.6 Examples of discrete and hybrid models

```
i TYPE DISCRETE Integer;
xlo, xhi TYPE DISCRETE Real;
WHEN x > xhi DO
  NEW(i) := round(x+0.5);
  NEW(xlo) := new(i) - 0.5;
  NEW(xhi) := new(i) + 0.5;
END;
WHEN x < xlo DO
  NEW(i) := round(x-0.5);
  NEW(xlo) := new(i) - 0.5;
  NEW(xhi) := new(i) + 0.5;
END;
```

Listing 6.2 An event model that is equivalent to a binding with the discontinuous function round.

6.6 Examples of discrete and hybrid models

The different formalisms for discrete event and hybrid models, presented in the previous chapter, are now used as examples and represented in Omola. The purpose is to demonstrate that the basic discrete event concepts and the structuring facilities of Omola are sufficient to represent a wide range of hybrid models.

Sampled data models

A sampled data model defined by the state-space equations:

$$\begin{aligned}x_{k+1} &= f(x_k, u_k, t_k) \\ y_k &= g(x_k, u_k, t_k),\end{aligned}\tag{6.9}$$

has only one type of event — the sampling event (see also Section 5.2). Therefore, all sampled data models have a common mechanism, the sampling clock, that generates the sampling events. It is natural to define a generic sampling clock as in Listing 6.3. The sampling period is constant and defined by a parameter that can be changed by the model user. The model defines an event called `Init`. This is a special event that is always fired by the simulator at the start of a new simulation. `Init` schedules the first sampling event and after that, each sampling event schedules the next one. `SampledModel` can be used as a base class for any specific sampled model. A generic definition of a sampled data model, according to (6.9), is given in Listing 6.4. The model is designed to operate in a continuous environment. The input is a continuous variable that is sampled

```

SampledModel ISA Model WITH
  Sample, Init ISAN Event;
  h ISA Parameter WITH default := 1.0; END;
  WHEN Sample OR Init DO
    schedule(Sample, h);
  END;
END;

```

Listing 6.3 Base class for sampled data models.

```

M ISA SampledModel WITH
  terminals:
    u ISA SimpleInput;
    y ISA DiscreteOutput;
  state:
    x TYPE DISCRETE Real;
  behavior:
    WHEN Sample DO
      NEW(x) := state update expression;
      NEW(y) := output expression;
    END;
END;

```

Listing 6.4 Generic Omola code for a sampled data model.

by the discrete model. The output is a discrete variable that works as a continuous zero-order hold reconstruction of the output sample.

It is also possible to represent structured sampled data models. For example, a simple digital filter can be structured as shown in Figure 6.1. Synchronous sampling is obtained by a common sample clock that generates the sample events and propagates them to each submodel. The sampler component has a continuous input signal and a discrete output signal. The reconstructor component has the reverse terminal configuration. The two filter components are purely discrete models, with discrete input and output signals. An example of a base class for a linear discrete model is shown in Listing 6.5. The base class can be specialized into a model with bindings specifying the model order and the system matrices. The behavior of the model is defined by the sample event containing two equations. Note the use of the new-operators. The new-operator is used for the input signal, since the input must propagate through the system without any time delay. At any sample occurring at time t_k , the model should compute y_k and x_{k+1} , depending on u_k and x_k , but u_k may be the output of another model, computed at this sample.

6.6 Examples of discrete and hybrid models

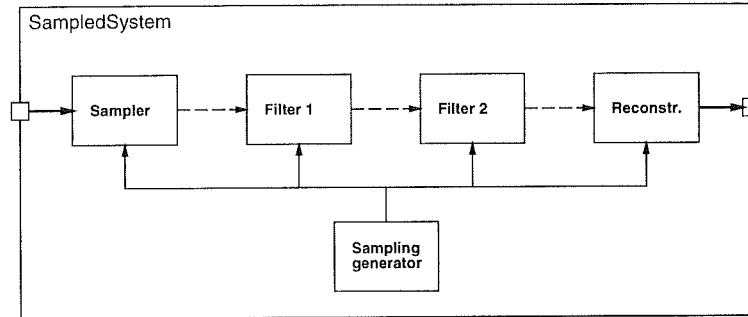


Figure 6.1 Example of a sampled data system composed of several discrete models. The fat arrows indicate continuous signal connections, the dashed arrows indicate discrete signal connections, and the thin arrows indicate event connections.

```

DiscreteLinear ISA Model WITH
terminals:
  Sample ISA InputEvent;
  u ISA DiscreteInput;
  y ISA DiscreteOutput;
parameters:
  N TYPE Integer; % model order
  A TYPE Matrix [N, N];
  B TYPE Column [N];
  C TYPE Row [N];
  D TYPE Real := 0.0;
state:
  x TYPE DISCRETE Column [N];
behavior:
  WHEN Sample DO
    NEW(x) := A*x + B*NEW(u);
    NEW(y) := C*x + D*NEW(u);
  END;
END;

```

Listing 6.5 A base class for a discrete linear model.

```
Server ISA Model WITH
  state TYPE DISCRETE (Passive, Active);
  Start, Ready, Init ISAN Event;
transitions:
  WHEN state == 'Passive AND Start DO
    NEW(state) := 'Active;
  END;
  WHEN state == 'Active AND Ready DO
    NEW(state) := 'Passive;
  END;
initialization:
  WHEN Init DO
    NEW(state) := 'Passive;
  END;
END;
```

Listing 6.6 An FSA server model in Omola.

Modeling of Finite-State Automata

In the following, it is shown how a deterministic finite-state automaton, according to the definition in the previous chapter, can be represented in Omola. It is also shown how composition of FSA:s is represented.

Each symbol in the input alphabet Σ of an FSA, corresponding to state transitions, is represented by a named event type. An anonymous event is defined for each combination of state and input symbol with a non-empty transition function δ . The state of the FSA can be represented by a discrete variable of integer or enumeration type. The idea is illustrated by an example in Listing 6.6 which shows an Omola model for the server FSA presented in Figure 5.3.

The suggested representation is not very compact for an FSA with many defined transitions. The number of event definitions may be as large as the number of input symbols times the number of states. It is possible to use a more compact representation if the events and states are coded by integers used for indexing a state transition table, represented by a constant matrix. Only one event for each input symbol has to be defined in this case.

An FSA is a non-autonomous model in the sense that all transitions are caused by events which are regarded as inputs to the model. The FSA model relies on some external mechanism inducing the transitions. An important observation is that there is no way an FSA model can prevent firing of an event. If an event is not enabled in current state, i.e., there is no defined transition for the event from current state, then firing is

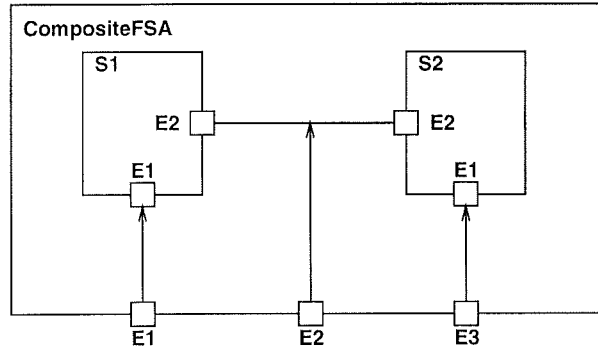


Figure 6.2 Full synchronous composition of two FSA submodels, involving the event E2

an empty operation. However, the environment of an FSA model can determine if an event is currently enabled using the active-operator. Only events for which active is true are currently accepted as input symbols. For that reason, an Omola FSA model is a true representation of an FSA.

A composite model may contain several FSA submodels. If there is no event synchronization between the FSA components, they are said to operate in parallel composition [Heymann, 1990]. There are other types of FSA compositions that involve elements of synchronization between the components. Two kinds of synchronizing compositions, introduced in [Heymann, 1990], will be discussed here. These kinds of model compositions are interesting because they are important in the design of controllers for discrete event dynamic systems [Ramadge and Wonham, 1989, Balemi, 1992].

One type of FSA composition is called *full synchronous composition*. In a full synchronous composition of two FSA:s, input symbols that belong to the alphabets of both FSA:s are accepted by the composite FSA only when they are accepted by both FSA:s separately. Full synchronous composition of Omola models is illustrated in Figure 6.2. The picture shows a composition of two FSA components, named S1 and S2. Each input symbol of an FSA is indicated by an input event terminal. According to the normal conventions of Omola, each component is defined by a separate class that constitutes a separate name space. Synchronizing events is the same as saying that an input symbol of one FSA is actually an alias name for an input symbol of another FSA. In the example, E1 of the composite model is an alias name for S1.E1 while E2 of the composite model is an alias name for S1.E2 and S2.E2 which are considered to be the same

```
CompositeFSA ISA Model WITH
events:
  E1, E2, E3 ISA EventInput;
components:
  S1 ISA ...
  S2 ISA ...
synch:
  S1.E1 := E1;
  WHEN E2 AND active(S1.E2) AND active(S2.E2)
    CAUSE S1.E2, S2.E2;
  S2.E1 := E3;
END;
```

Listing 6.7 Textual definition of the FSA composition in Figure 6.2

event. The composite FSA is a full synchronous composition of S1 and S2 if E2 is enabled if, and only if, S1.E2 is enabled and S2.E2 is enabled. This is accomplished by the conditional synchronization:

```
WHEN E2 AND active(S1.E2) AND active(S2.E2)
  CAUSE S1.E2, S2.E2;
```

In this way E2 is active and is propagated to the components only when both components are ready to accept the event. The textual definition of the composite FSA is given in Listing 6.7.

A more general kind of composition is called *prioritized synchronous* composition. In this case a subset of the input symbols of each component FSA is said to be prioritized by the component. A component with a prioritized symbol must agree and be enabled for the symbol for the synchronized event to occur. If synchronized events are prioritized by several components, then all these component must agree for the event to occur. This synchronization is strict, as in full synchronous composition. However, if an event is prioritized by one component only, then it is enabled in the composite FSA if, and only if, it is enabled in that component. The other components must accept the symbol, but fire the event only if they can. Further more, an event that is not prioritized by any of the components, is enabled by the composite FSA if, and only if, it is enabled in at least one of the components. Figure 6.3 shows an example with all four types of synchronizations. Component S1 has events E2 and E4 prioritized, while component S2 has events E2 and E5 prioritized. Listing 6.8 gives the corresponding Omola code.

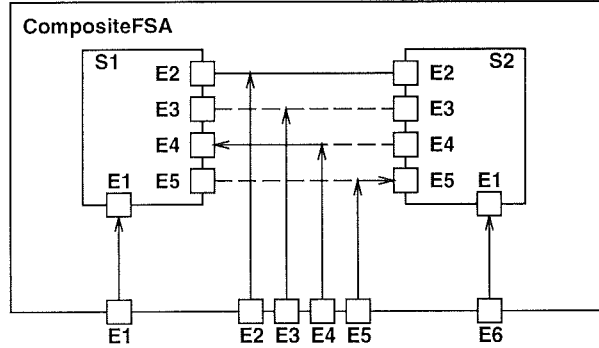


Figure 6.3 Graphical representation of prioritized Synchronous composition.

CompositeFSA ISA Model WITH

events:

E1, E2, E3, E4, E5, E6 ISA EventInput;

components:

S1 ISA ...

S2 ISA ...

sync:

S1.E1 := E1;

WHEN E2 AND active(S1.E2) AND active(S2.E2)

CAUSE S1.E2, S2.E2;

WHEN E3 CAUSE S1.E3, S2.E3;

WHEN E4 AND active(E1.E4) CAUSE S1.E4, S2.E4;

WHEN E5 AND active(E2.E5) CAUSE S1.E5, S2.E5;

S2.E1 := E6;

END;

Listing 6.8 Omola code for the prioritized Synchronous composition of Figure 6.3.

Modeling of GSMP

A GSMP (Generalized Semi-Markov Process) model can be represented as an Omola Hybrid Model. A generic GSMP model consists of a set of event types, a set of discrete-valued state variables, and a set of clocks, one for each event type. The clocks increase at a constant rate and they are restarted from zero when the event occurs. An event fires when its clock has reached the lifetime scheduled for the particular event type. Occurrence of an event results in a state change. Any OHM that has these


```

ServerWithQueueGSMP ISA Model WITH
  q TYPE DISCRETE Integer; % number of customers
  NewCust, ReadyCust, ServeCust, Init ISAN Event;
parameters:
  t1, t2, m, d ISA Parameter;
transitions:
  WHEN NewCust DO
    NEW(q) := q+1;
    schedule(NewCust, normal()*d+m);
  END;
  WHEN ReadyCust DO
    NEW(q) := q-1;
  END;
  WHEN Init DO
    schedule(NewCust, normal()*v+m);
  END;
  WHEN (q=0) AND NewCust OR (q>1) AND ReadyCust
    CAUSE ServeCust;
  WHEN ServeCust DO
    schedule(ReadyCust, rect()*(t1+t2/(q+1)));
  END;
END;

```

Listing 6.9 Example of an Omola GSMP model of the server with a queue.

properties is a GSMP according to the general definition given in [Glynn, 1989]. The following example shows a rather specific GSMP model.

Example: Server with queue (continued) This example is continued from Chapter 5. Listing 6.9 shows an Omola model of the server with the queue. The integer variable q represents the current number of customers in the system. The event `NewCust` represents the arrival of a new customer, the event `ReadyCust` represents completed service of a customer, and the event `Init` initializes the model. An additional event, `ServeCust`, occurs when the service of a customer is started. It is conditionally propagated from `NewCust` or `ReadyCust`. Parameters affect the distribution functions for the service time and for the time between arriving customers. \square

The GSMP formalism does not include any structuring or composition concepts. In practical modeling examples the modeled process often has some higher level structure which can be used to structure the model. For example, GSMP models are often used for queuing networks and manufacturing systems. In this case, natural structuring units are queues

and production cells. For this reason, the typical GSMP model is probably resulting from a set of model components, from some application oriented library.

DEVS modeling in Omola

The DEVS formalism was presented in Chapter 5. The formalism can represent structured discrete event models with a general state space of real-valued and discrete-values variables. A basic (atomic) DEVS model has an associated event type called the *internal* event. In addition it may have a set of input event types and a set of output event types. The input and output events are used for propagating internal events between different basic models in a multi-component model. The internal state of a basic model is changed as a result of either an input event or the internal event. The firing time of the internal event depends on the internal state. In other words, a basic model can schedule its own internal event.

When basic DEVS models are to be represented in Omola, it is natural to use a common base class defining the common properties. Listing 6.10 shows the definition of a base class for basic DEVS models. The global constant `inf` is used for representing the infinity which is the value of the time advance function when the internal event is not scheduled. Any real value which is not in the simulation time interval can be used for representing infinity. In this case we have chosen a negative value, assuming that all simulations start from time zero or later. Another possibility would be to use a very large value to represent infinity but this would give problems when quantities that occasionally assume the value of `inf` are displayed in a graph.

The base class `BasicDEVS` defines two variables: `next` which is discrete and represents the time of next scheduled internal event, or `inf` if the internal event is not scheduled, and `tr` which is continuous and represents the residual time function, i.e., the time that remains until next scheduled internal event. Two event types are defined in `BasicDEVS`: the internal event and the initial event. The class is intended to be used as a base class and specialized for particular DEVS models. A particular DEVS model defines additional state variables, input events, and output events.

Example: Simple server model (continued). This example continues the simple server model discussed in Chapter 5. An Omola definition of the DEVS model is given in Listing 6.11. The discrete state is represented by the symbolic variable `s`. The start event is represented as an input event and the ready event is represented as an output event. The internal event occurs when the current job terminates. It causes the output event,

Chapter 6. Discrete Events in Omola

```
inf TYPE Real := -1.0;

BasicDEVS ISA Model WITH
  next TYPE DISCRETE Real;
  tr TYPE Real := IF next == ::inf THEN ::inf
    ELSE next - Base::Time;

  InternalEvent, Init ISAN Event;
END;
```

Listing 6.10 The base class for basic DEVS models.

```
SimpleServer ISA BasicDEVS WITH
  s TYPE DISCRETE (Active, Passive);
  Start ISAN EventInput;
  Ready ISAN EventOutput;

  WHEN InternalEvent CAUSE Ready DO
    NEW(s) := 'Passive;
    NEW(next) := ::inf;
  END;
  WHEN Start AND s == 'Passive DO
    NEW(s) := 'Active;
    st TYPE Real := normal();
    NEW(next) := st + Base::Time;
    schedule(InternalEvent, st);
  END;
  WHEN Init DO
    NEW(next) := ::inf;
  END;
END;
```

Listing 6.11 Omola definition of the simple server DEVS model.

changes to passive mode, and sets the time of next internal event to infinity. The start event affects the system only if the mode is currently *Passive*. In this case, the mode is switched to *Active*, a normally distributed service time is generated, the time of next internal event is updated, and the internal event is scheduled. \square

Coupled DEVS models are represented in Omola using connections between input and output ports of basic and other coupled DEVS models. The formal definition of coupled DEVS models includes a *tie-breaking*

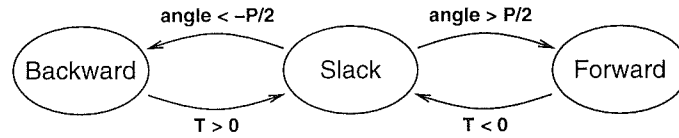


Figure 6.4 Discrete mode switches of the backlash model.

selector function. This function selects which one of the components in the coupled model should be allowed to fire its internal event when several components are scheduled at the same time. In Omola, the definition order of components determines in which order events scheduled at the same time should be fired.

6.7 A mode switching physical model

An Omola model of the backlash component in the rotational mechanics system discussed in Section 2.2 is now presented. The model is a good example of a physical, mainly continuous time system with switching modes. The backlash component has two rotating shaft interfaces. When the angle between the shafts is close to zero, they can rotate independently. If the angle difference reaches a certain positive or negative limit, the shafts get in contact and a positive or negative torque is transmitted. As discussed in Section 2.2, the model has three modes which are called *Slack*, *Forward* and *Backward*. The mode switches can be modeled by the FSM diagram in Figure 6.4. The transitions of the state diagram are labeled with logical transition conditions. These conditions depend on the continuous variables representing the angle difference between the shafts, and the transmitted torque T . The parameter P represents the maximum play angle.

The continuous time behavior equations (2.10) and the state diagram of Figure 6.4 can be translated into an Omola model in a straightforward way. The model is given in Listing 6.12. The model is defined as a subclass of `TwoCutModel` defined in Listing 3.2. Recall that a `TwoCutModel` has two terminals called C1 and C2 which have the components w representing the angular velocity and T representing the torque. Note the use of a conditional, mode dependent expression in the third equation. The mode switches are represented by four event definitions.

```
Play ISA TwoCutModel WITH
parameters:
  P ISA Parameter WITH default := 0.1; END;
variables:
  P2 TYPE Real := P/2.0;
  mode TYPE DISCRETE (Slack, Forw, Backw);
  angle TYPE Real;
equations:
  angle' = C1.w - C2.w;
  C1.T + C2.T = 0.0;
  0.0 = IF mode=='Slack THEN C1.T ELSE C1.w - C2.w;
events:
  WHEN mode=='Slack AND angle > P2 DO
    NEW(mode) := 'Forw;
  END;
  WHEN mode=='Slack AND angle < -P2 DO
    NEW(mode) := 'Backw;
  END;
  WHEN mode=='Forw AND C1.T < 0.0 DO
    NEW(mode) := 'Slack;
  END;
  WHEN mode=='Backw AND C1.T > 0.0 DO
    NEW(mode) := 'Slack;
  END;
END;
```

Listing 6.12 Omola model of the rotational mechanics backlash component.

6.8 Grafcet in Omola

This section describes how Grafcet specifications can be represented in Omola. The Grafcet formalism for representing sequential systems was discussed in Chapter 5. It is desirable to have an Omola library of basic Grafcet elements, which can be composed into complete Grafcet models according to the standard composition rules. The components must be defined so that a simulation of a grafcet represented in Omola behaves exactly according to the standardized Grafcet interpretation rules. Unfortunately, the standardized interpretation rules according to [IEC, 1988], are not completely unambiguous for all correct grafkets. An unambiguous interpretation algorithm is given in [David and Alla, 1992].

