LUND UNIVERSITY

Omola

An Object-Oriented Language for Model Representation

Andersson, Mats

1990

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Andersson, M. (1990). *Omola: An Object-Oriented Language for Model Representation*. [Licentiate Thesis, 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

Omola

—

An Object-Oriented Language for Model Representation

Mats Andersson

Lund 1990

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Licentiate Thesis

Date of issue
May 1990

Document Number
CODEN:LUTFD2/(TFRT-3208)/1-102/(1990)

Author(s)
Mats Andersson

Supervisor
Sven Erik Mattsson and Karl Johan Åström

Sponsoring organisation
The National Swedish Board of Technical Development (STU contracts 87-2503, 87-2425)

Title and subtitle
Omola — An Object-Oriented Language for Model Representation

Abstract

Models are essential in all kinds of control and process engineering. Computer based tools for control and process engineering are available but different tools do not communicate easily. This thesis presents a new language for structured dynamic models. The language, called Omola, is inteded to function as a common representation in an integrated environment of cooperating control engineering tools.

The thesis discusses the basic requirements on a new modeling languages and it presents the fundamental model structuring concepts and the design of Omola. Some basic algorithms for checking model consistency are outlined. Finally, as an example, an Omola model of a chemical reactor is presented.

Models can be decomposed into a multi-level hierarchy of submodels with abstract interfaces based on terminals and parameters. Models may have multiple descriptions of behaviour and terminals may be structured to model physical interaction. The framework for describing model behaviour is based on differential and algebraic equations but also more special descritions of behaviour are considered.

Omola is based on ideas from object-oriented programming. Models are represented as classes with attributes. Inheritance and hierarchical submodel decomposition improves model structure and facilitates reuse of models. The language is designed to be general and extendible in order to represent future, yet unpredicted, model representation concepts.

Key words
Computer Aided Control Engineering, object-oriented, modeling language, simulation language

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
102

Security classification

Recipient's notes

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, Telex: 33248 lubbis lund.

Table of Contents

Preface . . . 3
Acknowledgements . . . 3

1. Introduction . . . 4

2. Languages for Dynamic Models and Simulation . . . 8
2.1 Simulation languages . . . 9
2.2 Discrete event simulation . . . 11

3. Elements of Structured Models . . . 13
3.1 Abstraction beats complexity . . . 13
3.2 Basic model components . . . 17
3.3 The mathematical framework . . . 22
3.4 Other modeling concepts . . . 24

4. The Object-Oriented Paradigm . . . 25
4.1 The essence of object-oriented programming . . . 25
4.2 Object-oriented programming languages . . . 29
4.3 Object-oriented databases and environments . . . 31
4.4 Object-oriented modeling . . . 33

5. Omola . . . 34
5.1 Some Omola examples . . . 34
5.2 Basic Omola . . . 38
5.3 Model representation in Omola . . . 45
5.4 Discussion – representation of equations and connections . 50
5.5 Interpretation of Omola . . . 51

6. Model Operations . . . 53
6.1 Variable and parameter expressions . . . 54
6.2 Derivation of variable values . . . 57
6.3 Instantiation of Omola classes . . . 60
6.4 Simulation . . . 61

7. An Example . . . 63
7.1 The Process and its model . . . 63
7.2 A transfer function model representation . . . 68

8. Conclusions 71
8.1 Future work 72

9. References 76

A. Syntax Rules 81

B. Data Types and Model Classes 85
B.1 Omola Types 85
B.2 Omola Model Classes 86

C. Algorithms 88

D. Tank Reactor Example 92
D.1 The Reactor System and its Submodels 92
D.2 Standard Component Models 96
D.3 Terminals 98

Preface

This thesis describes the design of a new language. The purpose of this and every other language is two-fold: it is a framework for thoughts and it is a means of communication. In this case the language is designed to support our concepts of model representation in control engineering and to provide the means of communicating these ideas. To be more concrete, the language is designed as a new simulation language for continuous time systems with the ultimate goal that the language should be useful as a common representation in an environment of cooperating tools for system design. The design of a language with as wide ambitions as these is not a one man's work. In this case, the work has evolved during continuous discussions within the CACE (Computer Aided Control Engineering) group but it should still be viewed as a sketch which may be revised number of times before it reaches steady state.

This is an interdisciplinary work involving areas in control engineering and computer science, especially computer languages. The reader is assumed to have some basic knowledge in system and control engineering methodology and in programming methodology.

Acknowledgements

I would like to thank Professor Karl Johan Åström for initiating the project and for providing strong support and encouragement. I am also very grateful to my advisor Sven Erik Mattsson for his creative criticism and patience with my stubborness. A language is of little use if it has only a single user; Bernt Nilsson used Omola in his licentiate thesis, so in this case we are at least two users. I am very grateful to him for his valuable suggestions and for our creative discussions. Other members of the CACE group I would like to thank are Tomas Schönthal and Dag Brück for his help with the syntax specification and everything else. Finally I would like to thank Professor Björn Wittenmark for valuable comments on the manuscript and Leif Andersson for providing excellent computer support and TEX type-setting.

The financial support has been provided by the National Swedish Board of Technical Development (STU). The work has been carried out under contracts 87-2503 and 87-2425.

1. Introduction

This thesis proposes a new language for representing models of dynamic systems. The language is called Omola and it is based on ideas from object-oriented programming. The name is short for *Object-oriented MOdeling LAnguage.*

Omola is one of the outcomes from a larger project in computer aided control engineering – CACE. In this project it was realized that models are playing an essential role in engineering and in particular in the design of control systems. Most simulation languages and model representations used in various design tools are too specialized and unflexible to be used for general modeling. Omola has been designed to overcome some of these deficiencies.

In this introduction we will identify the reasons why a new modeling language is needed and then try to define the requirements of such a language.

An integrated CACE environment

One of the main goals of the CACE project is to design an integrated environment of cooperative tools supporting the various stages in process and control system design. We are not there yet, but Omola contributes as a common ground, or a core model representation, around which the tools may be arranged. The core model representation serves as a common database and a communication channel between the tools. In a sense, Omola is designed to be the Interlingua, or Esperanto, of modeling languages.

A CACE environment cannot be viewed as a single piece of software designed from a specification based on current knowledge about control design. Control systems design is a very inhomogeneous and fluctuating area. Different persons prefer different techniques and new methods are developed all the time. An environment for CACE must therefore be very flexible and adaptable to new tools and techniques. We may view a CACE environment as a collection of, more or less self-sufficient, tools.

Figure 1.1 shows a graphical view of a typical CACE environment. In the center there is the core model representation – a database* of pro-

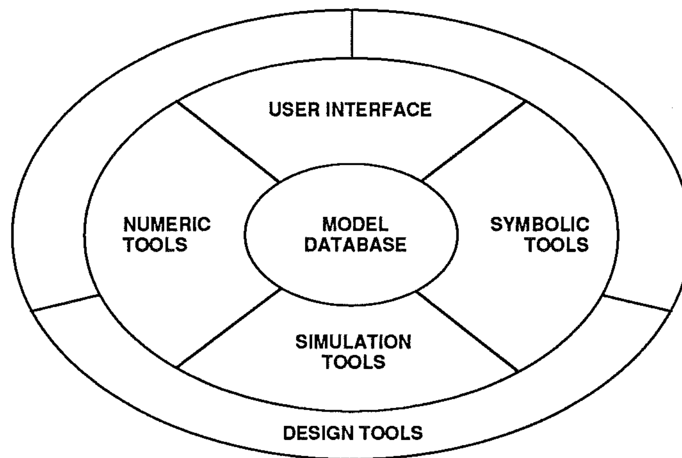

Figure 1.1 An integrated CACE environment

cess models, control systems, etc. The use of a database in a CACE environment is discussed in [Taylor et al., 1989] and [Tan and Maciejowski, 1989]. The database may include all kinds of data used in control systems design, such as control specifications, measured data, simulation results, etc. A number of tools are configured around the model database. One of the tools is an interactive user interface to manipulate and examine the models. Other tools are devoted to simulation, numeric computations and algebraic manipulation of mathematical expressions. In addition to a basic set of rather general tools there may be additional layers of more specialized tools devoted to particular areas of applications. The role that Omola plays in the CACE environment is that of being a textual representation of the core model database. In the earlier phases of the CACE project it was not recognized that a new language was needed. Model representation was discussed in terms of concepts and data structures rather than in language terms. Omola was actually invented as a tool for thinking about and discussing model structuring concepts but its potential as a general modeling language was soon discovered.

Requirements on a new modeling language

Many modeling languages in use today, have their origins in the seventies or even earlier. Most of these languages have poor structuring facilities and are not easily integrated within an interactive modeling environment. Also, most modeling languages are devoted to specific tasks, for example

---

* The word 'database' should here be interpreted as a general source of information and not necessarily as proper database manager

simulation, and the models are represented in a way that is special for the particular task. This makes it difficult to use the models for other purposes.