The basic elements of a grafcet are step, transition, parallel split, and synchronization. These elements are connected by arcs into a complete

```

StepLowerTerm ISA RecordTerminal WITH
  State ISA BooleanOutput;
  Fire ISAN EventInput;
END;

TransUpperTerm ISA RecordTerminal WITH
  State ISA BooleanInput WITH default := true END;
  Fire ISAN EventOutput;
END;

TransLowerTerm ISAN Event Output;

StepUpperTerm ISAN EventInput;

```

Listing 6.13 Terminal classes for step and transition interaction.

grafcet according to the syntactic rules. A starting point when a new model library is developed, is to analyze the interaction between the components and then define the necessary terminal classes. In a grafcet, the state of the model is maintained in the steps, while the decisions to change the state are made in the transitions. The decision to fire a transition is based on the condition associated with it, and on the state of the step above. Firing a transition is a discrete event and the state of a step is represented by a Boolean variable. The interaction between a transition and the steps above and below is illustrated in Figure 6.5. The necessary terminal classes are given in Listing 6.13. The different structures of the terminals, and the defined causalities, prevent steps and transitions to be connected in a way that is violating the syntactic rules of Grafcet.

The basic Grafcet elements *step* and *transition* can now be defined. The first attempt is shown in Listing 6.14. The transition defines a firing event condition and propagates the event through the terminals connected to the steps above and below. The firing condition requires that the step above is active and that the boolean condition is true. The step defines event actions for fire events from the transitions above and below. The proper actions are to switch to the active state if a fire event is coming from the transition above and to switch to the passive state if a fire event is coming from the transition below.

The suggested model for the step shows the general idea but in some cases it does not behave according to the firing rules of Grafcet. One firing rule specifies that simultaneously fireable transitions must be fired simultaneously. A grafcet in Omola will always fire the transitions in sequence according to the interpretation of Omola models discussed

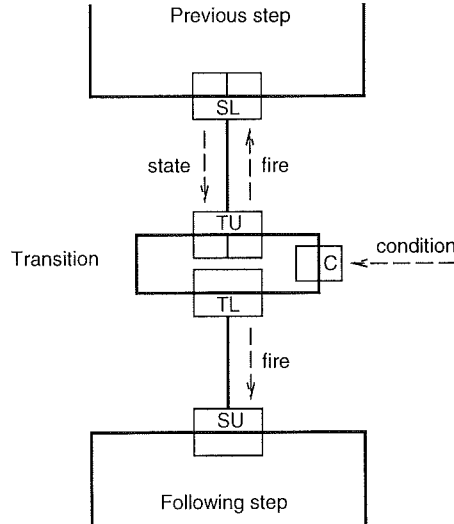


Figure 6.5 Illustration of the interaction in a grafcet. The terminals marked 'SL', 'TU', 'TL', and 'SU' are derived from the terminal classes StepLowerTerm, TransUpperTerm, TransLowerTerm, and StepUpperTerm respectively.

```

Transition ISA Model WITH
  U ISA TransUpperTerm;
  L ISA TransLowerTerm;
  C ISA BooleanInput;

  WHEN TU.State AND C CAUSE TU.Fire, TL.Fire;
END;

Step ISA Model WITH
  U ISA StepUpperTerm;
  L ISA StepLowerTerm;
  State ISA BooleanOutput;

  WHEN U.Fire DO NEW(State) := true; END;
  WHEN L.Fire DO NEW(State) := false; END;
END;

```

Listing 6.14 Simple Omola definitions for the basic Grafcet elements

```

Step ISA Model WITH
  U ISA StepUpperTerm;
  L ISA StepLowerTerm;
  State ISA BooleanOutput;

  Update ISAN Event;
  activate, deactivate TYPE Boolean;

  WHEN U.Fire DO
    NEW(activate) := true;
    schedule(Update,0.0);
  END;
  WHEN L.Fire DO
    NEW(deactivate) := true;
    schedule(Update,0.0);
  END;
  WHEN Update DO
    new(State) := activate OR State AND NOT deactivate;
    new(activate) := false;
    new(deactivate) := false;
  END;
END;

```

Listing 6.15 A definition of step that behaves correctly for simultaneous transitions.

in Section 5.10. Simultaneously or sequential firing matters only if the firing of one transition can affect the firing of another simultaneously enabled transition. This may be the case if the grafcet is such that some transition conditions directly depend on the current state of a step. In this case, the evolution of the Omola model may depend on the ordering of simultaneously fireable transitions, which is against the firing rules of Grafcet.

A more elaborate definition of the step model solves the problem with simultaneous firings. The new definition, shown in Listing 6.15, also handles correctly the Grafcet interpretation rule stating that a step that must be simultaneously activated and deactivated remains active. Instead of having the transition events updating the states of the steps directly, additional events called Update are scheduled with a zero time delays for each step that is affected by transition firings. All simultaneous firings are scheduled before the actual states are updated, thus assuring that updating one step does not prevent simultaneous firing of another.


```
Split2 ISA Model WITH
  U ISA StepUpperTerm;
  L1, L2 ISA TransLowerTerm;

  WHEN U.Fire CAUSE L1.Fire, L2.Fire;
END;

Synch2 ISA Model WITH
  U1, U2 ISA TransUpperTerm;
  L ISA StepLowerTerm;

  WHEN L.Fire CAUSE U1.Fire, U2.Fire;
  L.State := U1.State AND U2.State;
END;
```

Listing 6.16 Definitions of Omola models for parallel split and parallel synchronization with two output and input branches, respectively.

Parallel split and synchronization

A parallel split must always be preceded by a transition and followed by a number of steps, one for each parallel branch, according to the syntactic rules of Grafcet. For that reason, a parallel split represented in Omola has a single terminal that is a `StepUpperTerm` and any number of terminals that are `TransLowerTerms`. The model only has to propagate the firing events downwards from the preceding transition to each one of the following steps. Listing 6.16 shows a definition of a parallel split with two output branches.

A parallel synchronization has a terminal configuration that is the opposite of the parallel split. In addition to propagating events, the synchronization model must propagate the conjunction of the states of all preceding steps. The following transition must be enabled only if all the preceding steps are enabled, according to the rules of Grafcet. The definition is shown in Listing 6.16.

Omola has currently no way of defining models with a varying number of terminals. Therefore, separate model definitions have to be made for parallel splits and synchronization with different numbers of output branches and input branches. However, this is not a problem since it is possible to use components with superfluous terminals that are left unconnected. Proper behavior with unconnected terminals is accomplished by properly defined default values.

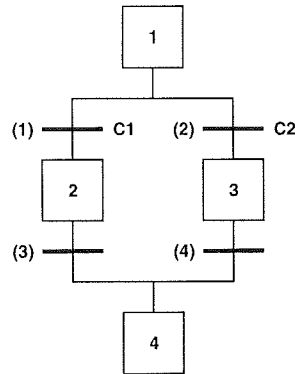


Figure 6.6 A grafcet with alternative paths.

Selection

The Grafcet syntax allows alternative paths, called *selections*, as illustrated in Figure 6.6. If Step 1 is active then Transitions (1) and Transition (2) are both enabled. If condition C1 is true then Transition (1) will fire and Step 2 becomes active. Alternatively, if condition C2 is true then Transition (2) will fire and Step 3 becomes active. The conditions C1 and C2 should be mutually exclusive, so that only one of the alternative paths are selected. If C1 and C2 become true simultaneously, there will be a *conflict situation* in the grafcet and the behavior is not well-defined. The alternative paths end with a *selection convergence* as shown in the figure.

No special arrangements are necessary to represent selections and convergence in Omola grafcets. The lower terminal of a step is simply connected to each one of the alternative transitions. The step will receive a fire event from either one of the connected transitions. The same applies to the convergence where the lower terminals of the last transitions of each branch are connected to the same upper terminal of the following step.

Conditions and actions

Interaction between a process model, which is often mainly a continuous time model, and a grafcet is defined in the conditions and the actions. Conditions can be any type of logical condition based on process measurements and the state of this or other grafcets. The actions can affect the process in many different ways. The control output from a step can be almost any kind of logic sequence, including for example delays and hold functions, activated by the step. Since the conditions and the actions are

```
MaxTempTransition ISA Transition WITH
  Temp ISA TemperatureInput;
  Tmax ISA Parameter WITH default := 100.0; END;
  C = Temp > Tmax;
END;
```

Listing 6.17 Definition of a transition with a temperature measurement as input. Its condition becomes fulfilled when the temperature exceeds the limit defined by a parameter.

normally special for each step and each transition, there is not much point in having steps and transitions with predefined actions and conditions.

There are basically two ways of defining the interface between a process and a grafcet. One possibility is to define each step and transition as separate classes, derived from the library definitions Step and Transition. The derived classes extends the definitions by adding the necessary interface terminals and equations. For example, the tank heater system described in Section 5.7 has a transition with a condition based on a temperature measurement. Listing 6.17 shows how such a transition can be defined. Steps can be extended with actions in a similar way.

Another way to structure the interface between the grafcet and the process, is to define all conditions and actions as separate components, connected to their associated steps and transitions by terminal connections. This adheres to the standardized way of drawing actions as separate boxes in connection with each step. Each action box or condition box defined in Omola has a standardized interface towards the step and the transition, but a specially designed interface towards the process model.

6.9 Summary

In this chapter we have enhanced the modeling capabilities of Omola to represent discrete event and hybrid systems. The concepts are based on the Omola Hybrid Model (OHM) formalism, and they make it possible to define discrete event behavior on a fairly low level. Events are defined as ordinary class components but a more compact, special syntax can also be used. Event conditions are defined as boolean expressions. A special operator is introduced to specify transient conditions. Special discrete commands for scheduling and descheduling events are also introduced.

The concepts of event synchronization and propagation are introduced and given a precise meaning. They make it possible to structure models containing discrete events according to the same principles as ordinary continuous time models.

7

Modeling and Simulation Environment

An environment for construction and simulation of Omola models has been implemented. The environment is called OmSim and it is a collection of tools for defining, editing, displaying, analyzing, and simulating Omola models.

OmSim is programmed in C++ and runs under Unix and X-Windows. It uses InterViews [Linton *et al.*, 1987] for the graphical user interaction. It also uses a set of various numerical packages for numerical integration and equation solving.

This chapter gives a general overview of the architecture of OmSim. Detailed descriptions are given for some parts of more fundamental importance. This chapter also defines some of the fundamental data structures used for representing models. Finally, a more general discussion is included on the architecture of OmSim in relation to a suggested reference model for open CACSD architectures.

7.1 Introduction

OmSim consists of two main parts: the modeling environment and the simulation environment. The basic structure of OmSim is shown in Figure 7.1. The modeling environment and the simulation environment are interactive tools. The modeling environment can read models, defined as Omola classes, from text files and store them internally as data structures. The user can then display the models, edit them, and include them as components in new models. A complete model definition can be used to generate a model instance which is sent to the simulation environment.

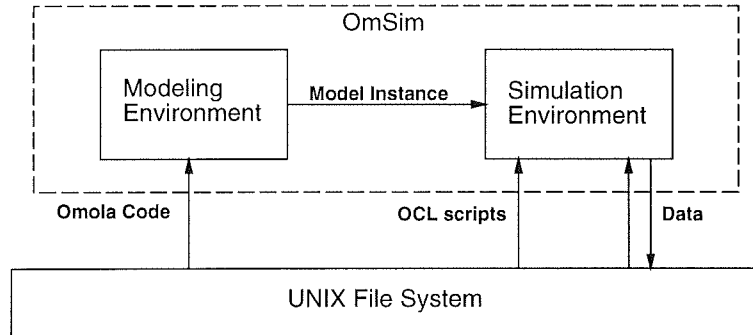


Figure 7.1 A top-view at OmSim

The simulation environment extracts necessary information from a model instance and manipulates it into a form suitable for numerical solution. The simulation environment allows the user to specify model parameters and problem specific attributes of the model, and to perform various simulation experiments. In addition to being controlled directly by the user, the simulation environment can also execute scripts written in a special command language called OCL (OmSim Command Language).

The main data structures manipulated by the modeling environment are Omola class definitions, while the main data structures in the simulation environment are the model instances. A model instance is a compilation of a set of class definitions. In Chapter 4, the meaning of Omola code was defined in terms of the created instances. A model instance is a representation of an Omola Hybrid Model (OHM) and additional information about the hierarchical structure. The OHM equations obtained from the model instance cannot be used directly for simulation. They have to be analyzed and checked to ensure that they make up a well-posed problem, and they have to be manipulated into a form that is suitable for the numerical simulation algorithms. Finally, before a simulation problem can be solved, executable simulation code must be generated from the manipulated OHM. The process is called *model compilation* and the main steps are illustrated in Figure 7.2. This chapter is mainly concerned with the architecture of the environment and of the different object representations. Chapter 8 discusses the algebraic model manipulations in detail.

OmSim has an interactive, graphical user interface. It is an important part of OmSim and necessary in order to fully support the object-oriented modeling methodology. The user interface is outside the scope of this thesis so it is only briefly discussed in this chapter.

7.2 Modeling environment overview

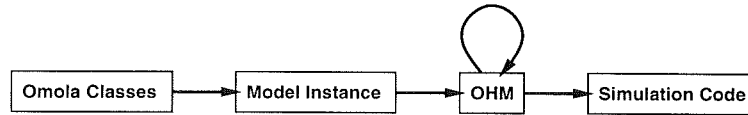


Figure 7.2 Data structure transformations from Omola classes into simulation code. The loop at the OHM box indicates successive steps of transformations in order to derive a suitable and efficient simulation model.

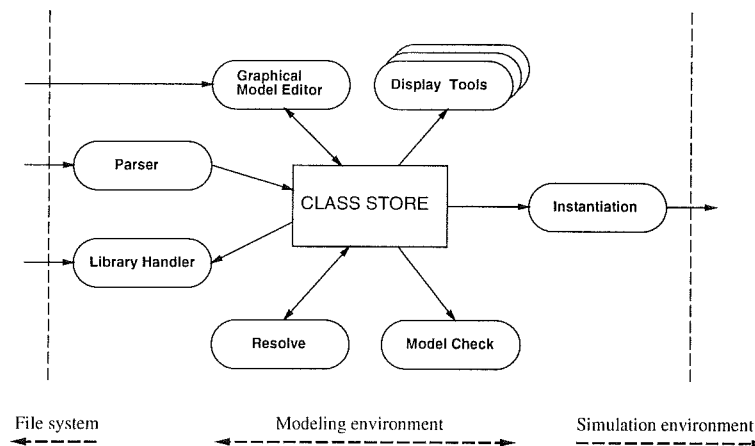


Figure 7.3 The basic structure of the modeling environment of OmSim

7.2 Modeling environment overview

The modeling environment of OmSim is pictured in Figure 7.3. The environment consists of a set of more or less independent tools operating on data structures maintained in the *class store*. The modeling environment has interfaces to the file system, where Omola class definitions are stored permanently, and to the simulation environment which is using the model instances. The tool components are normally controlled interactively by the user but they can also be activated on demand from the simulation environment. The tool components indicated in the Figure 7.3 are the following:

- A *parser* that reads Omola files, checks the syntax, and produces data structures in the class store.
- A *graphical model editor* which allows the user to define new classes representing structured models composed of submodels, terminals,

and connections.

- Various ways to *display* classes and model structures as textual presentations and tree diagrams.
- A class *instantiator* for producing model instances needed by the simulator.
- A *model check* tool for checking that symbolic references are valid.
- A *resolve* unit that processes class definitions and resolves symbolic super class references.
- A *library handler* that is capable of searching for class definitions externally in the file system and load them into the class store automatically.

The library handler has an interactive tool called the *Library Browser* where the user can find and select named Omola definitions. External class definitions in files, as well as internal, already loaded definitions can be selected in the browser. An external definition is loaded when it is selected for the first time. The browser currently implemented in OmSim is simple and displays only the contents of libraries as lists of names. It is clearly desirable to have much more advanced browsing facilities. For example, it should be possible to search for models with specific properties. The browser tool should also be connected to the display tools, so they would be automatically updated with the selected class. Advanced browsers for object-oriented environments were first introduced for the Smalltalk programming language [Goldberg, 1983, Goldberg, 1984].

Figure 3.2 shows the graphical model editor. A model can have an *internal view* and an *external view*. It is the internal view that is defined and edited by the model editor of OmSim. The external view is used when the model appears as a component of some other model. By default the external view is simply an annotated box, but it can also be a bitmap created and edited by a general bitmap editor external to OmSim. The graphical views of an Omola class are represented by ordinary class attributes. The graphical attributes are derived from special predefined classes, recognized by the graphical editor. The model instantiation tool also recognizes the graphical attributes and ignores them, since they are not needed in the model instance.

An advantage of including model view attributes as ordinary model attributes is that no special language has to be introduced. On the other hand, it may be a drawback that a new model version has to be defined if a different graphical view is wanted.

Examples of display tools are shown in Figure 3.5 and Figure 3.6. They present inheritance structure and component structure as tree dia-

7.3 Representation of classes

Table 3 Relations between entities in different domains of object-oriented modeling. Simple arrows indicate instantiation while double arrows indicate representation.

<i>Domain</i>	<i>Description</i>	<i>Instance</i>
Real world:	physical concept	physical object
	↓	↓
Model:	Omola class	model instance
	↓	↓
Abstract repr:	C++ class <i>Class</i>	C++ class <i>ClassInst</i>
	↓	↓
Concrete repr:	<i>Class</i> object	<i>ClassInst</i> object

grams. It is of course also possible to display a definition as Omola code. The displayed Omola code can be filtered so that attributes concerning the graphical representation are removed.

The following sections describe the representation of classes and model instances in more detail.

7.3 Representation of classes

The terminology used in the following may seem obfuscated because the discussion concerns objects, classes, and representations from several different levels and domains. In the modeling domain there are Omola models and model instances. In the implementation domain there are C++ classes and objects used for representing model domain entities. The concepts of *class* and *instance* occur in both domains. The levels and the concepts, and their relations, are illustrated in Table 3. The main focus of this chapter is the abstract representation level. This section discusses the representation of Omola classes. The following section is concerned with the representation of model instances.

When an Omola class is loaded from a text file into the internal memory, the parser checks the syntax and constructs an object representing the definition. The *object diagram* in Figure 7.4 shows the internal representation of Omola classes. The object diagram is made in the style of Rumbaugh et al. [Rumbaugh *et al.*, 1991]. It displays graphically the most important C++ classes used for representing Omola classes in the class store. Classes are represented by boxes annotated with class names. Some boxes also show some important attributes of the class. Lines between boxes represent *associations* between classes and between instances. Filled cir-

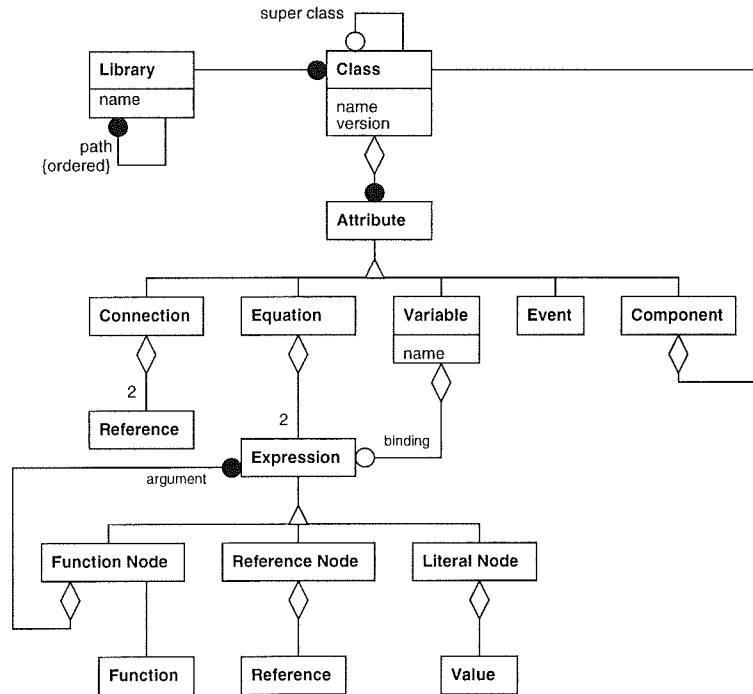


Figure 7.4 Object model of the internal representation of Omola classes.

cles indicate a multiplicity of zero or more, while white circles indicate a multiplicity of zero or one. In some case the multiplicity is given as a number. For example, the association between *Library* and *Class* means that a library may be associated with many classes but a class is always associated with exactly one library. A diamond at one end of an association indicates aggregation. This is a special kind of association where one object is considered to be subordinate to an aggregate object. For example, a *Class* object is an aggregate of zero or many *Attribute* objects, and an *Equation* object is an aggregate of two *Expression* objects. An association marked with a triangle indicates *generalization* and it is an association between classes and not between objects. Generalization is the same as a subclass – super class relation. For example, *Attribute* generalizes the different Omola class attributes *Connection*, *Equation*, *Variable*, *Event*, and *Component*.