In a project studying new concepts for model representation [Mattsson, 1989a], a number of important properties of a new modeling language have been identified:

- The language should support a number of mathematical and logical frameworks for representing model behaviour. For example, differential algebraic equations [Elmqvist, 1978] [Mattsson, 1989b], transfer functions, state space descriptions, discrete events and qualitative behaviour.
- It should include concepts for structuring of large models. At least, it should support hierarchical submodel decomposition as in Dymola [Elmqvist, 1978].
- It should be modular in order to support reuse of parts of models in other models.
- It should be possible to include "redundant" information in models for the purpose of documentation and automatic consistency check.
- It should be generally useful as an input language for different control design tools and simulators. It should also be useful for model documentation and as a standardized exchange language between users and tools. That means, it must fit within an interactive CACE environment.
- It should be useful for incremental model development and graphical manipulations.

Object-oriented modeling

Object-oriented programming has been an increasingly popular programming methodology. Many ideas and concepts of object-oriented programming applies to representation of dynamic models as well [Åström and Kreutzer, 1986]. We started to investigate the use of object-oriented concepts at an early stage in the CACE project and when Omola was introduced it was clear that the language had to be object-oriented.

The concept of object-oriented modeling was introduced in [Nilsson, 1989] which used Omola to model a fairly large chemical plant. The aim of this study was to investigate how an object-oriented approach

can improve the modeling methodology of chemical processes in order to facilitate reuse of models.

Object-oriented modeling is not a change of paradigm, but rather, it enhances the notion of structured modeling that has been around for a couple of decades. In order to get more users, especially in industry, to adopt the new techniques of modeling better tools are needed. We need tools to investigate large system models and display them from different perspectives. We also need tools that help the user to detect possible inconsistencies and errors in models. Expert system techniques for design support and intelligent help systems are developing, but they need the firm ground of a formal model representation in order to be applicable. Further, the traditional tools used in engineering design has to be adapted to the new modeling concepts, i.e., they have to be integrated into a single, reasonably consistent, environment.

Outline of the thesis

The thesis is organized as follows. Chapter 2 contains a short overview of existing languages for modeling and simulation. The focus is mainly on continuous time systems but also discrete event modeling is discussed briefly. Chapter 3 introduces the basic concepts of model structuring. It contains a general discussion on the importance of abstraction and a presentation of the basic model components. A general introduction to object-oriented programming and terminology is given in Chapter 4. The main results are presented in Chapter 5, which gives the design of Omola, and in Chapter 6 which discusses some operations on Omola models. An example of a chemical reactor modeled in Omola is given in Chapter 7. Finally, a concluding summary and some discussions on future work are given in Chapter 8.

2. Languages for Dynamic Models and Simulation

It is very common that computers are used for simulating the behaviour of systems. The systems simulated can be of very different nature such as mechanical, chemical or electrical systems. Also non-technical systems like economical, ecological and sociological systems are often simulated using computers.

All simulations are based on some kind of model which represents some a priori knowledge we have about the system we want to study. Simulation can be viewed as experiments on models. Models are usually based on a mathematical or logical framework such as differential equations, difference equations or probabilities.

Early simulation programs were implemented directly in a general purpose programming language like Fortran. In these simulations, the models were represented implicitly in the program code. Usually, code representing the actual model was mixed together with code for solving the model equations and code for presenting the simulation results. Such simulation programs are hard to maintain, unless they are trivially small.

The first step towards structured modeling and simulation is to separate the model descriptions from the simulation machinery including solution algorithms and code for collecting and presenting the results. This separation can to some extent be made by using the structuring concepts, like functions and procedures, available in a general programming language. Simulation systems, based on this concept, are often written in Fortran and distributed as packages of subroutines.

A model typically describes parallel activities. For example, a set of differential equations, describing a continuous time system, are valid at all times. In order to solve the equations by a computer, the equations have to be evaluated in sequence. If functions in an ordinary programming language are used for representing such models, it is the responsibility of the model designer to decide upon a proper order of evaluation. However, more advanced simulation systems allow the user to specify the equations in any order. Very often it is also possible to group sets of equations together in modules. Such a simulation system is some-

times called *equation oriented* and it uses a special purpose *simulation* or *model language* for defining models. Some equation oriented simulation systems use a pre-processor that sorts the equations and translates them into, for example, Fortran procedures which are then compiled and linked in a standard way. Other systems translate the models directly into executable code.

2.1 Simulation languages

We will here give a brief description of a few systems for continuous simulation. A fairly complete listing of simulation systems available on the market is given in [Simulation, 1988].

CSSL and derivatives

CSSL is an equation-oriented simulation language standardized by the Society for Computer Simulation [SCS, 1967],[Korn and Wait 1978]. There exists a number of simulation systems based on the CSSL definition. One of the most widely spread is ACSL (Advanced Continuous Simulation Language) [ACSL, 1986].

A model in CSSL may contain a number of state equations and auxiliary variable definitions. A state equation is an expression assigned to a state derivative, while a variable definition is an assignment to a non-derivated variable. A state derivative is indicated by a variable name followed by an apostrophe. As an example, the following code models a damped pendulum. It contains two state equations defining the first and the second derivative of the angular displacement and two auxiliary variables: damping and pi.

```
fi' = fidot
fidot' = -sin(2*pi*fi) - damping
damping = 0.3*fidot
pi = 3.14
```

When several intermediate variables are used it is necessary to avoid *algebraic loops*. That means it must be possible to sort the expressions in an order such that each variable can be computed explicitly without the need to solve implicit equations.

CSSL based systems are all based on Fortran and they translate the models into Fortran code which is then linked together with the

procedures for integration etc. This makes it relatively easy to include special purpose Fortran procedures in the models.

Simulators based on CSSL make a reasonably clear separation between model definitions and the simulation experiment setup. However, in many cases, changes in the experimental conditions, like initial conditions and step sizes, require recompilation of the model. For some systems the the experimental conditions, presentation of results, etc., can be controlled interactively.

CSSL includes rudimentary model structuring concepts by the use of language macros. A macro looks similar to a procedure and it has a list of dummy arguments. When a model definition is translated, the macros are expanded and the dummy variable names are replaced by the actual ones. In this way the model structure is not preserved in the actual simulation model which is a serious drawback.

Simnon

Simnon is an interactive, equation based, simulation system developed at Department of Automatic Control in Lund [Elmqvist, 1975][Elmqvist et al., 1990]. Simnon is not based on an ordinary programming language but translates the models directly into executable simulation code. This makes the turn-around time from changes in the model until its ready for simulation much shorter. Commands for controlling the simulation experiments can be entered interactively or they can be saved on files as macros (command procedures).

Models can be decomposed into a one-level structure with modules, called *systems*. Each system defines a number of input and output variables, state variables and state equations. A special superior system defines the connections between the input and output variables of all the other systems. Sampled (discrete) and continuous systems can be mixed.

Figure 2.1 shows a small example of continuous Simnon system with one input and one output.

Dymola, LICS and Hibliz

Dymola is a modeling and simulation language with more advanced concepts for model structuring [Elmqvist, 1978]. It allows models to be divided into submodels to any depth. Interaction between submodels is modeled by *terminal variables* which can be grouped into *cuts* and connected with other cuts of similar structure.

```
CONTINUOUS SYSTEM proc
"Integrator with input saturation
INPUT u
OUTPUT y
STATE x
DER dx
upr=IF u<-0.1 THEN -0.1 ELSE IF u>0.1 THEN 0.1 ELSE u
dx=upr
y=x
END
```

Figure 2.1 Example of a Simnon system.

Dymola accepts state equations given on implicit form, i.e, a state equation that is an equality relation between two expressions rather than an assignment to a state derivative. Also general algebraic equations are permitted.

LICS (Language for Implementation of Control Systems) [Elmqvist, 1985] has many concepts in common with Dymola. However, LICS is an interactive environment for defining control systems. LICS is based on a graphical representation of models and uses a computer with interactive graphics capabilities. The concept of information zooming is important in LICS. On the highest level, models are represented graphically as block diagrams in which the user may pan and zoom. When he/she zooms into a block it opens up and its internal structure becomes visible.

LICS was further developed into Hibliz (HIerarchical BLock diagrams with Information Zooming), a prototype simulator for dynamical systems allowing continuous and discrete models [Elmqvist and Mattsson, 1989].

2.2 Discrete event simulation

So far we have only discussed simulation systems for continuous time or sampled data models. There is another, equally important, category of simulation systems based on discrete event models. Discrete event models appear in areas like queuing theory, communication, operating systems, manufacturing systems and in sociological and ecological systems dealing with populations of individuals.

Discrete event simulations are naturally object-oriented and the first object-oriented programming language, namely Simula [Birtwistle et al., 1973], was developed for the purpose of discrete event simulation. A typical discrete event simulation models customers as objects. They are created, they spend some time in the system interacting with other objects and service resources, they collect statistics and they may finally be removed.