The most important entity in the representation of Omola is defined by the class *Class* in the object model in Figure 7.4. Important attributes

7.3 Representation of classes

of *Class* are *name* and *version*. A *Class* object may be composed of many *Attribute* objects representing connections, equations, variables, events, and components, which are other *Class* objects. An *Equation* object is composed of two *Expression* objects for the left and right hand sides. A *Variable* object has a name and possibly an *Expression* object which represents a binding. An *Expression* object is a tree of nodes representing functions with arguments, references to variables, and literal data. A *Reference* object is a symbolic reference to a variable or a component in the Omola structure. A reference is *resolvable* (valid) if it refers to a variable or a component according to the scope rules of Omola.

Libraries and class versions

A library is a collection of *global* Omola classes. A global Omola class is a class that is defined at the top-level of a file, i.e., not as a component of another class. Every global class belongs to a library. Global classes can be searched by name in libraries.

When a new class is added to a library which already has a global class with the same name, the new class is considered as a new version of the existing class. When a class is searched by name it is always the latest version that is obtained from the library.

A reason for keeping old versions of classes is that resolved classes which refer to a class that is updated with a new version, are still valid, even if they are not up-to-date. They can be updated at a later occasion. Another reason is of course to make it possible to retain old versions and to compare models of different versions. However, a proper versioning system for models must be based on permanent storage. This will be discussed further on.

A library is also associated with a list of other libraries used as search path for localizing super classes. The search path is defined by a special USES statement in an Omola file. For details see [Andersson, 1993]. The use of the search path is explained below.

Super class resolve

A *Class* object may be in two different modes: resolved and unresolved. When the object is first created, it is unresolved which means that super classes are referred to by symbolic references. The class has a resolve operation which resolves the super class reference and replaces it by a hard link to the actual super class object. The resolve procedure involves searching for names in libraries. The reason for not resolving super classes immediately when a class is loaded, is to have the flexibility to load a set of classes in any order, not necessarily so that super classes are loaded

first.

A symbolic reference to a super class consists of a class name and possibly a library name. If no library name is given and if a class with the given name is not found in the current library, it is searched in the libraries specified by the search path of the current library.

A library search path makes it possible to write models with generic super class names. The actual super class found during resolve will depend on the library into which the model is loaded, and the search path of that library.

Internal and external class storage

Omola classes are stored externally as Omola code in files or internally in the OmSim class store as objects. In a more general framework, the external storage may consist of a general engineering database. A general framework for computer aided control engineering including the use of databases is discussed in Section 7.6. A database for storing Omola class definitions may provide several important services in addition to providing permanent storage. Current version of OmSim considers only the file system as external storage. This section will discuss the facilities in OmSim for loading and storing class definitions on files.

An important issue in the communication between internal storage and an external file system or database, is the granularity of the communicated data, i.e., the smallest entity which can be loaded into OmSim in one operation. Consider three possibilities which are natural in the case of OmSim:

1. libraries,
2. classes, and
3. class attributes.

Loading a library means that a file containing a set of class definitions is parsed and brought into the class store. This is efficient if a major part of the library is needed in the environment. However, a library for a particular application may be large and may contain many definitions not needed to solve a particular task, and an application may need definitions from several large libraries. In this case it is not practical to load all definitions into the internal memory. A class-based loading policy, where only the definitions actually needed from each library are loaded, is more efficient. With an even finer loading granularity it is possible to load classes with only some of their attributes. It depends on the modeling methodology if this is efficient. For example, a model may include alternative behavior descriptions of different complexity represented as separate class attributes. Then it may be efficient to load only those attributes

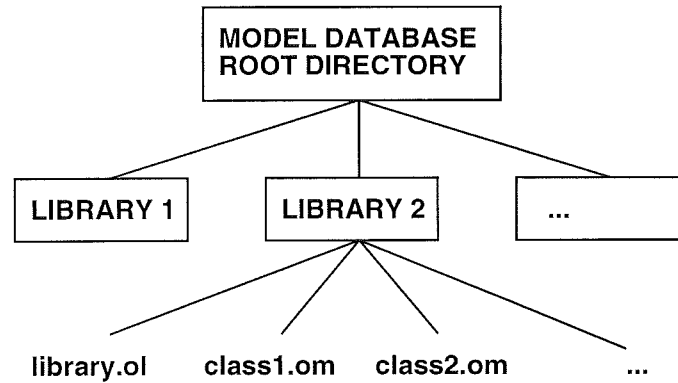


Figure 7.5 A model database directory tree

needed for solving the particular task. On the other hand, if models are minimal in the sense that all attributes are needed for most tasks, then this granularity is inefficient. This is the case for the modeling examples given in this thesis, so attribute-based loading is not considered in the following.

Current version of OmSim supports file-based loading directly controlled by the user, and automatic class-based loading. An Omola file may consist of any number of class definitions, and can make up a complete library or a part of a library. With file-based loading it is the user's responsibility to assure that the necessary files are loaded in a proper order. After a file has been loaded, the new class definitions are automatically super class resolved. If a super class is not found in the internal class store, an error occurs.

In addition to file-based loading controlled by the user it is possible to specify a *database search path*, which OmSim will use to localize and load missing class definitions from external files automatically. This requires that libraries and definitions are organized in a certain way shown in Figure 7.5. A *model database* is a directory containing a number of library directories. A library directory contains the class definitions; each one in a separate file with the same name as the class it defines. A library directory may also contain a special library header file, always named `library.ol`, which contains library specific documentation, global variable definitions, library path, etc. The database search path is a list of model database root directories used by OmSim to localize class definitions when they are needed for the first time.

The automatic, demand-directed loading of class definitions works by

a recursive definition of the procedure for loading files. The procedure *LoadFile* parses the file and then calls the procedure *ResolveSuperClass* for each new class definition. *ResolveSuperClass* tries to find the referenced super class by first searching among loaded classes in libraries specified according to current library search path. If the super class is not found in the internal class store, searching continues externally in the first model database in the model database search path, using the same library search strategy as before. If a file containing the super class definition is found, that file is loaded by a recursive call to *LoadFile*.

The arrangement of a model database search path offers the basic support for *concurrent engineering*, i.e., when a group of engineers are working on the same project. A version handling system, like the Unix RCS and CVS [Tichy, 1985], can then be used for storing and retrieving multiple revisions of models in the database. The individual users have database paths with their private database followed by the database common to the project. A user can check out a model version from the common database, experiment with different modifications, and then update the common version. The database path of OmSim will assure that the user's private version will be used if it exists, otherwise the public version will be used.

7.4 Representation of model instances

An important step in the derivation of simulation code from an Omola class is the creation of a *model instance*. A model instance is represented by a data structure consisting of a hierarchy of *class instances*. Every component and variable of a model instance is represented by a unique object. This is different from the *Class* object where several components and variables may share a common description. Another difference between the class object structure and the class instance structure is that the inheritance structure is not maintained in the instances. Only the composition structure is kept intact.

A model instance serves as an interface between the model representation and tools like the OmSim simulator. It provides a hierarchical storage for the variables, the equations, and the events that make up the mathematical description of a model.

Model instantiation was discussed briefly in Chapter 4. In this section we will focus on the representation of model instances as *ClassInst* objects, and on the instantiation procedure, i.e., the algorithm for creating *ClassInst* objects from *Class* objects.

The class instance object structure

Figure 7.6 shows the relations between the objects in the representation of instantiated Omola classes. The most important classes in the diagram are *ClassInst* for representing instantiated *Class* objects, *EquInst* for representing instantiated equations and connections, and *VarInst* for representing instantiated variables. Some of the classes in Figure 7.6, like *Class*, *Equation*, and *Variable*, are identical to those displayed in Figure 7.4.

The same basic classes are used for representing an instantiated expressions as are used at the class level in Figure 7.4. An expression is a tree where the nodes are functions and the branches are function arguments. The difference between an instantiated expression and a class level expression is in the representation of variables. At the class level, a variable in an expression is represented by a reference node, which is a symbolic reference to a variable attribute. The association to *Variable* object is only implicit in the name and in the search procedure used for resolving symbolic references. In an instantiated expression, each reference node is replaced by a *VarInst Node* object, which has a direct association to a *VarInst* object.

The instantiation procedure

The instantiation procedure is implemented as a method of *Class*. It creates a structured *ClassInst* object and builds it up in several passes. The passes of the instantiation procedure are the following.

1. Traverse the *Class* composition tree, visit each component attribute and build a corresponding composition tree of *ClassInst* objects.
2. Instantiate all *Variable* attributes and add the variable instances to the new tree.
3. Visit all non-array variables and instantiate their binding expressions.
4. Evaluate the dimensions of all array variables.
5. Instantiate binding expressions to array variables.
6. Instantiate all equations.
7. Instantiate connections into equations.
8. Instantiate events.
9. Do various model specific post processing.

All passes except the first one, involve a recursive traversal of the *Class* tree and the *ClassInst* tree in parallel. Data is extracted from the first tree and used for decorating the second tree. All passes except the last

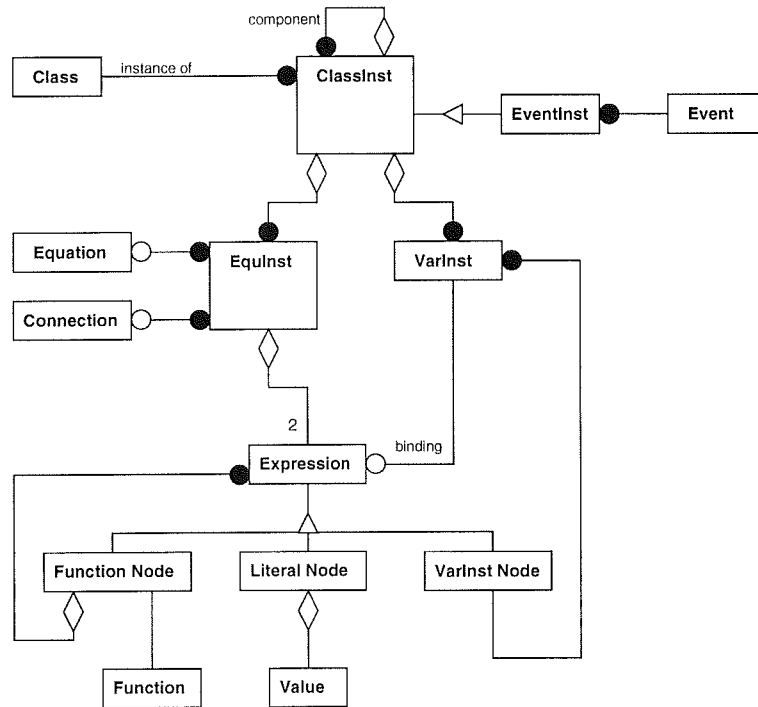


Figure 7.6 Object model for instantiated Omola classes.

one are universal in the sense that they do not discriminate between different Omola classes, i.e., models, terminals, parameters, etc. are all treated equally. In the last pass, however, different classes are treated differently.

The reason why bindings to scalar variables and to array variables are instantiated in different passes is that the size of array variables may depend on scalar variable bindings (see Listing 4.5 for an example). A binding expression must be instantiated before it can be evaluated. Array variables are instantiated with undetermined sizes in Pass 2, since they may depend on constant variables that cannot be evaluated at this stage. In Pass 3, the bindings of all scalar variables are instantiated, so that they can be used for determining the size of the array variables in Pass 4. Finally, in Pass 5, the bindings to the array variables are instantiated. Models may have complicated dependencies between array sizes and scalar bindings that cannot be resolved with this instantiation procedure. One possibility to handle more complicated cases would be to

7.4 Representation of model instances

iterate over Passes 3, 4, and 5 until all array sizes are determined.

The class-specific instance processing in Pass 9 recognizes the basic set of predefined Omola classes discussed in Section 4.8 and defined in Appendix B. It makes up the *model representation layer* in the layered view of OmSim in Figure 4.1. It has been a design goal to have as little class specific processing as possible, and to localize these computations to as few modules as possible. This makes it easier to modify assumptions about basic model representation in Omola.

The following classes, and all user-defined and predefined classes derived from them, receive special treatment at the final instantiation pass.

Variable The value attribute of non-constant variables will receive an explicit pointer to the initial attribute of the same class instance. The pointer is used during simulation to obtain a model-defined initial value for the variable.

Parameter The value attribute of non-constant parameters will receive an explicit pointer to the default attribute of the same class instance. The pointer is used during simulation to obtain a model-defined default value for the parameter.

BasicTerminal If the variability attribute of the terminal is set to Parameter the *value* attribute instance is marked as being a parameter. If the terminal is not connected on the outside of the model, and if the value attribute does not have a binding, then the binding to the default attribute will be used as a binding to the value attribute.

Class instance operations

A *ClassInst* object represents a model instance that provides all the necessary data needed by the simulator. The interface between the simulation tool and model representation is defined by a set of access methods of *ClassInst*. The most important methods are declared in C++ and described in the following. The methods are used for extracting the variables and the equations of a model for further manipulation by the modules in the simulation environment. The returned data types for several of the methods are written as 'LIST(*type*)'. This is a preprocessor macro defining that the returned object is a list of elements of the specified type.

```
LIST(VarInst) GetLocalVariables() const;
```

Returns all variable instances included in this class instance.
Variables declared *static* are not included.

Chapter 7. Modeling and Simulation Environment

LIST(ClassInst) GetLocalComponents() const;

Returns all class instances included as components of this class instance.

LIST(EventInst) GetLocalEvents() const;

Returns all event instances included in this class instance.

LIST(EquInst) GetEquations() const;

Traverse the composition hierarchy of *ClassInst* objects, beginning from this class instance, and return all *EquInst* objects.

LIST(VarInst) GetVariables() const;

Traverse the composition hierarchy of *ClassInst* objects, beginning from this class instance, and return all *VarInst* objects. Variables declared *static* are not included.

ClassInst* ClassInstResolve(const Name &) const;

Returns a component *ClassInst* if one exists with the given name.

VarInst* VarInstResolve(const Name &) const;

Returns a *VarInst* object if one exists with the given name.

The listed methods extract the OHM representation of the model instance. This representation is then analyzed and manipulated by the simulation environment.

7.5 Simulation environment overview

The simulation environment is the part of OmSim that simulates a model selected from the class store. It consists of several modules which can be divided into four groups:

- Model compilation modules
- Hybrid model simulator modules
- Simulation service modules
- Supervisor and user interaction modules

The task of the model compilation modules is to transform the instantiated model into code that can be used by the numerical simulation algorithms. Different modules handle different parts of the model, such as the continuous parts, the discrete event parts, etc. The simulator modules include numerical integrators, static solvers, and event handling. Service modules include a plotter module for presenting simulation results and tools for

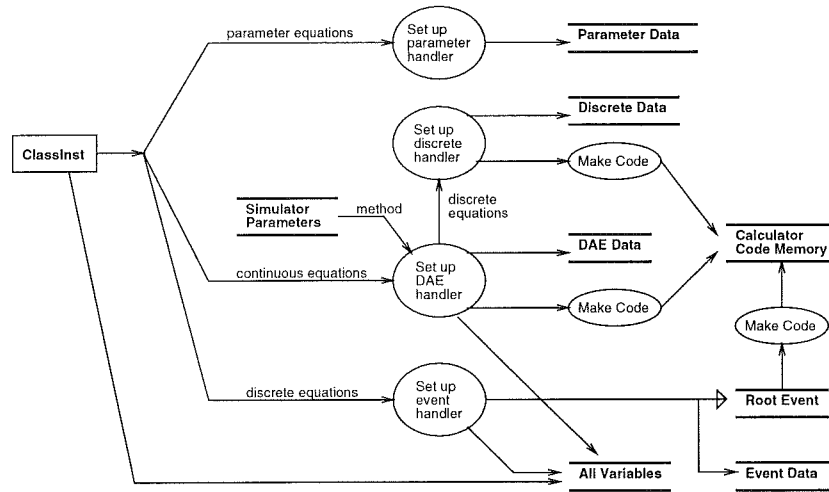


Figure 7.7 A functional model of the model compilation.

displaying and setting initial values and parameters. The supervisor is a logical controller that activates the different tools in the environment in response to user commands.

The focus of this section is the general architecture of the simulation environment from a functional point of view. Some modules concerning the model compilation and the hybrid simulator are described in more detail in the following sections.

A functional model of the OmSim simulation environment is shown in Figure 7.7, which mainly includes the model compilation parts, and in Figure 7.8, including the rest of the modules. The functional model is pictured as *data flow diagrams* using the graphical notation of [Rumbaugh *et al.*, 1991]. The main graphical elements are ellipses representing *processes* or data processing objects, and arrows representing *data flows*. A solid data flow arrow represents main data flows from a producer to a consumer. A dashed arrow indicates a flow of commands or control information. Another type of object is the *data store* drawn as two parallel horizontal bars with a name or a description in between. The rectangular boxes are *actors*. The actors drive the data flow. They work as inputs or outputs of data.

Model compilation

The model compilation part of the simulation environment produces simulation code from a model instance. This part is invoked each time the

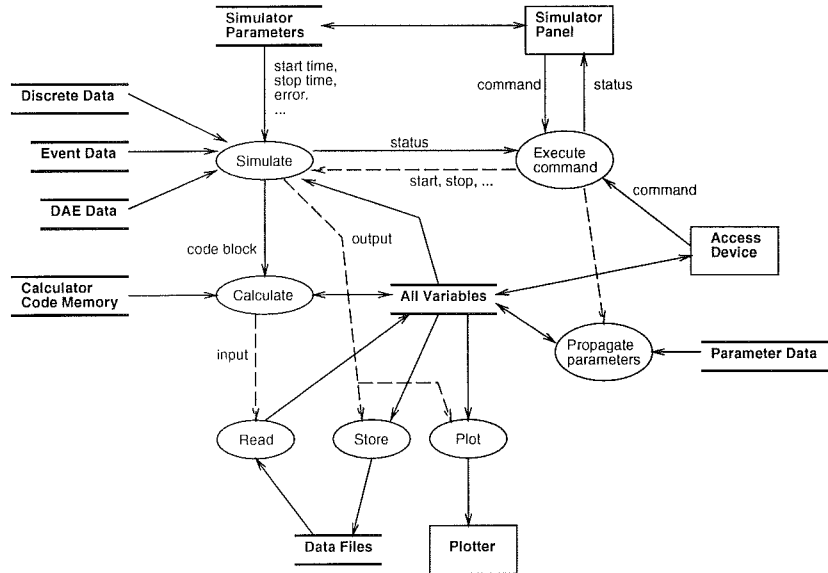


Figure 7.8 Functional model of the simulator, the service modules, and the interaction supervisor.

user selects a new model for simulation.

The data flow diagram in Figure 7.7 shows the model compilation part of the simulation environment. The source of data is the class instance, shown as an actor to the left in the diagram. The model instance, represented as a class instance, is obtained from the model environment and defines the raw model. Data from the model instance consists of equations, variables, and events. This data is used to set up four important modules: a *parameter handler* that propagates user defined parameter values, a *DAE handler* that stores the continuous time simulation models, an *event handler* that stores the discrete event model, and the *discrete handler* that stores the time constant parts of the model. The modules do different processing and transformations of the raw model data during their set up phases. The results are stored in different data stores until they are needed by the simulator modules. Some of the modules send their result to code generators, which produces simulation code that is stored in the *calculator* modules.

More detailed presentations of the DAE handler and event handler processes are given in Chapter 8.

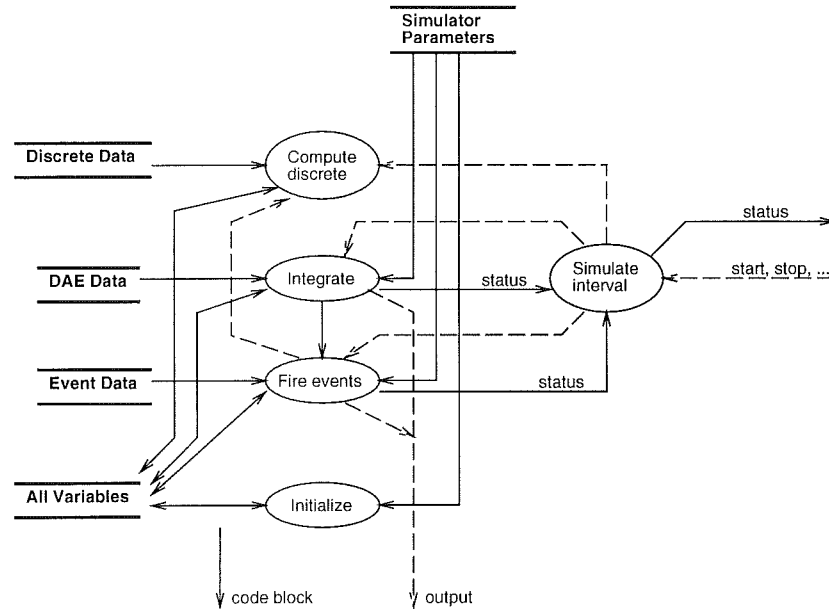


Figure 7.9 Expanded function model of the *simulate* process.