Simulation of discrete event systems involves sampling of the execution of the discrete event model. Often discrete event models contain non-deterministic elements represented by some statistic distribution function. Experiments on the model then involves sampling the distribution function of the overall system. Such simulations are often called *Monte Carlo simulations.* A comprehensive discussion on discrete event simulation techniques and languages is given in [Kreutzer, 1986].

Combined discrete event and continuous time systems

Despite the fact that many systems are most naturally represented by combined continuous time and discrete event models, the two ways of modeling have evolved in two separated communities. Discrete event models and simulation are not so common in control engineering and, therefore, in this report we are mostly considering continuous time and sampled data models. However, we do not commit ourselves to only represent continuous time models.

Some examples of simulation system allowing combined discrete event and continuous models are Simscript II.5 [Fayek et al. 1987], GASP-IV [Pritsker and Hurst, 1973]. G2 [Moore, 1987] combines rule-based, continuous and discrete models with a graphical presentation. The numerical problems of combined simulation are treated in [Cellier, 1979].

3. Elements of Structured Models

This chapter is a detailed description of our conception of model structuring. We will identify the basic concepts of model representation and we will try to give them a well defined meaning. This chapter serves also as a motivation for the design of the Omola language.

The first section is a general discussion about the importance of using abstraction and decomposition in model representation. Then the basic building blocks, called components, of structured models are presented. The next section talks about the mathematical framework needed to represent model behaviour. Finally, some modeling concepts are presented that are not indispensable but might be considered in future developments of the project.

3.1 Abstraction beats complexity

Abstraction is the standard way for humans to understand and to operate complex systems. For example, in order to use a television set, one does not have to know how it is constructed inside. This is because the TV designer has provided us with an abstract interface. In this case it consists of a number of control buttons and a users manual describing how they are used. The designer of the TV set, or somebody who is repairing it, must have a more detailed understanding of its internal structure. However, a TV set might be constructed from basic building blocks like integrated circuits. In order to use an integrated circuit it is not necessary to know how it is constructed internally because the designer of the integrated circuit has provided an abstract description, containing only the minimal amount of information needed to use it. From this examples we can understand that different people, designers and operators of complex systems, work on different levels of abstraction. In particular, if many people are involved in the design of a large system, abstraction is necessary since no single person can know the details of the whole system.

Abstraction in programming

The evolution of computer programming languages is a movement towards higher levels of abstraction. The early programming languages like Fortran and Lisp abstract away most of the details of the computer hardware and make it possible for the programmer to create new procedural or functional abstractions. A function or a procedure definition is an abstract interface to an algorithm. Structured programming was developed as a programming methodology based procedural decomposition of the program. More recent programming methodologies focus on data abstraction and new programming languages with better support for this have been developed. Pascal is an example of this generation of programming languages.

Object-oriented programming is the latest trend in programming methodology. Object-oriented programming is based on abstractions, called objects, containing data as well as procedures. Object-oriented programming will be discussed in more detail in the next chapter.

An interesting fact in the history of programming languages is that Lisp, which was one of the first high level languages, has survived and is still used for advanced programming, especially by the AI community. One reason for this is probably that Lisp provides good support for creating abstractions, not only for data and procedures but also language abstractions. This means that it is possible to define new, special purpose, high level, languages within Lisp.

Abstraction and modularity in modeling

A concept closely related to abstraction is *decomposition*. A system description may, at any level of abstraction, be divided into functional or structural blocks, sometimes called *components* or *modules*. A good decomposition is characterized by loose coupling between the components and strong cohesion within the components [Booch, 1983].

Abstraction and modularity are as important in modeling as in programming and other kinds of engineering. Especially when the model becomes large, representing for example a complete chemical process or a power plant. As an example, let us consider a typical chemical process like the one modeled by [Nilsson, 1989]. First, the process can be divided into three main functional parts: the preparation subprocess, the reaction subprocess and the separation subprocess. Each one of these functional blocks contains a number of physical components like pumps,

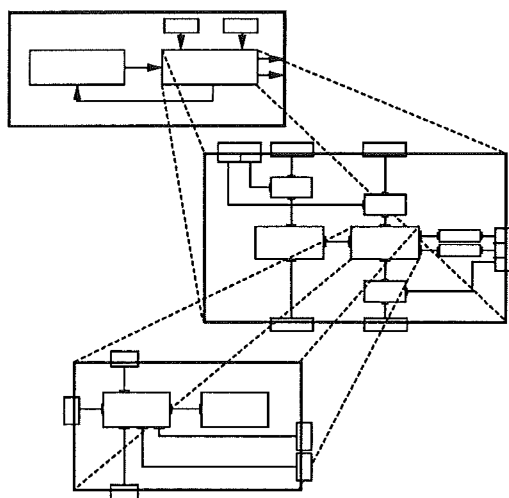

Figure 3.1 A multilevel hierarchy of models.

heat exchangers, valves, etc. Some of the components have complicated internal structures and can be further decomposed into subcomponents. It is also shown that some components can be decomposed into a machine model and a media model, thus separating the description of the physical equipment from the description of the chemical reactions. This kind of decomposition is based on the model designer's mental picture of what is going on in the system, rather than the physical appearance of the system. One can recognize two kinds of decomposition principles: functional decomposition and structural decomposition. These are two complementary ways to view a system and they may sometimes conflict and cause confusion. Which one is to prefer depends usually on the level of abstraction. Functional modeling is discussed in [Lind, 1987] and [Rasmussen and Lind, 1981]. Models can be decomposed in many different ways and there are no given rules of how it should be done. Some different methods of decomposition and some directions are given in [Nilsson, 1989]. The most intuitive method is decomposition by physical components. To make a good decomposition of a large model it requires quite good understanding of it. Very often the first attempt is not so successful but the model designer will get enough insight to make a new and better decomposition.

Modularization is a special case of decomposition. A module can be viewed as a component intended to be reused. The component models in the chemical process example can be treated as modules that can be reused in other similar models. A properly decomposed system model, with well defined abstraction interfaces, will not just help the model designer and the model user to understand the model, but it will also

support reuse of components. Model reuse is important since it makes it possible for a less knowledgeable user to develop new models in less time.

Model subtyping

A model *type* is a definition of a the characteristics of a particular type of system rather than one particular instance of a system. For example, when a certain kind of valve is defined as a model type, a composite model containing many instances of that type may be defined.

The type concept is general and useful. Everywhere in this report, when we are talking about models and model components, we are actually referring to model types and component types. A model *instance* is a representation of a particular system at a certain moment in time. An instance is needed when a model is going to be simulated, but it is less general and may be created automatically from its type definition. Model instantiation will be discussed further in Chapter 6. In the following chapters, when the object-oriented terminology has been introduced, the term *class* will sometimes be used for model type.

Once the type concept has been introduced it is natural to extend it by introducing the notion of *subtype*. Suppose we have defined a model type A and then want to define a new model type B which is identical to A except for a few additional features. Then it would be convenient to say that B is a subtype of A and then just give the additional attributes. Figure 3.2 shows a hierarchy of models with some typical attributes. For example, `TankWithHeating` is a subtype of `Tank` which is a subtype of `Vessel`. It inherits one attribute from `Tank` and two attributes from `Vessel` and it defines one local attribute.

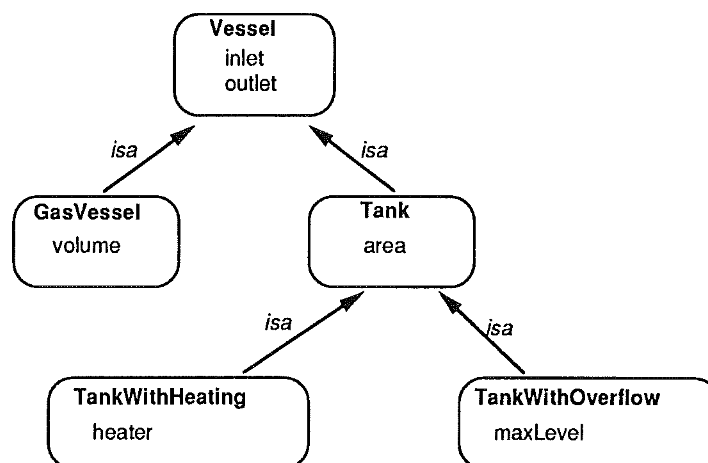

Figure 3.2 Example of a model type hierarchy.

Subtyping is a method of abstraction which is different from decomposition. There are cases in model design where it is not immediately obvious if subtyping or composition is appropriate but typically the distinction is clear. For example, it is natural to say that a truck is a special kind of motor vehicle but it is a bit odd to say that a truck is built from a motor vehicle or having a motor vehicle as a component. In this case subtyping should be used and the truck defined as a subtype of motor vehicle. On the other hand, the truck is composed of four wheels, an engine, etc. This is a typical example of decomposition where subtyping would be inappropriate.

Proper use of subtyping in models will not just save code, but also help users to understand the models more easily. Once a particular model type is well understood, it is often easy to understand the subtypes as well, in particular if they add few extra features. This will in its turn lead to models that are easier to maintain. Also, when models are stored in libraries, subtyping adds structure to the libraries and makes it easier for a model designer to find what he needs.