Hybrid model simulator

The processes *simulate* and *calculate* shown in the flow diagram of Figure 7.8, are handled by simulator modules. The process *simulate* is expanded into a more detailed flow diagram shown in Figure 7.9. The *simulate* process consists of the subprocesses *compute discrete*, *integrate*, *fire events*, and *initialize*. These processes are supervised by the subprocess *simulate interval*, which receives commands from the higher level supervisor represented by the *execute command* process in Figure 7.8. The processes in the simulator collect data about the simulation problem from various data stores. The data consists of the size and the type of the current model and start addresses to blocks of code in the calculator. The data flow labeled *code block* in the diagrams are commands from the *simulate* process to the *calculate* process to execute a specific block of code. The *simulate* process receives the computed results from the model variables, stored in the *all variables* data store.

It should be noted that the description of the simulation process is simplified and focussed on the conceptual ideas. The actual implementation is more complicated. For example, the integrator module contains several different integration methods. These are all implemented by nu-

merical routines obtained from different sources as ready-made packages of subroutines, mostly coded in FORTRAN. Unfortunately, all the different integrators have different calling interfaces and require slightly different driving procedures. It has been an important effort to encapsulate the differences and to define a standardized *Integrator* class which could serve as an interface to all the different methods. Particular methods are represented by subclasses of *Integrator*.

Interaction between modeling tools, like OmSim, and the actual numerical routines for simulation would be simplified by some kind of standardized interface for integration methods. One possibility would be to use DSblock [Otter, 1992]. Attempts to define uniform interfaces to modular numerical software, using object-oriented methods, have been made; for example, see [Gustafsson, 1993] and [Skjellum *et al.*, 1993].

Simulator service modules

The simulator service modules include tools for plotting results, for storing results in external files, and for reading input data from external files. There are also user interface modules for direct interaction with model variables and simulator parameters. Different modules handle the processes *read*, *store*, and *plot*. The actor *Access Device* handles the user interaction with model variables. Initial values and parameter values are displayed and changed by the access device. Different types of access devices have different ways of organizing the model variables. Figures in Chapter 3 show examples of control panels and graphical displays for the different modules.

An important service module is the *Calculator* used for evaluating numerical expressions of the simulation model. The module implements the *calculate* process in Figure 7.8 and it maintains the *calculator code memory*. It is used by the numerical integration routines, by the event simulator, and by the static equation solver. In the current OmSim implementation the calculator is a virtual stack machine that operates on model variable instances. A complication is that the numerical routines require the unknowns to be allocated consecutively in real arrays. In order to avoid excessive copying of variables, the calculator allows variable instances to be reallocated to such arrays.

The implementation of the calculator module has a major effect on the simulation speed. However, simulation speed has not been considered very important in the first OmSim prototype. It is relatively easy to replace the code generator and the calculator module, so that efficient simulation code in C or FORTRAN is generated, compiled, and dynamically linked into OmSim.

Simulator supervisor

The simulator is normally controlled interactively by the user. A simulation experiment is often a process involving several steps and iterations. A typical scenario for a simulation session involves the following activities. A session starts when the model has been successfully instantiated and introduced to the simulator. At this stage, all parameters and state variables have default values, defined in the model. If the model is well-defined, it should be possible for the user to just push the start button and run the simulation until the specified final time. The user repeats the simulation from the beginning with different parameters and different initial values. It is also possible to run a simulation up to a specific time, change some parameters and some state variables, and then continue the simulation from current time.

Each time the user changes a parameter, the simulator has to propagate the new parameter value according to the parameter equations of the model. Each time the user restarts the simulation with a new initial state, the simulator has to solve the static initialization problem, defined by the static relations of the model. The same applies if the user changes the value of any dynamic variable at a temporary stop in the dynamic simulation.

From the user's point of view, it is reasonable to consider the simulation process residing in two different modes: *initialization mode* and *active mode*. Each simulation run starts in the initialization mode and ends in the active mode, where the user has the option to continue the simulation run or to restart from the initialization mode. Parameters can be manipulated in both modes. In the initialization mode the user can manipulate initial values, which are remembered from one simulation run to the next. In the active mode, the user can manipulate current values of simulation variables but these settings are volatile.

The user's view of the simulation process in OmSim is similar to many other interactive simulators like, for example, Simnon [Elmqvist *et al.*, 1990]. Differences that make it more complicated in OmSim are the parameter propagation and the state initialization steps. These steps may themselves involve user interaction, and they may be interrupted or they may fail to complete their tasks.

Because of the complex dependencies between the atomic operations in the simulation process, a supervisory control system is introduced to aid the user. The control system is displayed in Figure 7.11. The purpose of the supervisor is to accept commands from the user and execute them as a proper sequence of atomic simulator operations. The supervisor should accept only those user commands which are appropriate at the particular

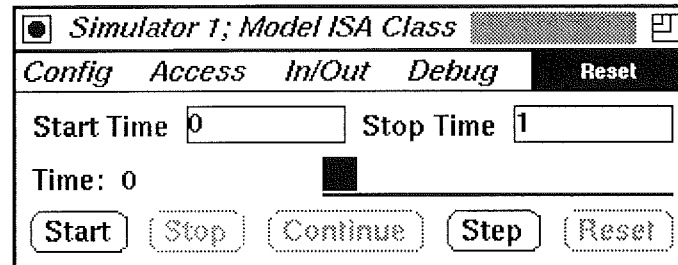


Figure 7.10 Control panel for the OmSim simulator.

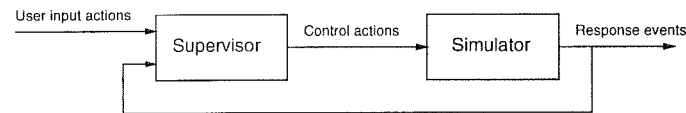


Figure 7.11 The supervisor viewed as a discrete controller for the simulator.

stage in the simulation process.

User input actions to the supervisor come from a command language interpreter or from the control panel shown in Figure 7.10. The upper part of the control panel consists of a set of pull-down menus for specific operations and for opening of various subtools. It also has a status field displaying the current status of the simulator. The panel has numeric input fields for specifying start time and stop time of the simulation, and a meter showing the current simulation time. In this section we will focus on the five buttons in the lower part of the panel labeled *Start*, *Stop*, *Continue*, *Step*, and *Reset*, which are used for controlling the simulation activity.

The purpose of the supervisor is to monitor and control the simulation process according to the user's commands. For that purpose, the supervisor has an internal model of the simulation process, represented as a state machine. This state machine, shown as a state diagram in Figure 7.12, completely determines the behavior of the supervisor.

The user's input actions are associated with the five buttons at the control panel, except for the *Stop* button which is special. In addition, the user can issue a *Model Reset* command from the *Config* pull-down menu. He or she can also modify parameters and variables by means of special model access tools. Such manipulations result in *Set Parameter* and *Set Variable* input actions to the supervisor. The supervisor also accepts a *fail* input from the simulator, indicating that the last command was not

7.5 Simulation environment overview

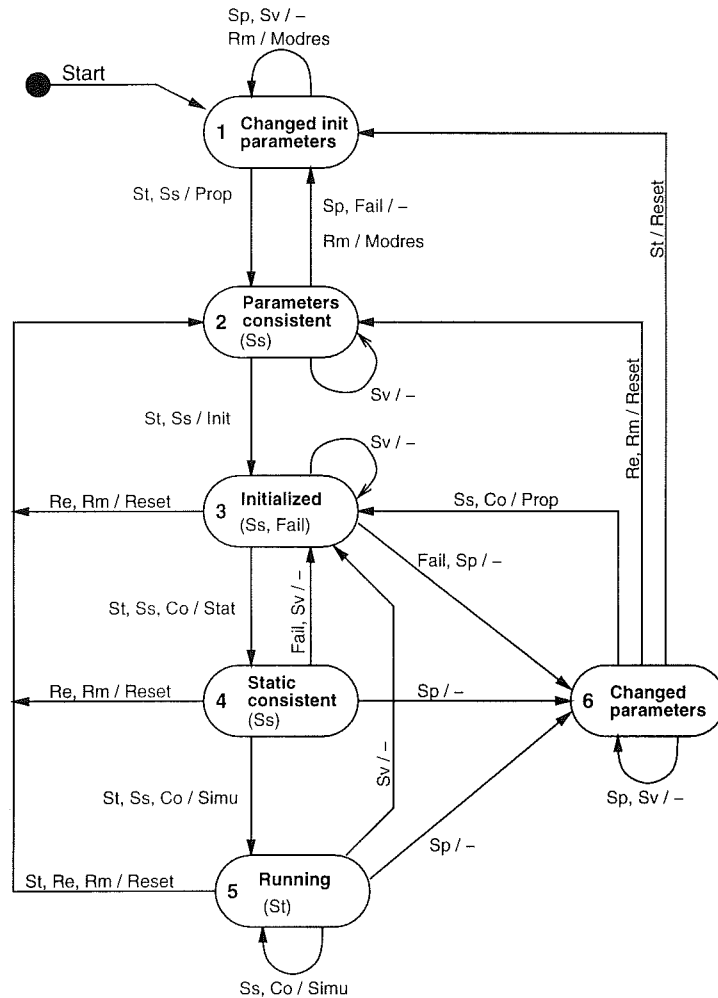


Figure 7.12 The supervisor state machine is a model of the simulation process. The transition arcs are labeled with input events followed by the output atomic simulator actions.

successfully executed. Mainly the parameter propagation and the static initialization operations may render a *fail*. The supervisor's input actions are summarized in the following list. Each item is labeled with the short form used in the transition labels of the state diagram of Figure 7.12.

Co The *Continue* button is used for restarting the current simulation

from the current time.

Fail The *fail* input comes from the simulator in response to a disrupted parameter propagation or static initialization.

Re The *Reset* button is used to reset the simulator to a pre-initialization state, and the current simulation time to the starting time.

Rm The *Reset Model* menu action is used to reset the simulation model to the original state, as directly after the instantiation. All user specified initial values and parameters are forgotten and replaced by model defaults.

Sp A *Set Parameter* command is issued when the user changes a model parameter.

Ss The *Step* button is used to advance the simulation process one step. The exact meaning of a *step* depends on the current status of the simulation.

St The *Start* button is used to restart the simulation from the initial state.

Sv A *Set Variable* command is issued when the user enters a new initial or current value for a model variable.

The output from the supervisor consists of atomic simulator operations. The control actions are specified as outputs in the state diagram of Figure 7.12, and described briefly in the following list. The items are labeled by the short names used in the diagram.

Init initializes the simulation by setting the initial value of all model variables and then firing the initial event. Tools for manipulating model variables are switched from accessing the initial values to accessing the current values.

Modres removes all user specified parameters and initial values and restores the default values defined by the model. This command puts the simulator and the simulation model in the same state as when it was first instantiated.

Prop propagates user specified parameter values according to the parameter constraints defined by the model. This command may result in a *fail* event, recognized by the supervisor.

Reset resets the simulator to a pre-initialization status. Current variable values are forgotten but user specified parameters and initial values are kept. The user gets access to modify initial values.

Simu runs the simulator to advance the current time. The command can be executed in single-step mode or until the specified final time is

reached. This command can be interrupted by the *Stop* button at the control panel.

Stat performs static equation solving in order to reach a consistent initial state for the model. This command may render a *fail* event which is recognized by the supervisor.

The state machine of the supervisor has six states. The initial state, state number 1, is entered when a model instance has been successfully introduced. A special concept of *halting states* for particular input events has been used in the supervisor. A halting state for an input is indicated by the input name in the state node. For example, as shown in Figure 7.12, states number 2, 3, and 4 are halting states for the Ss (step) input event. The meaning of a halting state is that the supervisor makes a halt and waits for a new input event if a state transition ends up in a halting state for the current input event. The normal behavior is otherwise to move on to another state, as long as there is a transition defined for the current input event. For example, if the supervisor is residing in State 1 and receives a St (start) input it will sequentially move to State 2, 3, 4, and 5. It stops in State 5 since this is a halting state for *Start*. During the sequence it will issue the simulator commands *Prop*, *Init*, *Stat*, and *Simu*. The same sequence can be achieved by Ss (single step) inputs but since the states 2, 3, and 4 are all halting states for Ss, four Ss input events are required to reach State 5.

The concept of halting states and asynchronous state transitions are introduced in order to get a simple state machine with few states. The same kind of behavior could have been achieved by a standard state machine, without the concept of halting states and repeated transitions, but this would have required more states.

It is straightforward to implement the supervisor based on the state machine in Figure 7.12. The state machine can be represented as a table which makes it easy to modify the behavior and to add additional states, inputs or actions. A by-product of the table-driven supervisor is that it also defines which input events are admissible in each state. This is used for enabling and disabling the corresponding buttons in the simulator's control panel.

7.6 Open architecture for OmSim

It is desirable to create an open environment of tools for computer-aided control engineering. Attempts to specify Open architectures and integrated environments for CACE have been made by different research

groups, for example, by the group at the University of Wales, Swansea [Barker *et al.*, 1993] and by the group at DLR, Germany, [Grübel, 1992]. Interesting issues are what kind of tools are needed and at what granularity they should be integrated, how tools should communicate and how they should be controlled to fulfill their tasks.

It was an original goal of the project in Lund to design an open and integrated environment for modeling, analysis and simulation of dynamic systems [Andersson, 1989b]. However, the current implementation of OmSim is an example of an integrated but not very open environment. The implementation is modular but it runs as a single process and the interfaces between the modules are not formally defined. We like to view OmSim as a prototype for an open environment of integrated tools where we easily can experiment with different tools and architectures without the need to specify the interfaces in advance (the hacker's approach to software development).

In [Barker *et al.*, 1993] a CACE framework reference model is suggested. The framework is inspired by a CASE (Compute Aided Software Engineering) framework reference model and it contains a set of common services that can be accessed by the tools in the environment. The services suggested in the CACE framework are the following:

- Modeling Services
- Database Services
- Task Management Services
- User Interface Services
- Message Services

The *Modeling Services* provide a standardized representation for models and engineering data. It provides a *high level model description layer*, a *neutral format model layer*, and a *control data object layer*. *Database services* maintain relationships between models of different versions, experimental frames, and input and output data, from all stages of the design process. Database support in CACE environments is considered important and has been much discussed and studied by several research groups, for example, in the GE-MEAD project [Taylor *et al.*, 1989]; see also [Tan and Maciejowski, 1989, Grübel and Joos, 1991, Hope *et al.*, 1991, Maffezzoni *et al.*, 1994].

Task management services standardize the control of the tools in the environment. Its role is similar to the simulator supervisor in OmSim but it must be more flexible and based on a common, standardized command language. The *user interface services* provides a standard “look and feel” to the different tools in the environment. It can work as a graphical

front end to some tools but should also provide direct text-based access. Standardized *message services* define a common protocol for messages and data. They provide the means for having tools running in parallel on a distributed net of compute servers.

Figure 7.13 shows the architecture of OmSim mapped onto the “Swansea toaster” reference model. To the right in the picture, the Modeling Services are represented as three data buses. The data format on the *High Level Model Bus* is Omola code. The *Neutral Model Bus* carries OHM representations. OHM is a mathematical and logical model representation in symbolic form. It fulfills the requirements of being a neutral format. It can be analyzed and manipulated by various tools in the environment. It can also be translated into more specific representations, for example, into a simulation model used by a particular simulation algorithm or into DSblock [Otter, 1992]. In order to turn OmSim into an open environment, it is necessary to give a precise definition of the OHM data structure. This may be done using some standard data modeling language like STEP and EXPRESS [Harbison-Briggs *et al.*, 1993].

The only *Control Data Object* that is used by OmSim is time response objects, produced as a result from simulations. Time responses can be displayed by the plotter. The different tools in the environment are controlled directly by the user through separate graphical user interfaces or by OCL scripts. The OCL interpreter can be viewed as a Task Manager. The class store in OmSim does not qualify as a proper database service but it has at least some of the properties of a real database. It is able to search and load models from permanent store.

An interesting question that arises when a CACE reference model is standardized, is the granularity of the tools and their interaction. For example, in Figure 7.13 the simulator is modeled as a single black-box that accepts a model on a neutral standardized format and returns simulation results. However, this chapter has shown that the simulator itself contains several modules, like numerical integrator, event detection and execution, static equation solver.

7.7 Summary

The architecture of OmSim was presented in two main parts: the modeling environment and the simulation environment. The modeling environment stores Omola models internally. They can be displayed in different ways: as Omola code or as tree diagrams showing composition and inheritance hierarchies. Models can be defined and displayed graphically by a component diagram editor. Omola definitions are organized in libraries. OmSim

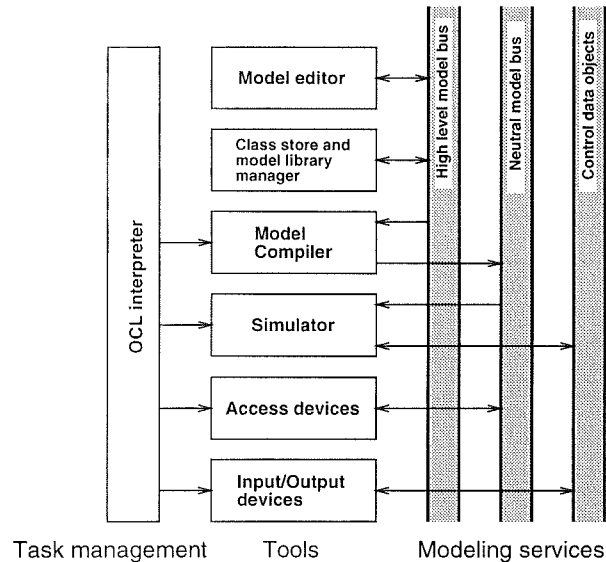


Figure 7.13 The architecture of OmSim viewed as CACE Open System Reference Model.

can search the external file system organized as multiple model databases for definitions that need to be loaded into the internal store.

The object-oriented data structures representing Omola class definitions and model instances were presented. The procedures for resolving symbolic references and model instantiation were also discussed in some detail.

The OmSim simulation environment compiles Omola models into simulation code. The environment contains an interactive simulator for hybrid models and a set of supporting tools for setting parameters, and initial values, for plotting simulation results, etc. Functional diagrams for compilation and data handling in the simulator were given. The finite state machine controlling the interactive simulation process was also defined. Finally, OmSim was discussed in the perspective of open environments for computer-aided control engineering.

8

Model Manipulation

The model instance, discussed in the previous chapter, is the data structure that contains all the behavior equations in their original form. An important task is to determine if the set of equations constitutes a well-defined model from a mathematical point of view. Basically, this means to analyze the equations and to check that every unknown variable of the model is determined by them. If the model is going to be used for simulation, it is also necessary to manipulate the equations and the event definitions into a form most suitable for the numerical simulation algorithms.

The topic of this chapter is to give an overview of the analysis and symbolic manipulations performed on Omola hybrid models in OmSim. The first section is devoted to the continuous time parts while the second section is concerned with the discrete event parts of the model. The final section discusses the algorithms for detecting continuous time state events during simulation.

8.1 Continuous time model

The raw model is analyzed in order to detect anomalies that make it impossible to derive a proper simulation model, and to detect undesired properties that are likely to cause problems for the numerical solvers. It is important that errors and potential problems are detected as early as possible, and presented to the user in a form that is related to the original Omola model, so that he or she can easily correct it. This is in general a very difficult problem.

Model transformations are made in order to obtain a representation that is suitable for the chosen numerical algorithm. Some manipulations are done independently of the algorithm, while others are specific for a particular integration method. The choice of method also depends on how

successful the manipulations are. For efficiency reasons, it is desirable to do as many general transformations first and to save the result. Method specific manipulations are postponed as far as possible. This makes it efficient to change integration method which is important since it is always a good idea to compare the results of different methods.

Some model manipulations are done mainly in order to improve efficiency during simulation. This makes it possible to design high-level Omola models, mainly focusing on the clarity of the description, without considering simulation efficiency. The simulation model can be optimized for efficient simulation by detecting and eliminating redundant variables resulting from the modeling methodology.