3.2 Basic model components

We will now define the concepts and different kinds of components needed to describe structure and behavior of dynamic systems in a modular way. Some concepts are well known and tested in other simulation and modeling languages while others are new results from the CACE project [Mattsson and Andersson, 1989].

Models

The *model* is the main structural entity and the most important type of module. A model has an interface and one, or many, descriptions of behaviour. The interface defines how the model interacts with the environment and the description of behaviour defines the internal behaviour of the model. The environment of a model consists of other models and the user. The model user should here be taken in a broad sense; it can be the model designer who wants to reuse the model or it can be a tool, for example a simulator.

The parts of a model describing its interaction with the environment are called *terminals* and *parameters*. Terminals may be connected to terminals of other models. A description of behaviour of a model is called

a *realization*. A model may also contain a number of internal variables representing the state of the model.

Terminals, parameters, realizations and internal variables are all different model *components*. They are themselves abstractions and they will be discussed in the following.

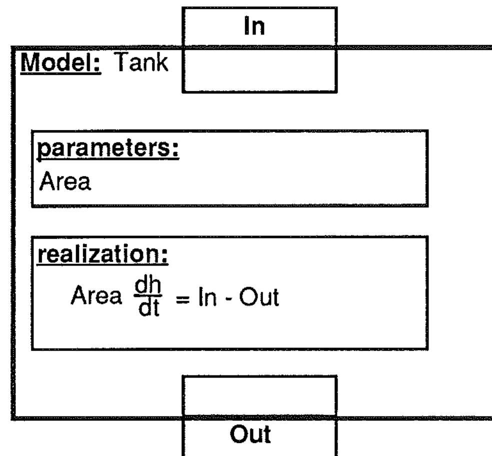

Figure 3.3 A model with interface and realization.

Terminals

A terminal is a model component and a part of the model's interface. In its simplest form a terminal is a variable which is shared between the model and its environment.

In models of physical systems, the terminals often represent physical quantities. For example, in a tank model the flow of liquid into and out from the tank are represented by two terminals. Terminals have values and a number of other attributes, for example, the name of the quantity, its measurement unit and the permitted range of the value. In most modeling languages, this kind of extra information can only be added informally as text comments to the model. If these terminal attributes were added to the terminal descriptions, the modeling tool could use this extra information for checking consistency when models are connected together.

A terminal is transferring information into or out from a model. The causality (input or output) should normally not be defined for terminals. In an equation based framework, causality can be derived automatically from the structure of the models and equations. In many cases the causality of a terminal is not a property of a particular model but depends on

how it is connected to other models. Models without defined causality of the terminals are more general [Mattsson, 1989b]. However, in some cases terminal causality is obvious, for example, the value from a measuring device is always an output terminal. This fact may be added to the terminal and it can be used for automatic consistency check.

Very often in physical models, interaction involves a group of quantities. For example, consider a tank connected with a pipe. A pipe connection may carry the quantities of mass flow rate, pressure and temperature, and possibly also information about the media flowing through the pipe. In this case, it is natural to group these quantities together in a single terminal. Such a terminal will be called a *record terminal* and it is a kind of structured terminal. Figure 3.4 shows a picture of two models connected through record terminals. A terminal that is not structured will be called a *simple terminal*. The components of a record terminal can be simple terminals or other structured terminals.

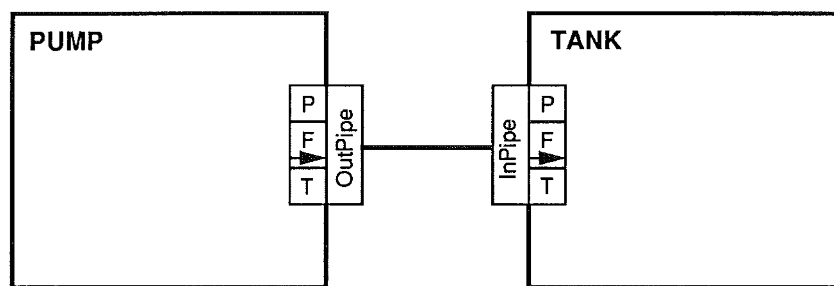

Figure 3.4 Example of structured terminals.

There is a second type of structured terminal called *vector terminal* that can be useful in some cases. A vector terminal is an array of identical components that can be indexed by an integer. A vector terminal may, for example, be used to represent a mixed flow of gases. The vector components may then hold the partial pressure of the gas components.

Realizations

A realization is a model component representing behaviour. There are different kinds of realizations based on different frameworks for behaviour descriptions. A realization based on some logical or mathematical framework is called a *primitive realization* since it does not depend on other models. A realization defining behaviour in terms of other models is called a *structured realization*. A model containing a structured realization is called a *structured model*.

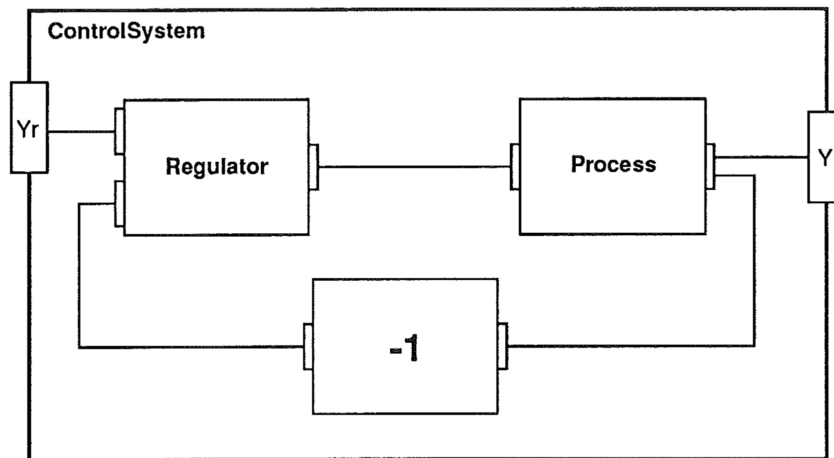

Figure 3.5 A structured model.

A structured realization has components called *submodels* and *connections*. A submodel is an ordinary model used as a component. A connection is a relation between two terminals. Structured realizations form the basis for hierarchical model decomposition. One reason for having the realization as an extra layer of abstraction between the model and its behaviour, is that it will allow a model to have more than one description of behaviour. Multiple realizations may represent different versions of the model, for example, a simple version and a more refined one. Of course, only one realization can be valid at a time, but it gives the user of the model a possibility to choose. Multiple realizations can also be used to represent different behaviours under different operating conditions. Suppose a primitive model described by a set of equations, most of which depend on some boolean condition, for example, if some temperature is below or above some critical point. In this case, the model might be more clear and easy to understand if the equations were separated into two different realizations. The correct realization may then be selected automatically depending on the boolean condition. An example of a model with two realizations, one linear and the other non-linear, is shown in Figure 3.6.

Connections

A connection is a relation between two terminals representing some kind of interaction between models. Connections are components of structured realizations. A connection may involve terminals of submodels and terminals of the model of which the realization is a component, in any combination.

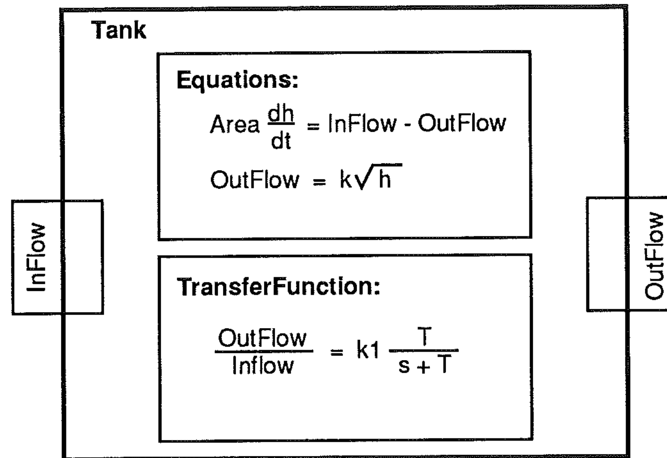

Figure 3.6 A model with multiple realizations.

In an equation based modeling environment, a connection should be interpreted as one or many equations between variables. We may call this interpretation the *semantics* of the connection. The semantics of a connection depends on the type of terminals involved. A connection between two structured terminals should be interpreted as connections between each one of the terminal components. A connection between two simple terminals representing across quantities should render an equality relation between the two quantities.

A tool for automatic consistency check of models may test that connected terminals are compatible in structure and in attribute values. For example, if terminals with defined quantity attributes are connected, the attribute values have to be the same if the model should be correct. Connection semantics is also discussed in [Mattsson, 1989c].

Parameters

Parameters are used by the model designer to make the model more general so that a user can adapt the model to his particular need. A parameter is part of the model's interface and it is a variable that is constant in time during simulation but may be changed by the model user.