The original model and the final result

The instantiated model provides the model equations in their original form together with equations generated from the connections. These equations serve as a starting point for the analysis and manipulations discussed in the following. The following information is extracted from the model instance.

- A set of continuous time variables denoted x
- A set of equations depending on x
- A set of assignments where variables in x are assigned to functions of x , like for example:

$$x_i := g_i(x)$$

Here and in the following, the subscript i is used to indicate an element in a set, e.g., $x_i \in x$. The notion of *sets* of variables and *sets* of functions are used in order to emphasize that variables and function are objects that can be rearranged and manipulated. A set of variables can be regarded as a vector, and a set of functions can be regarded as a vector function, with some assumed ordering of elements.

The functions returned from the model instance are in fact also depending on discrete variables and parameters, but they are left out in this discussion in order to simplify the notation. Discrete variables and parameters are regarded as unknown constants during equation manipulation and analysis, and as known constants during simulation, as far as only the continuous time aspects are regarded. The variables x contain state variables and their derivatives, up the highest appearing order, and algebraic variables. The equations and assignments correspond to equation (5.5) of the OHM in Chapter 6. They include all explicit equations in the model and all equations generated from connections. Variable bindings and connections with defined causality yield the assignments.

8.1 Continuous time model

The continuous model, which is a part of the OHM representation, is extracted from the model instance and transformed by a sequence of manipulations into a general form suitable for simulation with different integration algorithms. The resulting form is also useful for other purposes, such as analysis and design. The transformation may introduce additional variables and equations in the model. Equations and variables that are not needed to represent the dynamic behavior are separated, thus reducing the size of the problem. The transformation process also includes structural analysis for detecting possible errors as early as possible. Defective models result in error messages. The result from a correct model is available in the following form.

- A dynamic problem consisting of
 - two sets of variables: *dynamic state* variables y and their first-order derivatives y'
 - a set of *algebraic state* variables z
 - a set of *auxiliary algebraic* variables v
 - a set of *dynamic equations* and blocks of equations sorted in computational order
- An output problem consisting of
 - a set of *output* variables w
 - a set of *output assignments* for computing w
- A discrete problem consisting of
 - a set of *implicitly discrete* variables d
 - a set of equations to be solved for d

The variable sets are subsets of x augmented with auxiliary variables introduced in the transformation process. There is a one-to-one map assumed between the dynamic state variables y and the derivatives y' . The algebraic state variables z are variables which do not appear differentiated but must be solved from an implicit equation by the DAE solver. If the problem is on ODE form there are no variables of this kind. The auxiliary algebraic variables v can be computed explicitly from y , y' and z . They represent intermediate values and may be hidden for the integration method.

The output variables w are not needed for solving the simulation problem but may be of interest to the user for displaying and plotting. The implicitly discrete variables are a subset of the original variables x , which have been deduced to depend only on discrete variables and user parameters. The output variables and the implicitly discrete variables with their defining assignments can be separated from the main simulation problem.

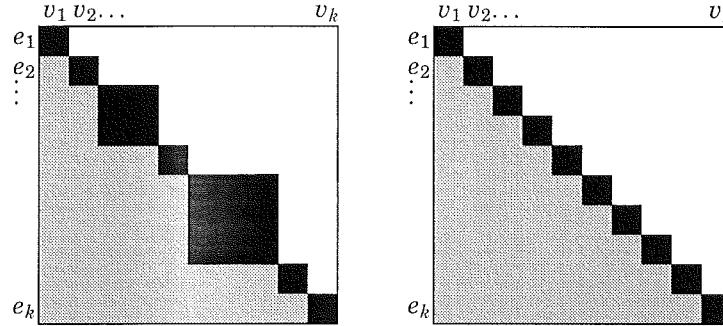


Figure 8.1 Illustrations of BLT forms. Each row corresponds to an equation and each column corresponds to a variable. Black areas means that a variable appears in the equation. The white area means that the variables do not appear in the equations. In the grey area variables can appear or not in the corresponding equations. The left matrix is an example of a general BLT form while the right matrix shows the special case where all blocks have single elements.

BLT form

The dynamic equations are returned as a sequence of blocks where each block contains one or more equations. The equations are all first-order differential and algebraic equations. Each block can be solved as a separate problem assuming all previous blocks are solved. This means that the dynamic equations are sorted in *Block Lower Triangular* (BLT) order. This can be illustrated by the incidence matrix, sometimes called the structural Jacobian, shown in Figure 8.1. The incidence matrix is a square matrix of Boolean elements indicating which unknown variables appear in each equation. A row in the matrix corresponds to an equation, while a column corresponds to an unknown variable. Element (i, j) is true (or 1) if variable v_j appears in equation e_i , otherwise it is false (or 0). In Figure 8.1, elements which are true are indicated by black, while false elements are indicated by white. Grey areas indicate elements that are either true or false. A BLT form is a permutation of equations and unknowns so that the incidence matrix is triangular or block triangular. The BLT form returned from the equation manipulations regards the state derivatives y' and the algebraic variables z and v as unknowns. The equations of the BLT are of three different kinds:

$$\begin{aligned} y_i' &:= f_{1,i}(y, y', z, v, d) & (a) \\ v_i &:= f_{2,i}(y, y', z, v, d) & (b) \\ 0 &= f_{3,i}(y, y', z, v, d) & (c) \end{aligned} \tag{8.1}$$

8.1 Continuous time model

Some equations are explicit assignments to state derivatives (8.1a). Some of these assignments are of the trivial form $y_i' = y_j$ and generated in the process of transforming the original model into a first-order differential equation model. Other equations are assignments to algebraic variables (8.1b). Assignments of the first two types always appear in single element blocks. The third type of equations can appear in larger blocks. They are implicit equations which can be written on residual form (8.1c). The BLT form means that the equations from the three groups are sorted so that the unknowns appearing in the equations of each block are either unknowns of the same block or solved from some of the previous blocks.

One goal of the equation manipulations is to get as few implicit equations as possible. If there are no implicit equations at all, it is possible to use a standard ODE solver to simulate the model, which may be more efficient.

The resulting representation, discussed so far, is independent of the chosen integration method. Now it remains to transform the equations (8.1) into a form required by the chosen integration algorithm. Three types of integration methods are considered here: general ODE methods, implicit Runge-Kutta methods for DAE problems, and general multistep DAE methods.

General ODE methods

A typical ODE method, for example, a Runge-Kutta method like RKF45 [Hairer *et al.*, 1987], solves problems of the form

$$\frac{dy}{dt} = f(y, t), \quad (8.2)$$

and requires a method for computing the function f when y and t are given. This is possible if the manipulated model has no implicit equations of type (8.1c). In this case, the BLT form has only single element blocks on the diagonal. The function f can be constructed from (8.1a) and (8.1b) by substituting derivatives and algebraic variables. In practice, the substitution is not done since the assignments can be used directly, evaluated in order as they appear in the BLT form.

An implicit Runge-Kutta methods for DAE systems

One possible representation for DAE systems is the form

$$B \frac{dY}{dt} = f(Y, t), \quad (8.3)$$

where B is constant and in general a singular matrix. This form is for example used by Radau5 [Hairer *et al.*, 1989].

The required form can be obtained by the following operations. Explicit derivative assignments of type (8.1a) are used as they are, and explicit auxiliary variable assignments of type (8.1b) are hidden for the integration algorithm. Only the implicit equations have to be manipulated. Consider a block of equations of type (8.1c) in the BLT form. In that particular block subsets of y' and z are regarded as unknowns, while the remaining variables are either computed by the previous blocks or do not appear at all. Let \bar{y} , \bar{y}' and \bar{z} denote the subsets of unknowns. The block of equations can then be written as the system of equations

$$0 = f(\bar{y}, \bar{y}', \bar{z}). \quad (8.4)$$

The dimension of f is equal to $\dim(\bar{y}) + \dim(\bar{z})$. A new set of algebraic variables \hat{z} is introduced; one element in \hat{z} for each element in \bar{y}' . When \bar{y}' is substituted by \hat{z} in (8.4) the new system of equations becomes

$$\begin{pmatrix} 0 & 0 & 0 \\ I & 0 & 0 \end{pmatrix} \frac{d}{dt} \begin{pmatrix} \bar{y} \\ \bar{z} \\ \hat{z} \end{pmatrix} = \begin{pmatrix} f(y, \hat{z}, \bar{z}) \\ \hat{z} \end{pmatrix}. \quad (8.5)$$

This equation is similar to the desired equation (8.3). When all blocks are considered, the state vector Y in (8.3) consists of y , z , and the additional variables \hat{z} are introduced for each block. The matrix B will contain ones and zeros.

General DAE method

DASSL is an example of a general multistep DAE method [Brenan *et al.*, 1989]. It assumes a model of the form

$$0 = g(y, \frac{dy}{dt}, t) \quad (8.6)$$

and it requires a method to compute the residual

$$\Delta := g(y, y', t) \quad (8.7)$$

when the arguments are given. This is easy to obtain if the assignment of type (8.1a) and (8.1c) are used as

$$\begin{aligned} \Delta_i &:= y_i' - f_{1,i}(y, y', z, v, d) \\ \Delta_i &:= f_{3,i}(y, y', z, v, d). \end{aligned} \quad (8.8)$$

Similar to the other methods, the algebraic variables v and their assignments (8.1b) can be hidden for the numerical integration routine.

Transformation of the raw model

The main steps in transformation procedure from the raw model obtained from the model instance, to the simulation model defined above, are now presented. A general discussion of the equation manipulations in OmSim is also given in [Mattsson *et al.*, 1993].

1. Get all variables and all equations from the model instance.
2. Create assignments from variable binding expressions, and add them to the set of equations.
3. Divide the variables into four groups with respect to their variability determined during class instantiation. The variability groups are:
 - a. Constants, i.e., variables with a known fixed value
 - b. Parameters
 - c. Discrete variables
 - d. Continuously time varying variables
4. Divide the equations into four groups with respect to their variability. The highest variability rank of the appearing variables determines the variability of the equation. Hence, the equation variability groups are:
 - a. Constant equations
 - b. Parameter equations
 - c. Discrete equations
 - d. Continuous equations
5. Check that the constant equations are consistent. If they are fulfilled, they can be disregarded in the following steps. If they are not fulfilled, the model is wrong and the analysis is terminated with an error message.
6. Use the parameters and the parameter equations to set up the Parameter Handler module.
7. Check that the set of continuous equations are structurally nonsingular. If they are singular, produce an error message and terminate the procedure.
8. Make a BLT partition of the continuous equations. For each block, eliminate the time derivative operator by introducing new variables. Divide the blocks in three groups
 - a. Constant BLT blocks
 - b. Discrete BLT blocks
 - c. Time varying BLT blocks
9. Try to solve the constant BLT blocks. If this is not possible, produce an error message and terminate the procedure. Hereafter, regard

the variables of the solved blocks as constants and disregard the corresponding equations.

10. Use the discrete BLT blocks to set up the Implicit Discrete Handler module.
11. Deduce the *differentiated index one* problem of each time varying BLT block and do index reduction when needed.
12. Manipulate each time varying BLT block and try to make it diagonal. Eliminate auxiliary algebraic variables.
13. Turn the problem to a first-order DAE system.
14. Separate the blocks into a *dynamic part* and an *output part*. The output part consists of those blocks not needed for solving the dynamic problem.
15. Check if the dynamic part consists of only assignments. In this case the problem is classified as an ODE problem, otherwise it is an DAE problem.

The check for structural nonsingularity in Step 7 checks if the number of equation and continuous variables are the same, and if each variable can be paired with an equation in which it appears. State variables and their derivatives are regarded the same in this analysis step. Speaking in terms of the incidence matrix introduced above, this step consists of permuting the rows or the columns of the matrix so that incidence is obtained for all diagonal elements. An efficient algorithm for this is for example described in [Duff *et al.*, 1986]. If the pairing of variables and equations is not successful, the problem is structurally singular. This means that the model has too many or too few equations, or that some equations involve the wrong variables. If it is found that the problem is singular, the missing or redundant number of equations is obtained from the algorithm. In this case the problem is to present an appropriate error message to the user.

The next step (Step 8) in the procedure consists of ordering the equations into a sequence of subproblems. This is done by symmetric row and column permutations, so that the incidence matrix becomes block triangular, with minimal blocks. An efficient algorithm for this is given by Tarjan [Tarjan, 1972, Duff and Reid, 1978].

Some blocks in the BLT partition may be deduced to be constant or discrete. They can be separated from the time varying problem. A block is constant or discrete if none of the equations refer to a time varying variable or the time explicitly, and if none of the unknown variables in the block appears differentiated. If any equation in the block refers to a discrete variable, then the block is considered as discrete, otherwise it is constant. The unknowns of the block are hereafter regarded as discrete or

8.1 Continuous time model

constant. The constant blocks are solved immediately while the discrete blocks are solved during simulation after each discrete event.

Step 11 uses Pantelides's algorithm [Pantelides, 1988] to determine how many times each equation in each block has to be differentiated in order to obtain an index-zero DAE system. This is discussed in more detail below.

After index reduction, each block is manipulated in order to reduce its size and to find algebraic variables which can be hidden for the numerical solver. If the unknowns appear linearly and if the size of the block is not too large it is possible to manipulate it into diagonal form. Another technique for reducing the size of a block is called tearing which is discussed in [Elmqvist and Otter, 1994].

Hierarchical modeling with several levels of submodels introduces a chain of equations and variables of the form

$$x_1 = x_2 = \dots = x_n,$$

represented as separate equations. The intermediate variables x_2 to x_{n-1} can be eliminated and the chain of equations reduces to the single equation $x_1 = x_n$. The eliminated variables are maintained with proper assignments. They will be sorted out as output variables at a later stage.

So far the model contains high order derivatives, defined in the original model or resulting from the previous index reduction. The model is turned into a first-order DAE system by introducing additional state variables and equations. In fact, since all derivatives are represented as separate variables already in z , only equations relating derivatives and states have to be added to the model. For example, assume there is a base variable x and chain of derivatives in increasing order denoted $x_{d,1}$, $x_{d,2}$, etc. Then the new equations $x' = x_{d,1}$, $x'_{d,1} = x_{d,2}$, etc., are added. This corresponds to a *controller canonical form* state-space realization [Kailath, 1980]. It should be noted that from now on, all occurrences of the highest order derivative should be interpreted as the first derivative of the state variable representing the second highest order derivative.

Finally in the model transformation procedure, the output variables and their assignments are separated from the main dynamic problem. These variables need to be computed only on request from the user.

The model has now reached the required form presented above as the BLT sorted equations (8.1). If there are no remaining implicit equations of type (8.1c), then the model can be classified as ODE (state-space) model, and it can be solved by ordinary ODE methods.

Problems with high index

The *index* is an important property of DAE systems. Different definitions of index can be found in the literature. In [Brenan *et al.*, 1989], index is defined as the minimum number of times all or parts of $0 = g(\dot{x}, x, t)$ have to be differentiated in order to determine \dot{x} as a function of x and t . Another definition of index is used in [Hairer *et al.*, 1989]. An ODE problem in state-space form, $\dot{x} = f(x, t)$ is index-zero. A DAE problem in the form

$$\begin{aligned}\dot{x} &= f(x, y, t) \\ 0 &= g(x, y, t)\end{aligned}\tag{8.9}$$

is index-one if the Jacobian $\delta g / \delta y$ is non-singular. This means that in principle y can be solved from the second equation and substituted into the first equation to get state-space form. A DAE problem in the general form $0 = g(\dot{x}, x, t)$ may have higher index. DAE systems with an index of two are difficult for the numerical solver. Available DAE solvers may solve some index-two problems but they fail when the index is higher. Pantelides's algorithm [Pantelides, 1988] can be used to determine the number of times each constraint has to be differentiated in order to reduce the index to one or zero.

Index is not a property of the modeled system but a property of a particular model representation, and therefore a function of the modeling methodology. For that reason, it should be possible to reduce the index by symbolic manipulations. High indices indicate that the model has algebraic relations between differentiated state variables. By using insight in the particular modeling domain it should be possible to eliminate the number of differentiated variables and thus reduce the index of the problem. However, this violates the object-oriented modeling methodology we want to support.

In OmSim it is desirable to have general methods of reducing the index of high index problems to index-zero or index-one. Two methods are currently utilized. One method is based on the elimination of redundant state variables and another method is based on the introduction of so-called dummy derivatives. A simple example is used to illustrate the methods. Regard the following DAE system.

$$\begin{aligned}\dot{u}_1 + u_1 - i_1 &= 0 \\ \dot{u}_2 + u_2 - i_2 &= 0 \\ u_1 - u_2 &= 0 \\ i_1 + i_2 &= i\end{aligned}\tag{8.10}$$

The model may result from two RC circuits connected in parallel as shown

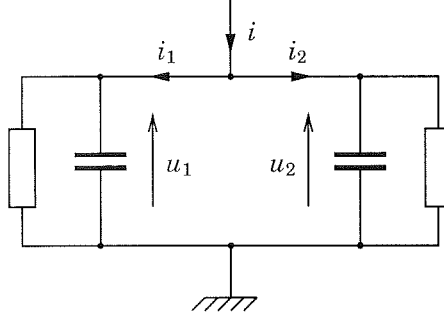


Figure 8.2 Electric circuit resulting in an index 2 DAE model.

in Figure 8.2, and is driven by the current i which is a known function of time. From the circuit diagram it is easy to realize that the model can be reduced to a single RC circuit with one state variable. The DAE model is index-two. It can be reduced to index-one by replacing the third constraint by its derivative. The new index-one problem, with consistent initial values, is mathematically equivalent to the original problem. However, due to numerical errors in the solver there is no guarantee that the original third constraint will hold in the final solution.

One way to reduce the index of (8.10) is to eliminate the redundant state variable. This is done by using a differentiated version of the third constraint to eliminate either \dot{u}_1 in the first constraint or \dot{u}_2 in the second one. It is then possible to transform the problem into explicit ODE form using symbolic Gaussian elimination. The resulting equations in computational order are:

$$\begin{aligned}
 u_2 &:= u_1 \\
 \dot{u}_2 &:= \frac{1}{2}(i - (u_1 + u_2)) \\
 i_1 &:= \dot{u}_1 + u_1 \\
 i_2 &:= \dot{u}_1 + u_2
 \end{aligned} \tag{8.11}$$

This method is used when possible in OmSim. An advantage of the method is that it reduces the size of the problem and, like in this case, often results in a state-space model. However, it is not always possible to find a state variable that can be eliminated.

Another possibility to reduce the index of a problem is to use the method of *dummy derivatives* introduced in [Mattsson and Söderlind, 1990, Mattsson and Söderlind, 1993]. The method consists of augmenting the DAE system with differentiated versions of equations and replacing some of the differentiated variables with new algebraic variables,

Chapter 8. Model Manipulation

called dummy derivatives. In the example above, the system is first augmented by the third equation differentiated. This gives the following over-determined system.

$$\begin{aligned} \dot{u}_1 + u_1 - i_1 &= 0 \\ \dot{u}_2 + u_2 - i_2 &= 0 \\ u_1 - u_2 &= 0 \\ i_1 + i_2 &= i \\ \dot{u}_1 - \dot{u}_2 &= 0 \end{aligned} \tag{8.12}$$

One of the derivatives of the new constraint is chosen and replaced by an algebraic variable in all constraints. For example, choose \dot{u}_2 and let u_2' denote the dummy derivative. The dummy derivative replaces all occurrences of the real derivative and the new system becomes the following.

$$\begin{aligned} \dot{u}_1 + u_1 - i_1 &= 0 \\ u_2' + u_2 - i_2 &= 0 \\ u_1 - u_2 &= 0 \\ i_1 + i_2 &= i \\ \dot{u}_1 - u_2' &= 0 \end{aligned} \tag{8.13}$$

The new DAE system is determined and index-one.

The index reduction techniques are applied to each time varying block in the BLT partition obtained from Step 8 in the model transformation procedure above.

8.2 Discrete event model

In this section, the compilation and simulation of the discrete event model is discussed. Three major issues in the compilation procedure are discussed. First, the event propagation is analyzed. The purpose is to establish chains of events that must be fired synchronously. Secondly, the effect of each set of synchronous events is analyzed. The purpose is to determine which variables are affected by the events and which equations are involved in the propagation of the effect throughout the model. This is called the restart problem. The third issue concerns the compilation of the continuous firing conditions. Last in this section, the simulation problem, in particular the localization of state events, is discussed.

Compiling the discrete event model

Preparing the simulation model consists of analyzing the event propagation and synchronization structure according to the model semantic rules described in Chapter 6. According to the OHM formalism, an event type has conditions defined as explicit model invariants, and actions defined as equations. Event conditions are classified whether they depend on continuous time variables or not. The event actions must, together with the other model equations, determine an initial value for the continuing simulation. Event actions can also consist of commands to event scheduler, print commands, etc.

Analyzing event propagation

The starting point for the preparation of the discrete event parts is the list of event instances obtained from the model instances. The list contains all instantiated event definitions from the model class to be simulated. Additional input consists of the discrete variables and the discrete equations collected during compilation of the continuous time model, discussed previously.

The compilation procedure starts by analyzing the synchronization structure. This is done by constructing a graph according to the following procedure.

1. Get all event instances from the selected model. Also get the discrete variables and the discrete equations from the continuous model compilation.
2. Collect all instantiated event synchronization and propagation declarations from the model instance.
3. Represent each event instance by a node. For each symmetric synchronization, add an undirected link between the corresponding nodes in the event graph. For each directed synchronization, add a corresponding directed link to the graph. If the propagation is conditional, annotate the directed link with the condition.
4. Identify and mark all events that may occur independently of other events. We call these events *root events*. Events that appear in schedule commands or have condition attributes bound to expressions are root events.
5. Cluster nodes that are connected by undirected links into a single node. Clusters which contain at least one root event are considered as root nodes. Only directed links remain in the graph. This is an application of Semantic Rule 12 in Chapter 6.

Chapter 8. Model Manipulation

```

E1, E2, E3, E4 ISAN Event;
WHEN E1 CAUSE E2;
WHEN E2 DO NEW(x) := 1; END;
WHEN y > 0 CAUSE E1;
WHEN E1 AND z < 0 DO NEW(y) := 0; END;
WHEN E3 DO schedule(E4,1.0); END;
E3 = E1;

```

Listing 8.1 Event definitions for the example discussed in the text.

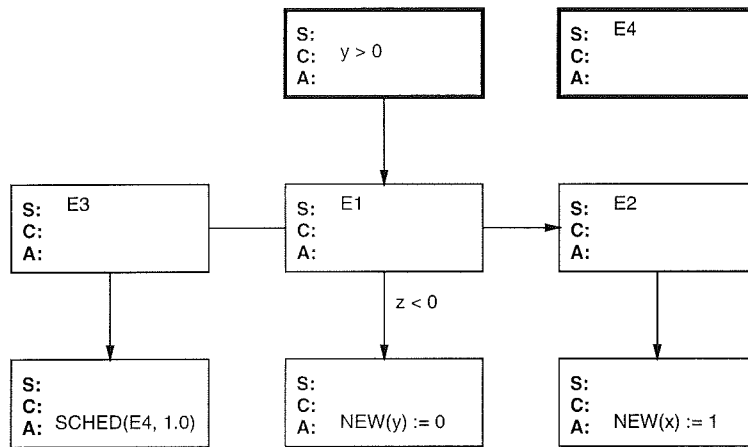


Figure 8.3 Event graph for the example discussed in the text.

6. Reduce the graph by removing empty events and circular directed synchronization according to Proposition 1 in Section 6.3.
7. Construct a tree for each root event node. The tree is a subgraph of the complete graph. It is constructed by starting from the root event and including every link and node that can be reached by directed links.

As an example of applying the graph construction and reduction procedure, regard the Omola event definitions in Listing 8.1. The graph resulting directly from the definitions is shown in Figure 8.3. The definitions of the event classes called E1 to E4 result in one event node each. The first WHEN-clause results in a link from E1 to E2. The second WHEN-clause results in a link from E2 to a new anonymous event node with an action, and so on. The symmetric synchronization between E1 and E3 result in an undirected link in the graph. The top nodes in the graph are root events, the top left node because it has an event condition, and E4 because it is

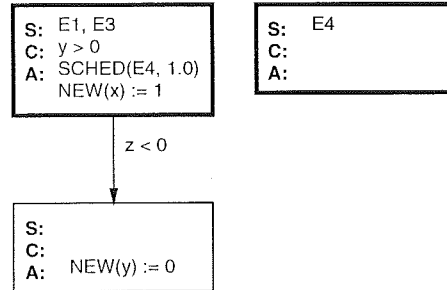


Figure 8.4 The graph in the previous figure reduced to minimal size.

scheduled. The graph in the example can be reduced by applying behavior preserving transformations described in Chapter 6. The minimal graph is shown in Figure 8.4. It is not always a good idea to reduce the graph into the minimal form since it may be difficult to trace the result back to the original model, for example, to produce error messages.

From the root event trees the discrete event simulation model can be constructed. Each root event defines a particular restart problem for the continuous simulator. First assume that a root event tree has only unconditional links between the nodes. In this case, the equations and other actions associated with the root event, are obtained by a complete traversal of its tree. The discrete equations collected from a root event tree together with equations from the continuous time model that are affected by the event, define a *restart problem*.

Analysis of the restart problem

The event equations define a new discrete and continuous state to be used as the initial value when the simulation is restarted. An event equation may explicitly assign a new value to a discrete or a continuous state variable. For example, the event equation

$$\text{NEW}(x) := 0;$$

defines the initial value of x for the continued simulation. Depending on the ordinary model equations, the effect of changing one variable may propagate to other variables of the model. In other words, as soon as one variable is changed as a result of an event, all equations must be considered in order to determine the new consistent state. For many events, however, only a small subset of all variables are affected and only a small subset of all equations have to be evaluated. By structural analysis of the equations, it is possible to determine the set of equations

that must be evaluated, and the set of variables that can change, as a result of an event.

First the special, but rather common case where the continuous time problem has been transformed into state space (ODE) form, will be considered. Also assume that only discrete variables and continuous state variables appear as unknowns in the set of event equations, and that these equations can be solved for the unknowns. In this case, the restart problem is solved and the event is executed by the following computational steps:

1. Use the solved event equations to compute new values for some discrete and some continuous state variables. Keep the current values for the remaining variables.
2. Evaluate the implicitly discrete equations.
3. Evaluate the continuous time dynamic equations.
4. Evaluate the output equations.

In this case the result from the analysis of the continuous time model (8.1) is used directly, and the dynamic equations and the output equations are evaluated in the same order as for the continuous time problem. Written as variable assignments, we get the following sequence defining the event restart problem:

$$\begin{array}{llll}
 q_i & := & \delta_{1,i}(q, y) & \} \text{ (a)} \\
 y_i & := & \delta_{2,i}(q, y) & \} \\
 d_i & := & \delta_{3,i}(q) & \text{ (b)} \\
 y_i' & := & f_{1,i}(y, y', z, v, d, q) & \} \text{ (c)} \\
 v_i & := & f_{2,i}(y, y', z, v, d, q) & \} \\
 w_i & := & f_{4,i}(y, y', z, v, d, q) & \text{ (d)}
 \end{array} \tag{8.14}$$

The subscript i is used as before to indicate that there are several equations of each kind. The first two groups of assignments, labeled (a), are event specific and sorted in computational order. The implicitly discrete equations (b), the dynamic equations (c), and the output equations (d) are event independent and obtained from the continuous model.

If the continuous model is not on ODE form, the evaluation sequence (8.14) can still be used as long as the variables computed by the event equations (a) are not appearing as unknowns in the continuous problem. The dynamic problem (c), however, includes implicit blocks of equations that have to be solved numerically.

In case the event equations do not fulfill the above requirements but contain derivatives, output variables, and intermediate algebraic variables as unknowns, then the suggested event execution cannot be used

directly. It means that the effect of an event propagates through the model in a direction which is different from the computational causality of the continuous model. For example, assume a model with an input-output submodel and an event which prescribes a new output value. The indirect result of the event is that the internal state of the input-output model is affected from the output. There are two possible ways to construct the event restart problem in such cases. One way is to substitute unknowns in the event equations using equations from the manipulated ODE model, and then solving for state variables which are not appearing as unknowns in the ODE model. This may be a useful method in some cases but it may also create a very inefficient event restart problem if many substitutions are needed.

The general approach to construct the event restart problem is to manipulate all the original model equations together with the specific event equations, without using the result of the continuous model manipulations. In this case, all continuous time variables and derivatives are considered as unknowns. In the event equations only the new-operated variables are unknowns. Note that for example $new(x)$ appearing in an event equation is considered the same unknown as x appearing in an ordinary equation. The procedure is now similar to the continuous model structure analysis using Duff's algorithm for pairing equations and variables and Tarjan's algorithm for sorting the equations in a BLT order. However, in general there are two few equations. The missing equations are the continuity assumptions, i.e., variables not affected by the event that keep their old values. What is missing is a number of equations of the type $new(x) := x$. Instead of actually adding such missing equations, the corresponding unknowns can be removed as unknowns from the problem and treated as known variables. The following procedure is used:

1. Use Duff's algorithm on the complete set of discrete and continuous equations and unknowns.
2. For variables that do not get paired with an equation, give them a dummy equation containing all unknowns.
3. Apply Tarjan's algorithm. Variables with missing equations will then end up as unknowns of the last block.
4. Apply the continuity assumption for state variables, i.e., for variables that appear differentiated, and remove them as unknowns from the block.
5. Apply Duff's algorithm to the last block.
6. If there are redundant equations: if these equations no longer contain any unknown variable they can be removed, otherwise produce an

error message and break.

7. If there are still missing equations, provide dummy equations and apply Tarjan's algorithm to the last block.
8. Again apply the continuity assumption to the last block. This time, derivatives and algebraic variables, which are unknowns in the last block, are assumed to keep their values and turned into known variables.
9. Remove redundant equations or report errors as before.

The idea of applying the continuity assumption in two steps, first for the state variables and then for the derivatives, is illustrated by the following example. Assume that one equation is $\dot{x} + x + y = 0$, where y is affected by the event and computed from some other equation and \dot{x} and x only appear in this equation. The equation and the unknowns \dot{x} and x will appear in last block of the first application of Tarjan's algorithm. Only x is now assumed to keep its value and is regarded as a known value. When Duff's algorithm is applied for the second time, the equation is paired with the remaining unknown \dot{x} .

The suggested algorithm terminates with an error if the continuity assumption cannot be applied consistently. For example, regard the equations

$$x + y = 0$$

$$y + z + w = 0$$

where x , y , and z are algebraic unknowns of the last block of the second application of Tarjan's algorithm, while w is computed from an equation of some earlier block. Only one equation is missing and when the continuity assumption is applied to x , y , and z , the two equations will be redundant. The second one cannot be removed since we cannot consistently assume that both y and z will remain unchanged by the event, and w will probably change. In this case an error is reported and the user has to provide additional equations in the event definition. On the other hand, if w was a state variable that was removed as an unknown when continuity assumption was applied in the previous step, then it could be consistently assumed that x , y , and z remain unaffected by the event. In this case the redundant equations can be removed.

It is sometimes an advantage if models with over determined restart problems are accepted. For this reason, redundant equations should not result in errors. Instead, they can be evaluated during simulation and checked for consistency.

The result obtained so far is a BLT ordering of all the equations relevant to solve the restart problem of the particular event type. Sometimes

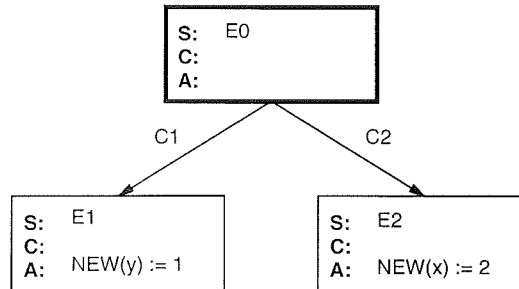


Figure 8.5 Event graph with conditional propagation. C1 and C2 are Boolean conditions determining if the root event E0 is propagated to E1 and E2.

it is desirable to obtain a triangular form, so that each unknown can be solved separately, the next step is to try manipulating each block and to solve it symbolically. This may be possible if the unknowns appear linearly and if the block is small enough. Remaining blocks, that cannot be reduced, are checked for the kind of unknown variables that must be solved from the block. If a block contains unknowns of type real only, then a numerical static equation solver can be utilized at simulation time, to solve the equations. If the block contains only discrete variables or a mix of discrete and real variables, then the restart problem is hard (see also the discussion in Chapter 6). One possibility is to use tearing, i.e., to exclude some of the unknowns, so that the problematic block splits into smaller blocks or become triangular [Elmqvist and Otter, 1994]. The restart problem must then be solved by iterations with guessed initial values for the excluded unknowns. Another possibility is to give an error message and force the user to reformulate the problematic event definitions.

Root events with conditional propagation

For root events with unconditional propagation it is reasonable to compile the restart problem associated with the event in advance, before the simulation is started. This is because each root event only results in one restart problem. However, if the propagation tree contains conditional links, each conditional link gives two alternative restart problems. This is because each condition affects the set of discrete equations involved in the event firing, and each different set of equations represents a different restart problem. If a root event tree contains n conditional links, then there are potentially 2^n different restart problems associated with the root event. An example is shown in Figure 8.5. If the event conditions E1 and E2 are independent, then the root event E0 may result in four different

restart problems: one where nothing is changed, one where y is updated, one where x is updated, and one where both x and y are changed.

Because of the combinatorial explosion of potential restart problems, it is not in general efficient to compile all of them in advance. Many of all the possible restart problems may not occur at all when the model is simulated. In this case it is better to defer compilation until simulation time, when the conditions are evaluated and a particular restart problem can be selected. Naturally, the compiled restart problem is saved and reused if the same situation occurs again or if the simulation is repeated.

If several propagation conditions in a root event tree are identical or each other's Boolean complements, then the number of possible restart problems is reduced. For some modeling methodologies this may be a common situation that is worth considering in the analysis.

Compiling the event conditions

Event conditions are Boolean expressions depending on discrete or continuous model variables. Each condition is associated with an event type. The event occurs as soon as the condition changes from false to true. It is useful to separate the event conditions into two groups: conditions which only depend on discrete variables and conditions which depend on continuous as well as discrete variables. The former group, called *discrete conditions*, needs to be evaluated only after some other event has been fired. Conditions in the other group are called *continuous conditions* and they have to be evaluated along with the continuous time solution.

Discrete conditions need no particular discussion. They are maintained by the discrete event simulator and evaluated when necessary. When a condition evaluates to true the corresponding event becomes enabled and scheduled for firing. The continuous conditions, however, need special treatment. A Boolean event condition needs to be converted into a smooth function that can be evaluated along the continuous time solution. The function should be constructed so that it has a zero where the Boolean condition changes from false to true. The integration algorithm can then be designed to locate the zero accurately and efficiently.

OmSim is translating a Boolean event condition so that the event occurs when a zero up-crossing is detected. For example, the event condition $x > x_{max}$ is translated into the continuous event function $x - x_{max}$. See Figure 8.6. Most numerical software is designed to detect any zero crossing and the decision to associate events with up-crossings in OmSim is arbitrary.

A Boolean event condition is an expression with the Boolean operators AND, OR, and NOT, the relational operators $<$, \leq , $>$, \geq , $=$, and \neq . The

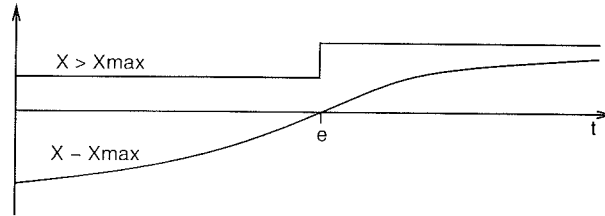


Figure 8.6 Boolean event function and a corresponding continuous one. An event is detected at the up-crossing root, at time e .

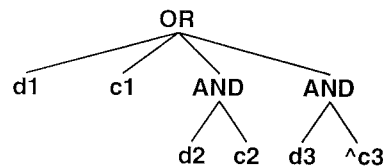


Figure 8.7 An event condition on normal form. The leaf nodes labeled with a 'd' indicate discrete conditions, while leaf nodes labeled with a 'c' indicate continuous conditions.

event condition may also include the *transition operator* \uparrow . The expression is transformed into a normal form using the rules of Boolean algebra. The normal form consists of an *or* expression with a number of *terms*. Each term is continuous relation, a discrete Boolean expression, or an *and* expression with a discrete Boolean *factor* and a continuous relation *factor*. The normal form is illustrated in Figure 8.7. The purpose of the normal form is to separate parts depending only on discrete variables from parts depending on continuous time variables. Each term of the root *or* expression constitutes a separate firing condition for the event. The continuous terms are translated into separate event functions as described above. Each *and* term with a discrete and a continuous factor is then translated into the continuous event function

$$\text{if } d \text{ then } c \text{ else } -1$$

where d is a the discrete factor and c is the continuous factor.

A continuous term or factor may be a compound of several relations depending on different continuous time variables. Regard for example

$$(x > x_{max}) \text{ and } (y > y_{max})$$

where x and y are continuous time variables. It is possible to form a single

Chapter 8. Model Manipulation

continuous event function using the min function:

$$\min(x - x_{\max}, y - y_{\max})$$

However, this may cause numerical problems for the root finder, since the derivative is not continuous even if x and y both have continuous derivatives. A better way, but more complicated, is to introduce two additional discrete Boolean variables and two additional events, one for each continuous condition.

The transition operator, \uparrow , is used for indicating that the event is enabled and fired only once, when the condition changes from false to true. Algebraic rules for the transition operator are given in [David and Alla, 1992, p235]. For example, the following distributive laws are used to manipulate event conditions.

$$\uparrow(a \text{ and } b) = (\uparrow a \text{ and } b) \text{ or } (a \text{ and } \uparrow b)$$

$$\uparrow(a \text{ or } b) = (\uparrow a \text{ and } b') \text{ or } (a' \text{ and } \uparrow b)$$

In [David and Alla, 1992] the hypothesis that two independent events never occur simultaneously, is used. This means that $\uparrow a$ and $\uparrow b = \text{false}$, if a and b are independent Boolean variables. It is a problem to determine if a and b are independent. One possibility is to forbid more than one transition operator in a term, and to force the user to reformulate the condition.

Event conditions are divided into two groups during compilation: *invariant conditions* are conditions without the transition operator, while *transition conditions* are event conditions with the transition operator. Invariant conditions enable the event unconditionally when the event function is positive. Transition conditions enable the event once each time the event condition changes from negative to positive. During simulation, every transition condition is associated with a Boolean status flag telling if the condition is currently regarded as negative or positive.

8.3 Detecting continuous events

Continuous time event conditions must be evaluated along with the continuous time solution. This means that the integration routine and the event detection algorithm are operating in close connection. DASSRT is a variant of the DAE solver DASSL [Brenan *et al.*, 1989] that has a built-in root finder. Brankin *et al.* [Shampine *et al.*, 1991] have developed a

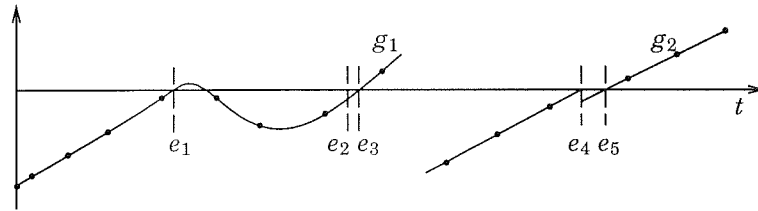


Figure 8.8 Event e_1 is not detected since the event function has two roots that happen to occur in the same integration step. Event e_3 is a spurious event because event e_2 was localized with a finite tolerance. Event e_5 is also a spurious event because a numerical error was introduced when e_4 was fired.

package of subroutines for event detection that can be used together with several standard integration routines. Event location is also one of the main topics in [Cellier, 1979].

All event location methods are based on the same simple idea. After the integration algorithm has completed a step and computed a new solution point, the event functions are evaluated and checked for sign changes between the previous solution point and the new one. If a sign changes, this indicates that one or more of the event functions have roots in the interval. Then the next task is to determine the first root and to localize it. The location of the first root is usually determined by a search combined with some standard root solving algorithm, e.g., some variant of regula falsi.

Though the basic idea of locating events is simple, there are many things that can fail. For example, if we look for sign changes of the event function by examining the end points of each integration step, event functions with more than one root in an interval may be missed. It is also a possibility to detect spurious events at the beginning of an interval for event functions that starts at zero or close to zero; see Figure 8.8 for an illustration of the problems. It should also be noted that localizing several roots which are close to each other, is an ill-conditioned problem. A small error in the event function may result in two separate roots, a second order root, or no root at all.