When many submodels with parameters are connected together the resulting structured model will typically have too many parameters. Usually the parameters of the different submodels will not be independent and there may be parameters of submodels that the model designer does not want the user to change. For this reason, a concept for defining parameter dependencies is needed. Such dependencies will be called *pa-*

rameter constraints. Parameter constraints can be used by the designer of structured models to tie parameters of submodels to certain values and to bring parameters of submodels "up to the surface", i.e, to tie parameters of submodels to the value of parameters defined in the structured model. If all parameters of submodels are either fixed to constant values or bound to parameters of the supermodel, the user of the supermodel may ignore its internal definition. This is totally in accordance with the concept of abstraction.

Parameters may be used in at least two different ways. A simple kind of use is a parameter that represents a time invariant property of a model, such as the size of a tank, the maximum allowable control value, etc. Another kind of parameter affects the structure of the model itself. Such a parameter is called a *structure parameter.* A structure parameter may, for example, define the dimension of a state vector or an equation. The value of a structure parameter must be fixed before the model is instantiated for simulation.

Internal variables

Simple terminals and parameters are two kinds of variables that have been discussed so far. All other model variables are called *internal variables* or just variables. Internal variables represent the state of the model and they are not part of the model's interface, i.e., they can not be accessed from other models.

3.3 The mathematical framework

In order to define the behaviour of primitive models, some kind of mathematical framework i needed. In control engineering continuous systems are most often described by ordinary differential equations (ODE) or differential algebraic equations (DAE). Sampled data systems are described by difference equations.

In case of linear systems, polynomial transfer functions and normal linear state space forms are often used. These forms can be viewed as special cases of DAE or ODE systems that can easily be transformed into sets of differential equations. However, it is important that linear models may be represented on their special forms since they are required by some tools and they make certain analysis possible.

ODE and DAE models can be simulated and linear models can be analyzed in various aspects. A more general form of models are described by partial differential equations (PDE). It is not difficult to introduce a representation of PDE models in our system but it is of little use since there are very few general tools that can handle this kind of models.

Expressions and equations

An equation is a relation between two mathematical expressions. Expressions are operations on model variables and their time derivatives. The data types and operations needed to represent model behaviour are identical to those used in tools for numeric and symbolic algebra like Matlab and Macsyma. Except for ordinary mathematical operations that are available in most programming languages we need a derivative operator. The independent variable is usually the time which is a global variable available to all models.

All kinds of data like values of variables and expressions are of a certain type. Model variables have a declared type and expressions and equations must be type consistent in order to be legal. To represent differential equations the only data type needed is real scalars. However, in order to represent other kinds of model properties, integer, complex, and string values are useful. Matrix and polynomial representations are also convenient since they are often used in control engineering and they are fundamental in numerical software like Matlab.

Model equations appear to be similar to assignments in imperative languages like Matlab and Fortran. However, an equation describing the behaviour of a continuous model has a meaning that is very different from an assignment in an imperative language. A model equation represents a fact about the model that is true at all times. The assignments in an imperative language are evaluated in a well defined sequence and, interpreted as equations, they are only valid directly after evaluation. Model equations are evaluated in any order decided by the integration algorithm in the simulator.

Sampled and discrete event models (see below) are more naturally described by a sequence of assignments that are executed in sequence at every sample instance.

3.4 Other modeling concepts

In the previous sections in this chapter we have discussed the basic concepts for representing model structures. We have also identified the necessary mathematical framework needed to describe model behaviour as differential or difference equations. We will here give some ideas about other things that could be useful to include in model representations.

Events and actions

There is a type of dynamic models that is not so common in control engineering called discrete event dynamical systems. Discrete event models represent systems where all state changes occur at specific instances in time called events. The time instances are usually not known in advance. Many real systems show behaviour that is a combination of continuous time and discrete event dynamics. One example is an industrial plant that is doing batch processing. Another example is "intelligent", or rule-based control, and supervisory control [Årzén, 1987]. This type of systems and controllers are addressed as one of the future challenges to control — the question of representation as well as the development of a control methodology [Anon, 1987].

Petri nets [Peterson, 1981] is a formal way of representing discrete event systems. In order to represent combined discrete event and continuous models we have to introduce some new concepts describing when an event will occur and how it affects the continuous and discrete states of the system.

Model presentation

In an environment for interactive model design and simulation, models are treated as objects defining some structure and behaviour. In a computer using graphics and direct manipulation to access and manipulate models, the model objects are presented on the screen as graphical objects. A structured model may be presented as a block diagram or a flow sheet. Used as a submodel the same model may appear as named box or an iconic picture.

The way of presenting itself is an important property that may be included in the model descriptions.

4. The Object-Oriented Paradigm

Object-oriented programming is a program structuring methodology that has gained in popularity in the last few years. Object-oriented programming claims to increase the programmer's efficiency and facilitate reuse of code.

An object is an entity containing data as well as behaviour. An object-oriented program models the domain of discourse by a set of objects communicating by sending messages to each other. This is a powerful concept for abstraction and decomposition which in many cases is close to programmer's view of the problem domain.

This chapter is devoted to object-oriented programming from a general viewpoint. The important concepts and the characteristics of object-oriented programs are explained. Then follows a short overview of a few different object-oriented languages. Object-oriented database systems and specification languages are covered briefly. Finally, as an introduction to the next chapter, the notion of object-oriented modeling is introduced. A short introduction to object-oriented programming can also be found in [Stefik and Bobrow, 1986].

4.1 The essence of object-oriented programming

It is not possible to give a single, clear, definition of object-oriented programming. Instead, we will here describe object-oriented programming by its characteristic features. These features act as a common ground, and most of them are present in every object-oriented program. In [Mayer, 1988] these characteristic concepts are presented as the "seven steps towards object-oriented happiness". The steps are:

1. *Object-based* modular structure is used, which means that systems are modularized on the basis of their data structures rather than on the basis of their functional behaviour.

2. *Data abstraction* is used. The data of an object is not directly accessible for others. Rather, data is manipulated and accessed through

an abstract interface of access procedures. This means that the actual representation of the internal data is hidden for other objects.

3. *Memory management* is done automatically. Fully automatic memory management means that memory space is reclaimed automatically by the system when it is no longer needed, i.e., when there is no other object referring to the object at the particular location. A garbage collector, which may decrease the run-time preformance substantially, is needed for fully automatic memory management. A semi automatic approach to memory management is when the application explicitly calls a procedure deallocating objects that are no longer needed.
4. *Classes* are used to describe objects. A class is a definition of an abstract data type. It defines the representation of the data of all objects of that type, and it defines the access functions, i.e., the abstract interface of the objects.
5. *Inheritance* of attributes may take place between classes or objects. This means that one class may inherit functionality from another class, called the superclass.
6. *Polymorphism and dynamic binding* may be used. Polymorphism means that type declared variables may take values of the declared type or of any subtype. Dynamic binding means that the code that gets executed by a function call depends on the actual type of the argument values.
7. *Multiple inheritance* is possible. This means that a class may inherit from more than one superclass.

A programming languages that supports most of these seven concepts may be called an object-oriented language. Some of the most important object-oriented languages do not support multiple inheritance and there are examples of class-less languages that still are object-oriented. In many cases it is also possible to adopt an object-oriented style of programming in languages that are not considered to be object-oriented.

The real nature of object-oriented programming and the question of what is and what is not object-oriented is under much debate. This is mainly because object-oriented programming is a relatively new and a very "hot" area. Object-oriented programming is also very much a "style" rather than a well defined methodology.

Some of the important concepts of object-oriented programming mentioned in the seven steps above, are discussed in more detail in the

following.

Objects

The most basic concept in object-oriented programming is, of course, the *object*. An object contains data and it has some means to access or manipulate that data. Objects can be created, they can respond to stimuli from the environment and they can be terminated when they are no longer needed. An object has a unique identity but its state may change throughout its life-time.

A function that is associated with with a certain type of object and has access the internal state of the object is called a *method*. Methods are used to query the object about its current state, to change its state and to carry out various actions. Different objects may have the same methods but with different implementations. For example, consider two objects representing a circle and a rectangle. Both objects have a Draw method and a Move method. The Move methods may have the same implementation, changing the current position of the objects, while the Draw methods, causing the objects to be displayed on the screen, are different.

Calling a method of an object is sometimes viewed as a message is sent to the object. Object interaction is often called *message passing*. The message passing paradigm is sometimes emphasized by the syntax of the language. For example, suppose we have the object `Rectangle` and the integer variables `x` and `y` and want to evoke the Move method. In Flavors this would be coded as:

```
(send Rectangle 'Move x y)
```

while in Simula or C++ it looks more like an ordinary function call:

```
Rectangle.Move(x,y);
```

No matter whether the message passing metaphor is emphasized or not, it is the recipient object that decides which procedure to call in response. Just like functions, many methods return a result value to the sender. Every method is implemented as a function that has the exclusive right to access the object data. It is the responsibility of the function to leave the object in a consistent state. From a puritanically object-oriented view there should be no other way to access an object attribute but calling one of its methods.