If the method is modified to avoid spurious events, then there is the risk that some real events are excluded as well. In the OmSim environment, it is very important that the event location algorithm is robust and can handle most cases correctly. It is also important that warnings are issued when borderline cases are detected so that the user can modify his model or at least become aware that the simulation results may be incorrect.

OmSim is using DASSRT with the built-in root finder or the routines by Brankin et al. in combination with Radau5. The DASSRT implementation has a problem if it is used directly. It cannot be started if an event condition is close to zero. This is a common situation. The method by Brankin et al. is discussed in the following. It is presented in [Shampine *et al.*, 1991] and consists of a set of subroutines and a main driver routine called ALEVNT. Even though ALEVNT is not used directly in OmSim, the name will be used as a collective name for the package of subroutines.

ALEVNT is designed to be used with integration algorithms that can provide solution points between the steps. This is often called *dense output* and is provided by many modern methods like DASSRT, Radau5, and DOPRI(4)5 [Hairer *et al.*, 1987]. Polynomial interpolation is used to provide solutions between integration points. The interpolants are often provided by the integration method at almost no extra cost. ALEVNT constructs interpolation polynomials for representing the event functions between solution points. Then it uses Sturm sequences to check if any root is present in the interval [Stoer and Bulirsch, 1980]. If a root is found, interval bisections and Sturm sequences are used to find an interval where the first root is located. Finally it uses a standard bisection and secant algorithm [Press *et al.*, 1992] to localize the root precisely. This method is efficient and capable of detecting multiple roots within an integration interval.

ALEVNT is efficient since it uses polynomial approximations of the event functions. If the order of the polynomials are the same in the integration method and in ALEVNT, then the approximations of the event functions are correct if the event functions are linear in the unknown variables. Since OmSim accepts also non-linear event functions, ALEVNT is actually misused. Remedies are discussed in the following.

Some notation is now introduced. Let $y(t)$ be the true solution of the DAE problem and let $\dot{y}(t)$ be its derivative. Let $\bar{y}(t)$ and $\dot{\bar{y}}(t)$ be the solution at $t_{k-1} \leq t \leq t_k$ provided by the integration method after step k . Let $g = g(t, \bar{y}(t), \dot{\bar{y}}(t))$ be the event function based on the computed solution. Finally, let $\bar{g}(t)$ be the polynomial approximation of $g(t)$ used by ALEVNT. In general $\bar{g}(t) \neq g(t)$ but if the event functions are not too nasty, then we can expect that $\bar{g}(t) \approx g(t)$. There are still different possibilities to use ALEVNT:

1. Use ALEVNT for the linear event functions and use some other method based on $g(t)$ for the others
2. Convince the user only to use linear event functions
3. Introduce auxiliary variables that are solved by the integrator that makes the event functions linear

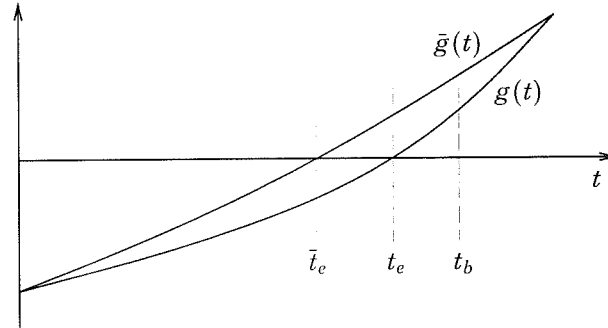


Figure 8.9 An event first detected and localized approximately using $\bar{g}(t)$ and ALEVNT, and then localized exactly using $g(t)$

4. Use a higher order polynomial for $\bar{g}(t)$
5. First locate the event approximately using ALEVNT, then iterate using some standard bisection or secant method based on $g(t)$.

The last method has been implemented and tested in OmSim. It gives maximum efficiency for well-behaved event functions and it is robust with more difficult ones. Figure 8.9 illustrates the method. The event first located by ALEVNT occurs at time \bar{t}_e . This time is used as one side (t_a) of a bracket for $g(t)$, i.e., a time interval where $g(t)$ has a sign change. The other bracket (t_b) is obtained by a “double Newton-Raphson step”, where $\dot{g}(t)$ is obtained from ALEVNT and used instead of $\dot{g}(t)$:

$$t_b = t_a - 2 \frac{g(t_a)}{\dot{g}(t_a)}.$$

For very peculiar event functions it may occur that a bracket for $g(t)$ is not found in this way. In this case OmSim gives up with an error message but it would also be possible to apply some search method.

After a bracket for $g(t)$ is found, some iterative method can be applied to narrow the bracket until the desired tolerance is achieved. The tolerance can either be specified in terms of $|t_b - t_a|$ or in terms of $|g(t)|$. The later is of course affected by the scaling of the event function. This can be an advantage for the conscious user since different accuracy can be achieved for different conditions. On the other hand, the tolerance specified in terms of time is not affected by scaling and is more robust.

Spurious events

Event functions that start close to zero, for example when integration is restarted after an event, may cause spurious events. Unfortunately such

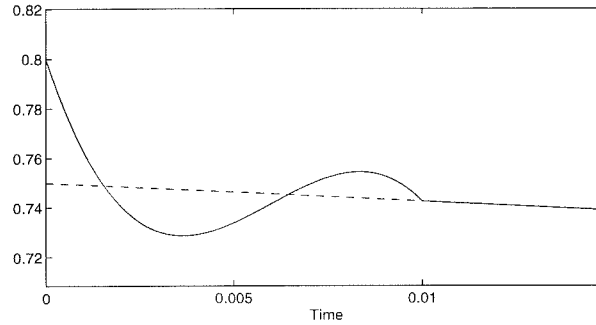


Figure 8.10 Interpolated result returned from Radau5 during the first step. The first solution point is at $t = 0.01$. The dashed line shows the correct solution when the integrator is started with consistent initial values. The solid line results from starting the integrator with initial values slightly off consistent.

event functions are rather common. As shown in Figure 8.8, the spurious events may occur because the event function is deviating from an exact zero because of numerical errors. Even if we make sure to start with the event function slightly positive, spurious events may occur due to bad behavior of the interpolated solution returned by the integrator. Figure 8.10 shows the solution for the first step when Radau5 is restarted with an initial state that is not exactly consistent. The slightly inconsistent starting point is accepted by Radau5 and does not affect the accuracy of the first solution point, but it affects the interpolated solution during the first step considerably. In order to avoid trouble with spurious events, ALEVNT is not used for the first step after the integration method is restarted. This can be justified by the fact that integration methods are usually started with a default step size which is considerably shorter than what is motivated by the required accuracy. Instead of ALEVNT, a simpler method is used that fails to detect an even number of roots in the interval. The method used for the first step is as follows.

Each event function is classified if it is negative, zero, or positive at the initial point and at the first solution point. Function values that are absolutely less than a certain tolerance ε are classified as zero. The tolerance should be set slightly larger than the event location tolerance and the accuracy of the initial value computation. Table 8.3 gives all possible combinations of event function values at the initial point (t_0) and the first solution point (t_1). The numbers in the table refer to the

8.3 Detecting continuous events

t_0	t_1	Comment
-	-	No root
-	0	Event at t_1 (1)
-	+	Root in first step (2)
0	-	No root
0	0	No root (3)
0	+	Possible event at t_0 (4)
+	-	Root but no event (5)
+	0	No event (6)
+	+	No root

Figure 8.11 Combination of event function values at the initial point t_0 and at the first solution point t_1 . The numbers in the third column refer to comments listed in the text.

following comments.

1. A root is present close to t_1 . Iterate with initial steps of different length until desired precision.
2. A root is located between t_0 and t_1 . Iterate with initial steps of different length until desired precision.
3. The event function remains close to zero. This is an undesirable property of the event function, but it is sometimes hard to avoid. Issue a warning and treat the next step as an initial step.
4. The event function becomes positive. If this is an invariant condition, then the step should be rejected and the event fired at t_0 . If this is a transient condition, no event occurs and integration is resumed.
5. Down-crossing root but no event. Update event function status and resume integration.
6. No event but redo the initial step with different step size, in order to move away from the zero. This is necessary in order to update the event function status properly.

Event functions that remain close to zero for longer periods should be avoided in the model. It is not appropriate to apply ALEVNT until all event functions have clearly moved away from zero. If it is known from the model that a particular event function remains zero for periods, then the model should be modified so that the period is explicitly represented as a discrete mode, and the event function should be deactivated in that mode. For proper models, event functions that remain zero after the first step are rare. It may occur either if the event function is not properly scaled, or if it has a high order root at the initial point. Event functions

with high order roots should also be avoided since the location problem is ill-conditioned. This justifies the strategy of comment 3.

Another strategy to handle event functions that start close to zero is used in DYMOLA and described in [Elmqvist *et al.*, 1993]. When an event function g causes an event and if g is close to zero when the integration restarts, then g is first replaced by two other event functions, $g + \textit{eps}$ and $g - \textit{eps}$, where \textit{eps} is a small constant quantity. This means that g is considered to be zero as long as it remains within a small interval close to zero, and events will occur when it leaves the interval. After that, the original event function g is restored. The method is probably better at handling event functions that remain zero for longer periods in time. However, such event functions should be avoided in any case. A drawback of the method is that two event functions are needed for each event condition in the model. The method also wastes computations by localizing the exact time when the event function leaves the region $\pm \textit{eps}$. Experimental studies with practical models must be performed to compare the methods.

8.4 Summary

A general overview of the symbolic model manipulations applied to OHM representations was given. The continuous time and the discrete event parts of the model were discussed separately. Analysis and manipulation of the continuous time parts mainly consist of the following steps:

1. Check that the model has the right number of equations so that each unknown variable can be determined.
2. Reduce size and index if necessary, and do other symbolic manipulations in order to simplify the solution.
3. Transform the model into a format required by the chosen numerical integration algorithm and generate simulation code.

These model manipulation steps have been discussed in some detail.

Main steps in analyzing the discrete event model were discussed. The steps include analysis of event propagation in order to determine a minimal set of fundamental discrete event types. For each event type, the reinitialization problem is analyzed and transformed into event simulation code. The continuous time event conditions are also analyzed and transformed into a representation suitable for the numerical algorithms. Finally, a general discussion on discrete event detection in combination with numerical integrators was given.

9

Conclusions

Models are important in all areas of engineering as representations of real processes and potential designs. They are used for analysis, design, and simulation. Different engineering disciplines have developed different traditions in modeling. The computer tools that have been developed are also specialized either for a particular engineering domain or for a particular task. In the fields of control engineering and process design, there is a need to represent mixed domain models in a form that can be used for different tasks such as numerical simulation and symbolic analysis. There is also a need to represent combined continuous time and discrete event models. Since good models are hard to develop, it is very important that models and model components can be reused.

This thesis defined two model representations: a high-level modeling language called Omola and low-level hybrid model formalism called OHM. It also presented an integrated environment for model development and simulation called OmSim.

Omola is defined with a syntax which is intended to be understandable by people and computers. The size and the complexity of real systems are major difficulties when models are developed. For this reason, Omola is provided with powerful concepts for creating abstractions and for hierarchical decompositions of models. Concepts like classes and inheritance from the field of object-oriented programming are used to support reuse of models and model components.

Models represented as classes make it possible create model libraries where definitions are organized in inheritance hierarchies. General model definitions are represented by separate classes which can be refined and specialized in multiple levels. This helps to organize the model libraries and increases the possibilities for reuse.

Model behavior is represented in Omola as differential and algebraic equations and discrete events. Component interaction is defined by con-

Chapter 9. Conclusions

nections that relate terminal interfaces. The meaning of a connection is defined in terms of equations and event propagations, and it depends on terminal attributes. Composite models can be defined and displayed using graphical editors.

Omola is defined as two separate layers. A basic layer has concepts for general, object-oriented definitions of data structures. A second layer — the model representation layer — is defined by a set of predefined classes with defined meanings as model components. The layered approach makes it easier to extend the language and to include new modeling concepts.

OHM is a model formalism for representing hybrid continuous time and discrete event systems. The representation consists of sets of variables, parameters, equations, event conditions, and event actions. It has a well-defined meaning in terms of mathematics and logic. The main purpose of OHM is to represent behavior so that the model can be analyzed and manipulated algebraically. OHM is an intermediate representation, that serves as a common mathematical framework that can be translated automatically into more specialized representations. For example, one representation may be the code required to simulate a model, another can be the transfer function that represents the linearized behavior at a particular operating point.

The thesis defined the procedure for translating an Omola model into an OHM representation. An intermediate step in this procedure is to create a *model instance* which is an instantiated version of the top-level Omola class defining the model. A model instance preserves the composition hierarchy of the model but removes the inheritance hierarchy. There are simple methods for extracting the OHM representation from a model instance. This done by traversing the model hierarchy and extracting all variables, equations, and event definitions.

An OHM representation can be analyzed algebraically and checked for various structural properties. Well-known graph theoretical methods can be applied to sort equations and to analyze the degrees of freedom in the model and its index. Superfluous variables and equations can be eliminated to reduce the size of the model. Proper models can be translated into more specific representations suitable for numerical simulation. Methods for this were discussed in the thesis, though they are mainly based on previously known results.

OmSim is an implementation of an environment supporting development and simulation of Omola models. OmSim organizes model libraries permanently stored in the computer's file system as Omola code. Models are parsed and loaded into an internal model store. OmSim contains tools

for presenting and defining models graphically as component diagrams. It also contains tools for checking consistency of connections. Further more, OmSim instantiates models, manipulates them and creates simulation code for a set of different numerical integration routines. Instantiated models can be simulated interactively.

The thesis documented the architecture of OmSim and discussed some of the important design issues. The architecture of OmSim was compared with a suggested standard for open architectures for computer aided control engineering.

Applications

Omola and OmSim have been used in several application studies. These include a modeling and simulation study of the Nordic power net [Persson, 1992], a Petri-Net modeling and simulation study of production systems [Nilsson, 1991], a development of an object-oriented modeling methodology for chemical processes [Nilsson, 1993], a modeling and simulation of a heat exchanger system [Ericsson and Östberg, 1993, Mattsson *et al.*, 1994], a study of models of dry friction [Eborn, 1994], a model library for multibody mechanical systems [Anell, 1994], and object-oriented modeling of flows [Ramos González, 1994].

The need for a new model representation standard

The representation of dynamic models is central in software tools for control and process engineering. A complete environment for model representation, development, analysis, simulation, and engineering design, is a huge piece of software. It cannot be realized as monolithic program supported by a single vendor. For this reason, it is of vital importance that representations are standardized so that different vendors and scientists can contribute to an integrated environment of independent tools.

The main contribution of this thesis consists of the model representations Omola and OHM. They can be viewed as a proposal for a standard for representation of hybrid models [Mattsson and Andersson, 1991, Mattsson, 1993].

A high-level language, such as Omola, is important since it makes it possible to structure models in order to make them understandable and reusable. With a standardized language, component manufacturers are stimulated to provide product documentation in terms of dynamic models that could be used directly together with models of other components. This would simplify system engineering and design drastically. Omola can also be used as an intermediate representation which can be used in many engineering fields. With a standardized common representation it

Chapter 9. Conclusions

is possible to combine mixed domain models developed in different high-level modeling tools.

A formal representation, such as OHM, is important since it has a well-defined meaning expressed in mathematical and logical terms. When this representation is standardized, different tools for analysis, simulation, and design can be developed independently and used in an integrated environment.

Future work

Omola may be extended in various directions.

Regular structures. A convenient syntax may be introduced for specifying models with arrays of similar components connected in a regular way. This has been suggested in [Nilsson, 1993] and [Mattsson *et al.*, 1994]. The extension requires that the syntax of Omola is augmented and that the instantiation procedures are updated.

Abstract classes. The discussion at the end of Chapter 4 suggested a way of defining local classes that are not automatically instantiated as model components. This is a simple, easily implemented extension.

Strict encapsulation. There may be reasons for providing more strictly encapsulated model components. For example, it may be useful to forbid access to model attributes that are not part of the model's interface. This is a modest extension of the language and the variable resolve procedures in the modeling environment.

Formalized assumptions. It may be useful to document models more formally. For example, assumptions about range of validity may be added to the model. They can then be checked automatically during simulation.

Sequences and external functions. Some discrete event actions are most conveniently defined as a sequence of operations. It would be possible to extend Omola to allow an event action to be defined as a sequential algorithm executed as a result of the event. However, in order to avoid that Omola grows into a complete sequential programming language, it may be better to provide an interface for external functions. Sequential event actions can then be coded in an ordinary programming language and dynamically linked to OmSim.

A data model for OHM. If OHM is going to be used as common representation in integrated environment it must be given a precise definition as a language or a data structure. This can be done either by defining a concrete syntax for OHM or by defining OHM in some standardized data definition language. OHM may also provide an interface to tools for symbolic manipulations.

Models with variable structure. It is possible to extend Omola and OHM to represent models with variable structure. This means models where parts of the system are added and removed from the model as a result of the system's behavior. This would enhance the modeling power, in particular for discrete event systems. The extension means that language concepts for introducing and initializing components dynamically must be defined. OHM also needs to be extended to represent sets of equations that can be activated or deactivated as a result of discrete events.

Extensions to the modeling environment. It is easy to imagine many tools that are useful to support model development and analysis. The following gives a few examples:

- An interface to a proper database where different versions of models can be associated with simulation and analysis results.
- An interactive tool to operate on Omola models and make it possible to modify hierarchical models, for example, to aggregate or disaggregate components and to introduce or remove hierarchical levels.
- A tool for presenting and operating on OHM representations so that the user can understand the algebraic properties of the model and direct the symbolic manipulations.
- A code generator that translates discrete event parts of an OHM into executable real-time procedures. This may be used for automatic implementation of control systems represented in Omola.

10

References

- Abelson, H. and G. J. Sussman (1985): Structure and Interpretation of Computer Programs. MIT Press.
- AFCET (1977): "Normalisation de la représentation du cahier des charge d'un automatisme logique." Technical Report, France. Final report of the AFCET Commission.
- Aho, A. V. and J. D. Ullman (1977): Principles of Compiler Design. Addison-Wesley.
- Alur, R., C. Courcoubetis, T. A. Henzinger, and P.-H. Ho (1993): "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems." In Grossman *et al.*, Eds., Hybrid Systems, Lecture Notes in Computer Science 736, pp. 209–229. Springer-Verlag.
- Alur, R. and D. Dill (1992): "The theory of timed automata." In de Bakker *et al.*, Eds., Real-Time: Theory in Practice, Lecture Notes in Computer Science 600, pp. 45–73. Springer-Verlag.
- Andersson, M. (1989a): "An object-oriented language for model representation." In 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD), Tampa, Florida.
- Andersson, M. (1989b): "An object-oriented modeling environment." In Iazeolla *et al.*, Eds., Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation, 1989 European Simulation Multiconference, Rome, pp. 77–82. The Society for Computer Simulation International.
- Andersson, M. (1989c): "Omola—An object-oriented modelling language." Report TFRT-7417, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Andersson, M. (1990): Omola—An Object-Oriented Language for Model Representation. Lic Tech thesis TFRT-3208, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Andersson, M. (1993): "OmSim and Omola tutorial and user's manual." Report ISRN LUTFD2/TFRT-7504--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Andersson, M., S. E. Mattsson, and B. Nilsson (1991): "On the architecture of CACE environments." In Preprints of the IFAC Symposium Computer Aided

- Design in Control Systems, CADCS'91, pp. 63–68, Swansea, UK.
- Anell, P. (1994): "Modeling of multibody systems in Omola." Master thesis ISRN LUTFD2/TFRT-5516--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Årzén, K. E. (1987): Realization of Expert System Based Feedback Control. PhD thesis TFRT-1029, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Årzén, K.-E. (1989): "An architecture for expert system based feedback control." *Automatica*, **25:6**, pp. 813–827.
- Åström, K. J. and W. Kreutzer (1986): "System representations." In Proc. IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD), Arlington, Virginia.
- Åström, K. J. and B. Wittenmark (1990): Computer-controlled systems. Prentice-Hall.
- Balemi, S. (1992): Control of Discrete Event Systems: Theory and Application. PhD thesis, Automatic Control Laboratory, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland.
- Balemi, S., G. J. Hoffmann, P. Gyugy, H. Wong-Toi, and G. F. Franklin (1993): "Supervisory control of a rapid thermal multiprocessor." *IEEE Transactions on Automatic Control*, **38:7**.
- Barker, H. A., M. Chen, P. W. Grant, C. P. Joblin, and P. Townsend (1993): "Open architecture for computer-aided control engineering." *IEEE Control Systems*, **13:2**.
- Barstow, D. R., H. E. Shrobe, and E. Sandewal (1984): Interactive Programming Environments. McGraw-Hill.
- Barton, P. I. and C. C. Pantelides (1991): "The modeling and simulation of combined discrete/continuous processes." In Proc. from Process System Engineering '91, Montebello, Canada.
- Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard (1973): Simula Begin. Auerbach, Philadelphia, Pa.
- Booch, G. (1983): Software Engineering with Ada. Benjamin/Cummings.
- Brenan, K. E., S. L. Campbell, and L. R. Petzold (1989): Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations. North-Holland, Amsterdam.
- Cellier, F. E. (1979): Combined Continuous/Discrete Systems Simulation by Use of Digital Computers: Techniques and Tools. PhD thesis, ETH.
- David, R. and H. Alla (1992): Petri Nets & Grafcet: Tools for modelling discrete event systems. Prentice Hall.
- Driankov, D., H. Hellendoorn, and M. Reinfrank (1993): An Introduction to Fuzzy Control. Springer Verlag, Berlin Heidelberg.
- Duff, I. S., A. M. Erisman, and J. K. Reid (1986): Direct Methods for Sparse Matrices. Clarendon Press, Oxford.
- Duff, I. S. and J. K. Reid (1978): "An implementation of tarjan's algorithm for the block triangularization of a matrix." *ACM Transactions on Mathematical Software*, **4:2**, pp. 137–147.