Objects correspond closely to the way most people view the world. In order to comprehend the world around us, we like to view it as a

number of separate entities with limited interaction. Every entity has its current state which evolves in time due to interaction with other entities. It is relatively easy to model real world entities as objects. It is also not difficult to realize how more abstract entities of a real or an imaginary world can be represented as objects.

Classes

The structure of any object in a system is described by its *class*. A class is an object *type* definition. It can also be viewed as an implementation of an abstract data type. In comparison with a traditional programming language like Pascal, a class corresponds to a record declaration while objects are record variables. A class defines not only the data fields of the objects but also their methods. Objects that share the same class are said to be *instances* of that class. All instances of a particular class shares the same functionality and in particular they have the same set of methods with identical implementation.

Object data fields, corresponding to the fields of a record, are defined in the class. They are called *instance variables* since every instance has its own set. Some object-oriented languages also have *class variables* which are common, and accessible, to all instances of that class.

It is important to realize that classes are *descriptions* defining the properties of a set of similar objects. In a compiled object-oriented language like Simula or C++, classes appear only in the source code, defining objects which are created at run time. However, in dynamic object-oriented languages like Smalltalk or CLOS, also classes are objects. That means that new classes can be created and manipulated at run time. The class of a class object is called a *meta class*. The meta class defines the methods for all class objects. A typical class method is Make Instance (sometimes called New) which controls the creation of new instances.

Inheritance

A class may be defined to be a *subclass* of a previously defined class. The 'sub'-property is a *relation* to another class, not an absolute property of the class itself. The inverse relation will be called *super* class. A class will *inherit* all properties of its superclass. This applies to methods as well as data fields. It means that instances of the subclass will have all the properties that instances of the superclass have as well properties defined in the subclass itself. Inheritance works to any depth in a multi-level hierarchy of classes and subclasses.

A class is always an extension of its superclass. That means it has at least the attributes of its superclass and in particular, it has all the methods of its superclass. However, the subclass may change the implementation of a method. It is not possible to inherit only some of the attributes of a superclass or to remove inherited attributes. This is important since in means that everything that is true for some class is also true for all the subclasses. This makes it possible to do compile-time type checking of polymorphic variables. A polymorphic variable is a variable that is declared to be an object of a particular class but which at run time may take an object of any of its subclasses.

Inheritance makes it easy to define objects that are almost like objects of a previously defined class. That means we can reuse code and abstraction defined in other classes. Proper reuse of abstraction makes programs that are easy to read and to understand. It means that a concept that is a special case of another concept is represented by a subclass of the more general one. In reality, it is quite difficult to reuse code and abstraction, even in an object-oriented environment. Very often there is a conflict between reuse of abstraction and reuse of code. Object-oriented programs striving for maximum reuse of code are usually very hard to understand while programs striving for maximum reuse of abstraction may be relatively easy to understand.

4.2 Object-oriented programming languages

In this section we will give a short overview of some of the more important object-oriented programming languages. An overview of implementation languages for CACE software is also found in [Brück, 1987].

Simula

Many of the basic concepts of object-oriented programming originate from the Simula programming language [Birtwistle et al., 1973]. Originally Simula I was a special purpose language for discrete event simulation. It then developed into a general purpose programming language called Simula 67.

Simula is a block oriented language based on Algol. It has a class concept allowing data and procedures to be grouped together as a unit. A class can be *prefixed* with another class. A prefix class is the same as a superclass in our terminology. All declarations of a prefix class

are available in the subclass so the important concept of inheritance is supported.

Simula supports virtual procedures which means dynamic binding of procedures. It is the actual type of an object, determined at run time, that decides which procedure to call in respond to a message. Simula also uses a garbage collector to reclaim the memory of any object to which there is no reference.

Smalltalk

Smalltalk is an interactive, object-oriented, programming language [Goldberg and Robson, 1983]. It is also the origin of most of the object-oriented terminology used today. Smalltalk is totally object-oriented in the sense that everything is an object and all computing is done by message passing. Smalltalk-80 is also a complete programming environment including class browser, for efficient reuse of code, special editors, etc.

Since Smalltalk is an interactive language, there is no distinction between the compile time and the run time environment. New classes can be defined at run time as a result of a message to a meta class. This makes Smalltalk well suited for modeling purposes where classes are used to represent models and class instances are simulations of the models.

Lisp extensions

Lisp itself is not an object-oriented language but it is easily extended in various directions. This has resulted in a number of Lisp add-ons to support object-oriented programming. One of the more used ones is Flavors [Allen et al., 1984]. Flavors support multiple inheritance and it is dynamic in the sense that new classes (flavors) can be created at run time.

CLOS (Common Lisp Object System) [Keene, 1989] is a relatively new extension to Common Lisp adopting many of ideas form earlier object-oriented add-ons and from Smalltalk. ANSI has accepted CLOS as a standardized extension to Common Lisp.

C++

C++ [Stroustrup, 1986] is an extension to C that supports object-oriented programming with multiple inheritance. It is also intended to be "a better C".

C++ does not include a garbage collector. Temporary objects are destructed automatically while others have to be terminated by an explicit call to a destructor function when they are no longer needed.

The use of C++ is rapidly increasing, especially within industry, probably because of its origin in C and because it is supported by a few influential organizations. Recently a committee within ANSI has been formed to work on the standardization of C++ [Brück, 1990].

Frames

Frames [Minsky, 1975] is not really a programming language but a scheme for knowledge representation developed in the AI community. A frame is an entity representing an object, a class, a relation or any other concept. It has a number of attributes called *slots* and a slot can in its turn have attributes called *facets*. Values as well as procedures can be attached to slots or facets. Frames can be related by 'IS-A' and 'A-KIND-OF' links indicating subclass or instance relationships.

Frame systems have been developed as extensions to Lisp. Many knowledge representation and knowledge engineering systems use frame based representations. An example is KEE*(Knowledge Engineering Environment) that has been used extensively for prototyping in this project.

4.3 Object-oriented databases and environments

Because of the strong impact that object-oriented ideas have had on programming methodology, many ideas have migrated over to other areas of computer science and to engineering in general. In particular, this applies to the powerful data type concepts of classes with inheritance. Object-oriented database management systems (OODBMS) is a new research are that has been attracting much interest lately [Beech, 1987].

The question of how to represent data in a computer is called *data modeling* and it is a research area that is closely related to OODBMS. Also data modeling has been greatly influenced by object-oriented programming and by research on knowledge representation in AI like Frames.

Integrated environments for software engineering have been an active research and technology development area for some year now. Also

---

* KEE is a trademark of IntelliCorp

in this area interests have been attracted towards object-oriented thinking. Much inspiration for the CACE project has come from the development of integrated environments for software engineering. Further on, research on OODBMS and data modeling apply to model representation in Omola. Therefore, this section will discuss some of the main points in these research areas.

Object-oriented databases

The *relational data model* is the current state-of-the-art in database systems. This scheme of data modeling is based on *entities* and *relations* [Elmasri and Shamkant, 1989]. Much research in database systems has been aimed at representing a large number of fairly simple objects in an efficient way. The relational model is poor on representing complex objects with complex relations in between. There is much hope that an object-oriented data modeling scheme will overcome this deficiency.

Object-oriented database systems have been approached from two different directions (at least). One approach has been to extend a conventional database system with possibilities to declare new data types and classes with inheritance. Another approach has been to extend an object-oriented programming language with database facilities. These extensions are sometimes referred to as persistent objects, i.e., objects that survives from one execution to another. In order to have full database capabilities with persistent objects, it must also be possible to share the objects in a multi-user environment.

The problem of data modeling, i.e., how to represent data in the computer, is inherent in many application like engineering support systems (CAD, CACSD, etc). Such applications usually involve a modest amount of data with very complex relations. Traditional data models have been used but they are usually not the optimal solutions. Therefore, a number of more elaborate data models, better adapted for these kind of applications, have been developed; one example is the data modeling language Express [Schenck, 1988]. The data models support a number of object relations that are used frequently. For example, the relation between an object representing an assembly of parts and the objects representing the parts themselves is common in various engineering and product databases.

In software development support systems we have objects representing specifications, program definition and implementation modules in source code or compiled code. In such applications a number of different

relations between objects are needed and also some kind of version management system keeping track of the different versions of the software.

4.4 Object-oriented modeling

We have discussed how ideas of object-oriented programming have influenced other areas of technology and in particular the representation of data in database systems and in systems for engineering support. An approach to object-oriented modeling, rather similar to the one taken in this thesis, is found in [Piela, 1989]. Our approach will be the subject of the following chapter.

5. Omola

Omola is an object-oriented language designed to represent the modeling concepts presented in Chapter 3. One of the main design principles has been to make Omola as clean and simple as possible. That means, the language should contain as few concepts as possible and that these concepts should have a well defined and easily understood meaning.

Omola represents models as class objects with attributes. There are two kinds of relations between class objects: `IS-A` and `PART-OF` relations. An `IS-A` relation indicates that a class is a specialization of another one. A `PART-OF` relation means that a class can be a component of another class, i.e., classes may be nested.

The purpose of this chapter is to describe the basic concepts of Omola in detail and to show how they may be used to represent dynamic models. We will start to introduce Omola by some small examples of how it is used for model representation. In the following section we will present Omola in more detail and on a more general level. Finally we will return to discussing Omola as a modeling language, this time in a more systematic way.

5.1 Some Omola examples

Omola is designed to represent structured dynamic models in the fashion discussed in Chapter 3. In this section, we will introduce Omola by showing some examples of models and model components. The examples will not be explained in all details; only the most important features are discussed.

Models and model components are all represented as Omola classes. Every Omola class has a name, a superclass, and a set of attributes. When Omola is used in an environment for model representation, there are a number of predefined classes in the system, among these are: Model, Terminal, Realization, and Parameter. They will be used as superclasses in some of the following examples.

Primitive models

Suppose we want to define a tank model with some terminals and parameters. In Omola the tank might be defined as:

```
Basic_Tank ISA Model WITH
  terminals:
    inflow ISA Terminal;
    outflow ISA Terminal;
  parameter:
    tank_area   TYPE Real := 5.0;
END;
```

The defined model is named `Basic_Tank` and it is a subclass of the predefined class `Model`. The key-word `WITH` indicates the start of the class body which defines the attributes of the class. The tank has three attributes called `inflow`, `outflow` and `tank_area`. The words `terminals:` and `parameters:` indicate which role the subsequent attributes play in the model. `Inflow` and `outflow` are two terminals while `tank_area` is a parameters of the model. The terminal attributes are classes themselves while the parameter, in this case, is a simple variable with a type and a value.

In the basic tank model only the interface, but no behaviour, is defined. We may use inheritance and specialize the tank model by adding some description of behaviour. We do this by defining a new model:

```
Tank ISA Basic_Tank WITH
  realization:
    mass_balance ISA SetOfDAE WITH
      variable:
        level TYPE Real := 0;
      equation:
        tank_area * dot(level) = inflow - outflow;
    END;
END;
```

Since `Tank` is a subclass of the previously defined `Basic_Tank`, it will inherit its terminal and parameter attributes. We may say that the `Tank` specializes the `Basic_Tank`. The new model defines one local attribute named `mass_balance`, which is a subclass of `SetOfDAE`. `SetOfDAE` is a predefined class used as a superclass of primitive (equation based) realizations. In this case the realization defines a state variable, `level`, and a mass balance equation. The equation refers to the parameter, the

terminals and the variable of the tank. The dot operator, used in the equation as dot(level), denotes the time derivative of a variable.

A structured model

The tank defined in the previous example was called a primitive model because it did not contain any submodels. A model that is not primitive is called a *structured* model. Whether a model is structured or primitive depends on its realization. Here is an example of a structured model containing two buffer tanks connected in series:

```
Tank_System ISA Model WITH
  terminals:
    in ISA Terminal;
    out ISA Terminal;
  realization:
    tank_structure ISA Structure WITH
      submodels:
        tank1 ISA Tank;
        tank2 ISA Tank;
      connection:
        in AT tank1.inflow;
        tank1.outflow AT tank2.inflow;
        tank2.outflow AT out;
    END;
END;
```

The structured model called Tank_System has two terminals and a realization. The realization is a subclass of Structure which is a predefined in the system. A structured realization typically has components which are submodels and connections. The submodels are subclasses of the previously defined Tank. The connections are written as pairs of terminals with the key-word "AT" in between. A dot-notation is used to reference terminals of submodels.

Structured models are more naturally represented graphically as block diagrams. Figure 5.1 shows a block diagram of Tank_System.

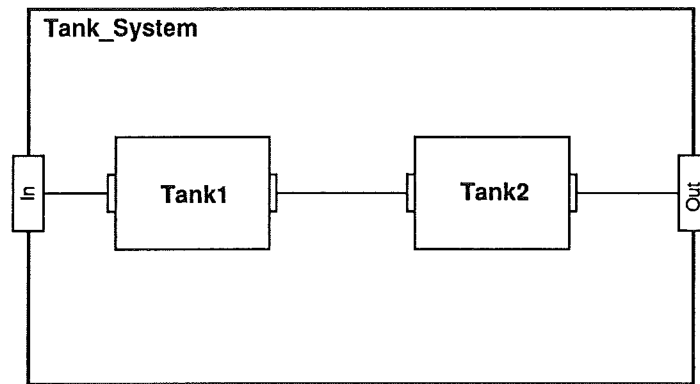

Figure 5.1 The tank system.

Terminals

Terminals are model components for defining interaction between submodels. A terminal class may be defined as a model attribute, as in the previous examples, but it may also be defined as a global class. Global terminal classes representing various physical quantities may be defined and saved in libraries. A terminal representing a single physical quantity may have a number of different attributes in addition to its value. Such a terminal is called a simple terminal and it is a predefined class in the system. It is instructive to see part of its Omola definition:

```
SimpleTerminal ISA Terminal WITH
  attributes:
    value         TYPE real;
    default_value TYPE real;
    quantity      TYPE symbol;
    unit          TYPE string;
    direction     TYPE (In, OUT, Across);
    ...
END;
```

The simple terminal has a number of attributes which may be given specific values in user defined subclasses. For example, we may define a voltage terminal in the following way:

```
VoltageTerminal ISA SimpleTerminal WITH
  quantity := Voltage;
  unit     := "V";
  direction := Across;
END;
```

The voltage terminal will inherit all the attributes defined for simple

terminals and three of those are given explicit values. We define a current terminal in a similar way and then construct a structured terminal representing an electric terminal:

```
ElectricTerminal ISA StructureTerminal WITH
  components:
    i ISA CurrentTerminal;
    u ISA VoltageTerminal;
END;
```

The superclass `StructureTerminal` is a predefined class.

We have now seen a few examples of models written in Omola. In the following sections Omola will be described more formally and in greater detail.

5.2 Basic Omola

It has been a desire to design Omola in such a way that new types of models and new model structuring concepts may be incorporated and expressed in the language. For this reason, Omola has become a general object-oriented data modeling language. It is instructive to view Omola on two different levels: The *basic level* of Omola contains a small number of basic concepts based on nested classes with attributes and inheritance. On this level there is no reference whatsoever to models and model structures. This level of Omola will be discussed some detail in this section.

On top of the basic level of Omola is the *model representation level.* This level defines how models and model structures, as discussed in Chapter 3, are represented in terms of basic Omola data structures. The model representation level of Omola will be discussed in the next section.

The advantage of defining and viewing Omola on the two different levels is flexibility. The basic level has a well defined syntax and semantics in terms of data structures. It is less likely that anything on this level has to change, even though completely new modeling concepts are introduced in the future. The model representation level, on the other hand, is more like a set of conventions between different users and tools on how to interpret the Omola code as models. This set of conventions may be extended in order to include new types of models and to satisfy the needs of new tools.

Omola classes

The most important kind of entity in Omola is the *class*. Every class has a name, a superclass and a set of attributes. A class definition looks like:

```
<name list> ISA <name of superclass> WITH
  <class body>
END;
```

The `<name list>` may be a single name or a list of names separated by a comma. A new class will be defined for every name in the name list. For example, to define two classes with the same superclass we can write

```
A ISA Class;
B ISA Class;
```

or with a shorter notation having the same meaning

```
A,B ISA Class;
```

The class body contains definitions of class attributes. If the class does not contain any local attributes the body may be omitted and the class definition would just be:

```
<name list> ISA <name of superclass>;
```

The key word `ISAN` may be used as a synonym for `ISA`.

A class body may contain variable definitions, variable assignments, component definitions and category tags. These kinds of class body items will be explained in the following.

A variable is a class attribute which has a name, a type and possibly a *binding*. A variable declaration has the following format:

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

The name list may be a single name or a list of names separated by comma for defining a number of terminals of identical type. Type designators will be discussed later but it is basically a name of a predefined data type. The ending assignment part is called the *binding* and it is optional in a variable declaration. A binding is an expression that sooner or later should evaluate to the correct type. The expression may be a literal of the correct type, the name of another variable attribute or a more complicated expression. A variable declaration with an assignment part gives the variable a defined value (which may or may not be known at this level); we may say that the variable (or parameter) is *bound* to an expression or a constant. This will be discussed in more detail in a section about parameter propagation.

A variable assignment has the following format:

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

And it is a short form for giving a new binding to an inherited variable. It redefines the inherited variable with the same type but with a new binding. As an example regard the following class definitions:

```
A ISA Class WITH
  v1 TYPE Real := 1.0;
  v2 TYPE Real;
END;

B ISAN A WITH
  v1 := 2.0;
  v2 := 2*v1;
END;
```

Class `A` defines two variable attributes: `v1` with a defined value of 1.0, and `v2` without a known value. Class `B` inherits the variables form `A` but it changes the defined value of `v1` to 2.0. Also in `B`, the variable `v2` is bound to the value two times the value of `v1`.