Chapter 10. References

- Eborn, J. (1994): "Modelling and simulation of an industrial control loop with friction." Master thesis ISRN LUTFD2/TFRT-5501--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Elmqvist, H. (1978): A structured Model Language for Large Continuous Systems. PhD thesis ISRN LUTFD2/TFRT-1015--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Elmqvist, H. (1994): Dymola – Dynamic Modeling Language, User's Manual. Dynasim AB.
- Elmqvist, H., K. J. Åström, T. Schönthal, and B. Wittenmark (1990): Simnon User's Guide. SSPA, Göteborg, Sweden.
- Elmqvist, H., F. E. Cellier, and M. Otter (1993): "Object-oriented modeling of hybrid systems." In Proceedings of the ESS'93 European Simulation Symposium, Delft.
- Elmqvist, H. and M. Otter (1994): "Methods for tearing systems of equations in object-oriented modeling." In Guasch and Huber, Eds., Proceedings of the Conference on Modelling and Simulation ESM'94, pp. 326 – 332, Barcelona, Spain.
- Ericsson, M. and P. Östberg (1993): "Dynamisk provning av värmeväxlersystem," (Dynamical testing of heat exchanger systems). Master thesis ISRN LUTFD2/TFRT-5490--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Forbus, K. D. (1988): "Qualitative physics: Past, present, and future." In Weld and de Kleer, Eds., Readings in Qualitative Reasoning About Physical Systems. Morgan Kaufmann, San Mateo, California.
- Glynn, P. W. (1989): "A GSMP formalism for discrete event." Proceedings of the IEEE, **77:1**, pp. 14–23.
- Goldberg, A. (1983): Smalltalk-80: The Interactive Programming Environment. Addison-Wesley.
- Goldberg, A. (1984): "The influence of an object-oriented language on the programming environment." In Barstow *et al.*, Eds., Interactive Programming Environments, pp. 141–174. McGraw-Hill.
- Grübel, G. (1992): "AnDeCS: a concurrent control engineering project." In Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design, CADCS '92, Napa, California.
- Grübel, G. and H.-D. Joos (1991): "RAPS and RSYST – two complementary libraries for concurrent control engineering." In Barker, Ed., Prep. 5th IFAC Symp. Computer Aided Control Systems — CACSD'91, pp. 101–106, Swansea, U.K. IFAC.
- Gustafsson, K. (1993): "Object-oriented implementation of software for solving ordinary differential equations." Scientific Programming, **2**, pp. 217–225.
- Hairer, E., C. Lubich, and M. Roche (1989): The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods., volume 1409 of Lecture Notes in Mathematics. Springer-Verlag.
- Hairer, E., S. P. Nørsett, and G. Wanner (1987): Solving Ordinary Differential Equations I – Nonstiff Problems., volume 8 of Springer Series in Computational

- Mathematics. Springer-Verlag.
- Harbison-Briggs, K. A., H.-D. Joos, and J. M. Maciejowski (1993): "EXPRESS data definitions for control engineering." In Preprints of the 12th IFAC World Congress, volume 8, pp. 377–380, Sydney, Australia.
- Harmon, P. and D. King (1985): Expert Systems. John Wiley & Sons, Inc.
- Heymann, M. (1990): "Concurrency and discrete event control." IEEE Control Systems Magazine, **10:4**, pp. 103–112.
- Ho, Y.-C. and X.-R. Cao (1991): Perturbation Analysis of Discrete Event Dynamic Systems. Kluwer Academic Publishers.
- Hopcroft, J. E. and J. D. Ullman (1979): Introduction to Automata Theory, Languages, and Computation. Addison-Wesley.
- Hope, S., P. J. Fleming, and J. G. Wolff (1991): "Object-oriented database support for computer-aided control system design." In Barker, Ed., Prep. 5th IFAC Symp. Computer Aided Control Systems — CACSD'91, pp. 200–204, Swansea, U.K. IFAC.
- Howard, R. A. (1971): Dynamic Probabilistic Systems., volume II. John Wiley & Sons, Inc.
- IEC (1988): "IEC 848, Preparation of function charts for control systems." Technical Report 848:1988, International Electrotechnical Commission.
- Johansson, R. (1993): System Modeling and Identification. Prentice Hall, Englewood Cliffs, New Jersey.
- Kailath, T. (1980): Linear Systems. Prentice-Hall.
- Karnop, D. and R. Rosenberg (1975): System Dynamics: A Unified Approach. John Wiley & Sons.
- Kim, T. G. and B. P. Zeigler (1991): "The DEVS-scheme modelling and simulation environment." In Luker and Schmidt, Eds., Knowledge Based Simulation, Methodology and Application, number 4 in Advances in Simulation, pp. 20–35. Springer-Verlag.
- Koenig, H. E., Y. Tokad, H. Kesavan, and H. Hedges (1967): Analysis of Discrete Physical Systems. McGraw-Hill Book Company.
- Kröner, A., P. Holl, W. Marquardt, and E. D. Gilles (1990): "DIVA - an Open Architecture for Dynamic Simulation." Computers & Chemical Engineering, **14:11**, pp. 1289–1295.
- Kuijper, M. (1994): First-order Representations of Linear Systems. Systems & Control: Foundations & Applications. Birkhäuser.
- Kuipers, B. (1986): "Qualitative simulation." Artificial Intelligence, **29**, pp. 289–338.
- Larsson, J. E. (1992): Knowledge-Based Methods for Control Systems. PhD thesis ISRN LUTFD2/TFRT-1040--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Linton, M., P. R. Calder, and J. R. Vlissides (1987): "Interviews: A C++ graphical interface toolkit." Technical Report, Computer Systems Laboratory, Stanford University.
- Lund, P. C. (1992): An Object-Oriented Environment for Process Modeling and Simulation. PhD thesis, University of Trondheim.

Chapter 10. References

- Maffezzoni, C., R. Girelli, and P. Lluka (1994): "Object-oriented database support for modular modelling and simulation." In guasch and Huber, Eds., Proc. European Simulation Multiconference, pp. 354–361, Barcelona, Spain. SCM.
- Maler, O., Z. Manna, and A. Pnueli (1992): "From timed to hybrid systems." In de Bakker *et al.*, Eds., Real-Time: Theory in Practice, Lecture Notes in Computer Science 600, pp. 447–484. Springer-Verlag.
- Marquardt, W. (1992): "An Object-oriented Representation of Structured Process Models." In European Symposium on Computer Aided Process Engineering – 1.
- The MathWorks, Inc. (1992): Simulink User's Guide.
- The MathWorks, Inc. (1993): MATLAB User's Guide.
- Mattsson, S. E. (1993): "Towards a new standard for modelling and simulation tools." In Iversen, Ed., SIMS'93, Applied Simulation in Industry — Proceedings of the 35th SIMS Simulation Conference, pp. 1–10, Trondheim, Norway. SIMS, Scandinavian Simulation Society, c/o SINTEF Automatic Control. Invited paper.
- Mattsson, S. E. and M. Andersson (1991): "Towards a universal modelling language." In ISA/91 Anaheim Proceedings. Instrument Society of America. Invited paper.
- Mattsson, S. E., M. Andersson, and K. J. Åström (1993): "Object-oriented modelling and simulation." In Linkens, Ed., CAD for Control Systems, chapter 2, pp. 31–69. Marcel Dekker Inc, New York.
- Mattsson, S. E., M. Ericson, and P. Östberg (1994): "An object-oriented model of a heat-exchanger unit." In Proceedings of the European Simulation Multiconference, ESM'94, pp. 297–303, Barcelona, Spain. SCS, The Society for Computer Simulation.
- Mattsson, S. E. and G. Söderlind (1990): "A new technique for solution of high index differential-algebraic equations." In The 1990 Conference on the Numerical Solution of Ordinary Differential Equations, Helsinki, Finland.
- Mattsson, S. E. and G. Söderlind (1993): "Index reduction in differential-algebraic equations using dummy derivatives." SIAM Journal of Scientific and Statistical Computing, **14**:3.
- Mitchel and Gauthier Associates, Concord, Massachusetts (1986): Advanced Continuous Simulation Language (ACSL), Reference Manual.
- Murata, T. (1989): "Petri nets: Properties, analysis and applications." Proceedings of the IEEE, **77**:4.
- Nagel, L. W. (1975): "SPICE2: A computer program to simulate semiconductor circuits." Technical Report ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA.
- Nerode, A. and W. Kohn (1993): "Model for hybrid systems: Automata, topologies, controllability, observability." In Grossman *et al.*, Eds., Hybrid Systems, Lecture Notes in Computer Science 736, pp. 317–356. Springer-Verlag.
- Newton, I. (1687): *Philosophiae naturalis principia mathematica*. Cambridge.
- Nilsson, B. (1989): Structured Modelling of Chemical Processes—An Object-Oriented Approach. Lic Tech thesis TFRT-3203, Department of Automatic

- Control, Lund Institute of Technology, Lund, Sweden.
- Nilsson, B. (1993): Object-Oriented Modeling of Chemical Processes. PhD thesis ISRN LUTFD2/TFRT-1041--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Nilsson, H. (1991): "Implementation of Petri-net and Grafset primitives in Omola and modelling of Markov-processes." Master thesis TFRT-5452, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Otter, M. (1992): "DSBlock: a neutral description of dynamic systems." Open CACSD Electronic Newsletter, **1:3**.
- Pantelides, C. (1988): "The consistent initialization of differential-algebraic systems." *SIAM J. Sci. Statist. Comput.*, **9**, pp. 213 – 231.
- Persson, M. (1992): "Modellering och simulering av kraftnät i Omola," (Modelling and simulation of power systems in Omola). Master thesis TFRT-5455, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Piela, P. C. (1989): ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis. PhD thesis, Carnegie-Mellon University, Pittsburg, Pennsylvania.
- Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992): *Numerical Recipes in C*. Cambridge University Press, second edition.
- Ramadge, P. and W. Wonham (1989): "The control of discrete event systems." *Proc. of the IEEE*, **77:1**, pp. 81–98.
- Ramos González, J. J. (1994): "Object-oriented modelling of flows in process systems." Report ISRN LUTFD2/TFRT--7521--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.
- Räumschüssel, S., A. Gerstlauer, E. D. Gilles, B. Raichle, M. Zeitz, and W. Marquardt (1993): "An architecture of a knowledge-based process modeling and simulation tool." In *International Symposium on Mathematical and Intelligent Models in System Simulation*. IMACS/IFAC.
- Reisig, W. (1982): *Petri Nets: An Introduction*. Springer-Verlag.
- Rothenberg, J. (1989): "The nature of modeling." In Widman *et al.*, Eds., *Artificial Intelligence, Simulation & Modeling*, pp. 75–92. John Wiley & Sons.
- Rumbaugh, J., M. Blaha, W. Premerlain, F. Eddy, and W. Lorensen (1991): *Object-Oriented Modeling and Design*. Prentice-Hall.
- Sargent, R. W. H. and A. W. Westerberg (1964): "SPEED-UP in Chemical Engineering Design." *Transaction Institute in Chemical Engineering*, **42**, pp. 190–197.
- Shampine, L. F., I. Gladwell, and R. W. Brankin (1991): "Reliable solution of special event location problems for ODEs." *ACM Transactions on Mathematical Software*, **17:1**, pp. 11–25.
- Skjellum, A., A. P. Leung, S. G. Smith, R. D. Falgout, C. H. Still, and C. H. Baldwin (1993): "The multicomputer toolbox — first-generation scalable libraries." In *27th Hawaii International Conference on Systems Sciences*.
- Spiby, P. (1992): *ISO/DIS 10303-11 EXPRESS Language Reference Manual*. ISO. TC184/SC4 Documant N151.

Chapter 10. References

- Strauss, J. C., D. C. Augustin, M. S. Fineberg, B. B. Johnsson, R. N. Linebarger, and F. J. Sansom (1967): "The SCi continuous system simulation language (CSSL)." Simulation, dec.
- Stroer, J. and R. Bulirsch (1980): Introduction to Numerical Analysis. Springer-Verlag.
- Strömberg, J.-E. (1994): A mode switching modelling philosophy. PhD thesis 353, Linköping University, Sweden.
- Stroustrup, B. (1986): The C++ Programming Language. Addison-Wesley.
- Tan, C. Y. and J. M. Maciejowski (1989): "DB-Prolog: A database programming environment for cacs." In IEEE Control System Society Workshop on Computer-Aided Control System Design, Tampa, Florida.
- Tarjan, R. E. (1972): "Depth first search and linear graph algorithms." SIAM J. of Comp. 1, pp. 146–160.
- Taylor, J. H., D. K. Frederick, C. M. Rinvall, and H. A. Sutherland (1989): "The GE MEAD computer-aided control engineering environment." In IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD), Tampa, Florida.
- Thoma, J. U. (1975): Introduction to Bond Graphs and their Applications. Pergamon Press.
- Tichy, W. F. (1985): "RCS — a system for version control." Software — Practice & Experience, **15**:7.
- Tittus, M. and B. Egardt (1993): "Controllability and control-law synthesis of linear hybrid systems." Technical Report CTH/RT/I-93/0007, Control Engineering Lab, Chalmers University of Technology, Gothenburg, Sweden.
- Waite, W. M. and G. Goos (1984): Compile Construction. Springer-Verlag.
- Willems, J. C. (1986): "From time series to linear systems — part I. finite dimensional linear time invariant systems." Automatica, **22**:5, pp. 561–580.
- Willems, J. C. (1991): "Paradigms and puzzles in dynamical systems." IEEE Transactions on Automatic Control, **36**:3, pp. 259–294.
- Willems, J. C. (1993): "LQ-control: A behavioral approach." In Proc. of the 32nd IEEE Conference on Decision and Control, volume 4, pp. 3664–3668, San Antonio, Texas. IEEE.
- Winston, P. H. (1984): Artificial Intelligence. Addison-Wesley, 2nd edition.
- Zeigler, B. P. (1976): Theory of Modelling and Simulation. John Wiley & Sons, New York.
- Zeigler, B. P. (1990): Object-Oriented Simulation with Hierarchical, Modular Models. Academic Press, Boston, Mass.

A

Omola Grammar

```
omola-definitions ::=
    /* empty */
    omola-definitions id class-def ;
    omola-definitions id type-declaration ;
    omola-definitions LIBRARY id ;
    omola-definitions USES name-list ;
    omola-definitions block ;
    omola-definitions COMMENT
```

```
class-def ::=
    super-class-def
    super-class-def WITH body END
```

```
super-class-def ::=
    ISA lib-id
```

```
lib-id ::=
    id
    id :: id
```

```
body ::=
    /* empty */
    body body-item ;
    body COMMENT
    body TAG
```

```
body-item ::=
    name-list class-def
```

Chapter A. Omola Grammar

```
name-list type-declaration
reference := expr
expr = expr
reference AT reference
event-handler
```

```
event-handler ::=
    WHEN expr cause-list event-body
```

```
cause-list ::=
    /* empty */
    CAUSE expr-list
```

```
event-body ::=
    /* empty */
    DO actionbody END
```

```
actionbody ::=
    /* empty */
    actionbody name-list type-declaration
    actionbody function-designator := expr ;
    actionbody function-designator ;
    actionbody cond-action
    actionbody expr = expr
    actionbody COMMENT
```

```
cond-action ::=
    IF expr THEN function-designator ELSE function-designator ;
    IF expr THEN function-designator ;
```

```
type-declaration ::=
    TYPE var-kind type-designator
    TYPE var-kind type-designator := expr
```

```
var-kind ::=
    /* empty */
    DISCRETE
    STATIC
```

```
type-designator ::=
```

MATRIX [expr , expr]
ROW [expr]
COLUMN [expr]
REAL
INTEGER
STRING
(name-list)
BOOLEAN
REFERENCE
SYMBOL

expr-list ::=
 expr
 expr-list , expr

expr ::=
 IF expr THEN expr ELSE expr
 expr AND expr
 expr OR expr
 NOT expr
 expr REL-OP expr
 expr ADD-OP expr
 expr MUL-OP expr
 expr HAT expr
 expr DOTTHAT expr
 ADD-OP expr
 expr QUOTE
 primary

primary ::=
 reference
 QUOTE id
 matrix
 REAL
 INTEGER
 STRING
 (expr)
 function-designator

matrix ::=
 [rows]

Chapter A. Omola Grammar

```
rows ::=
    columns
    rows ; columns

columns ::=
    expr
    columns , expr

range ::=
    expr
    expr .. expr

function-designator ::=
    id ( expr-list )
    id ( )

name-list ::=
    id
    name-list , id

indexed ::=
    id
    id [ range ]
    id [ range , range ]

indexed-list ::=
    indexed
    indexed-list . indexed

reference ::=
    indexed-list
    :: indexed-list
    id :: indexed-list
```

B

The Base Library

The following is a listing of the *Omola Base Library* which contains a set of definitions always available in Omola. The classes defined in this library must be used as super classes, directly or indirectly, of all user-defined models and model components.

```
LIBRARY Base;

Time TYPE Real;

Layout ISA Class WITH
  x_pos TYPE Real;
  y_pos TYPE Real;
  x_size TYPE Real := 400;
  y_size TYPE Real := 300;
  invisible TYPE Integer := 0;
END;

Model ISA Class WITH
  attributes:
    Graphic ISA Layout;
END;

Event ISA Class WITH
  condition TYPE Boolean := false;
END;

EventTerminal ISAN Event WITH
  Graphic ISA Layout;
END;
```

Chapter B. The Base Library

```
EventInput ISAN EventTerminal WITH
  causality TYPE STATIC Symbol := 'input;
END;
```

```
EventOutput ISAN EventTerminal WITH
  causality TYPE STATIC Symbol := 'output;
END;
```

```
Connection ISA Class;
```

```
Parameter ISA Class WITH
attributes:
  value TYPE Real;
  default TYPE Real := 0.0;
END;
```

```
Variable ISA Class WITH
attributes:
  value TYPE Real;
  initial TYPE Real := 0.0;
END;
```

```
DiscreteVariable ISA Variable WITH
attributes:
  value TYPE DISCRETE Real;
  initial TYPE Real := 0.0;
END;
```

```
Terminal ISA Class WITH
  Graphic ISA Layout;
END;
```

```
BasicTerminal ISA Terminal WITH
attributes:
  value TYPE Real;
  quantity TYPE STATIC String := "number";
  unit TYPE STATIC String := "1";
  variability TYPE STATIC (TimeVarying, Parameter) :=
    'TimeVarying;
  default TYPE STATIC Real;
END;
```

```

SimpleTerminal ISA BasicTerminal WITH
    causality TYPE STATIC (Undefined, Input, Output) := 'Undefined;
END;

SimpleInput ISA SimpleTerminal WITH
    causality := 'Input;
END;

SimpleOutput ISA SimpleTerminal WITH
    causality := 'Output;
END;

ZeroSumTerminal ISA BasicTerminal WITH
    direction TYPE STATIC (In, Out) := 'In;
END;

DiscreteTerminal ISA SimpleTerminal WITH
    value TYPE DISCRETE Real;
END;

RecordTerminal ISA Terminal WITH
    components:
END;

InputObject ISA Class WITH
    value TYPE Real;
    default TYPE Real := 0;
END;

ContinuousInput ISAN InputObject WITH
END;

DiscreteInput ISAN InputObject WITH
    value TYPE DISCRETE Real;
END;

```