A component is a class attribute that is a nested class definition. That means a class locally defined in the body of another class. Components are used to define objects which have other objects as parts. Terminals and realizations are examples of model components.

A category tag is a word ending with a colon. A *category* is a group of attributes in a class. The category tag indicates the category of the attributes following it and it is valid until the next category tag or to the end of the class body. In the previous examples we have seen examples of categories like terminals, parameters and realizations. These are standard categories with a predefined meaning that will be discussed below. For all standard categories singular or plural form of the category tag can be used synonymously, for example, `terminal:` and `terminals:` denotes the same category. Attributes defined in a class body prior to any category tag are assumed to belong to the standard category `attributes:`.

Class inheritance

A class inherits all attributes of its superclass. This means that attributes (components as well as variables) that are part of the superclass definition are also a part of the definition of the subclass. A class definition may define additional attributes or it may redefine inherited attributes. Any attribute may be redefined in order to change its superclass, its type or its binding. A redefined attribute stays in its original category. It is not an error to redefine an attribute under another category tag but it will not change the inherited category and its bad style and may result in a warning from the Omola parser.

Here is an example of how inheritance may be used:

```
C1 ISA Class WITH
  % unimportant body
END;

C2 ISA Class WITH
  % unimportant body
END;

C3 ISA Class WITH
  cat_a:
    x,y ISA C1;
END;

C4 ISA C3 WITH
  y ISA C2;
END;
```

The class objects `C1` and `C2` are used as component superclasses in `C3` and `C4`. The class object `C3` defines two components, `x` and `y`, in the `cat_a:` category. Since `C4` has `C3` as a superclass, `x` and `y` will also be attributes of `C4` and belong to the same category. However, y is redefined with a different superclass: `C2`.

Data types

Omola defines a fixed set of data types. Among these are the ordinary algebraic data types like Real, Integer and Boolean that are present in most programming languages. A Cardinal is non-negative integer. There are also the symbolic data types String and Name. Further, Omola has

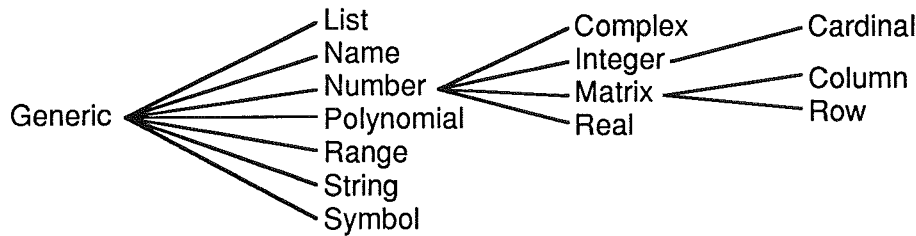

Figure 5.2 The Omola type hierarchy.

some aggregate types like Matrix, Polynomial and List. It is also possible to declare variables of generic type or as a number which can be a Real, an Integer or a Matrix. The Omola data types are ordered in a hierarchy with the generic data type on top and the most special data types at the bottom. A variable may take values of types that are below the declared type in the hierarchy. A complete list of Omola data types is found in Appendix B. Matrix and vector data types have parameterized type designators. They are written as:

`matrix[`*m,n*`]` `of` *element-type*
`column[`*m*`]` `of` *element-type*
`row[`*n*`]` `of` *element-type*

where the "`of` *element-type*" parts are optional. The parameters *m* and *n* indicate the number of rows and columns and they can be given as a constant number or an expression. The element type may be any simple type or Polynomial; if it is not specified it is assumed to be Real. A row vector defined by `row[n]` is identical to `matrix[1,n]` and a column vector defined by `column[m]` is identical to `matrix[m,1]`.

Expressions

Expressions are constructs that may appear to the right of the assignment operator "`:=`" in variable declarations or assignments, or as we saw above, in matrix type definitions. An expression may be a literal like 2.0, a variable name like `v1` in the example above, or a more complicated arithmetic expression. The value of an expression may not be known at class level but sooner or later, in some context, it should evaluate to the type of the variable on the left hand side of the assignment. In principle, it should be possible to derive and check the type of an expression on the class level. In reality, it might not be feasible to enforce type consistency at class level in a dynamic world of class definitions.

The expression syntax is similar to most programming languages

and to Matlab. Its formal definition is found in Appendix A and will not be discussed here. Expressions may contain calls to standard and library functions and operators. The most important variable operator is the time derivative: `dot(x)` refers to the time derivative of the variable `x` and `dot(x,n)` refers to the n'th derivative of `x`.

Basic scope rules

The rules determining which objects are visible, and may be referred to by name in a certain context, are called the *scope rules*. On this basic level of Omola the scope rules are simple and quite liberal. In the next section, describing model representation, we will make some further scope restrictions.

Classes in Omola are either *global*, i.e., defined on the top level, or they are *local* defined as components inside another class. A global class may be referred to by name in any context where it makes sense to refer to a class.

Class attributes may be referred to by *dot notation* which means that the class name followed by a dot and then the attribute name. In the last example we may refer to the `y` attribute of `C4` by '`C4.y`' which denotes a component class.

Every class body has its own local name space. The name space of a class is the name space of its superclass extended with all the local attributes. If we have a hierarchy of class bodies inside other class bodies, a name is resolved by first searching the own class body then the surrounding class body and so on, until finally the name-space of global classes is searched. This probably needs an example:

```
D ISA Class WITH
  x TYPE Real;
END;

E ISA D WITH
  y TYPE Real;
  z ISA Class WITH
    y TYPE Integer;
    w TYPE Real := x+y;
  END;
END;
```

Regard the expression '`x+y`' that binds the variable `w` in the component class `z` in class `E`. In this expression `x` is resolved to the variable defined

in class D since the name is not in the name-space of z but it is in E which has D as its superclass. The variable y in the same expression is resolved to the integer defined in the body of z.

The superclass of a component may either be

- a global class,
- a local component, or
- an inherited component.

In order to distinguish between the different cases without ambiguity, two special dot notations are used: `this` and `super`, where `this.c` refers to the component c defined locally in the same class body and `super.c` refers to an inherited component with the same name. No other kind of dot notation is permitted in the superclass position of a component definition. If none of these special notations are used, a reference to a global class is assumed. In the following example we can see how this works:

```
X ISA Class;

F ISA Class WITH
 X ISA Class;
END;

G ISAN F WITH
 X ISAN X WITH ... END;
 Y ISA this.X WITH ... END;
 Z ISA super.X WITH ... END;
END;
```

In the body of class G the first component X is a specialization of the global class named X. The second component, named Y, specializes the first component while the third component, named Z specializes the X defined in the superclass F.

Classes may not be subclasses of themselves and for this reason a component declared as 'X ISA this.X' is an error.

In order to make the task of an Omola parser easier, a class has to be defined before it is used as a superclass. This applies to global classes as well as to components within a class body. Any implementation of an Omola parser may relax this restriction.

5.3 Model representation in Omola

In the previous section, we have looked at Omola on the basic level and defined the primitive concepts. Now we will see how these concepts may be used to represent structured dynamic models. This part of the Omola definition is called the model representation level. It will be described in three aspects:

- A set of predefined classes,
- for each predefined class a set of categories with special meaning, and
- for each of these categories some rules limiting the kind of attributes defined in it and their interpretations as model components.

Some basic model classes

The basic classes defined and discussed in this section will serve as superclasses, directly or indirectly, of all user defined model component classes. Tools operating on models may rely on these superclasses as a classification of model components. In the following, when we say that something *is* a model or a terminal we mean an Omola class that is a descendant of the predefined classes Model or Terminal. The class–subclass hierarchy of all predefined classes is shown in figure 5.3. These classes will be described in more detail in the following.

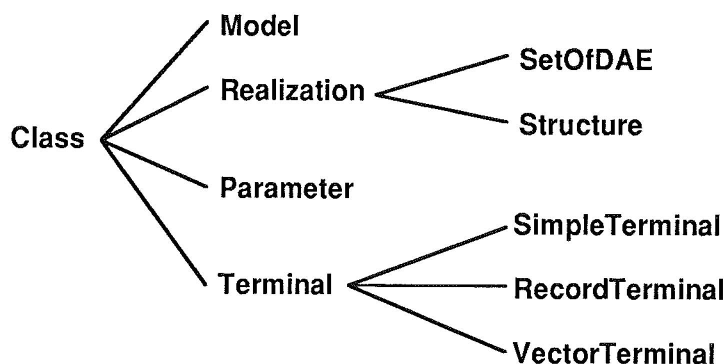

Figure 5.3 Predefined Omola Classes.

The Model class

The class Model is the Omola representation of the model concept discussed in Chapter 3. Its definition is simple since it has only one attribute:

```
Model ISA Class WITH
    primary_realization TYPE Name;
  terminals:
  parameters:
  variables:
  realizations:
  constraints:
END;
```

In the body we have also listed the categories of special importance; this is legal but of no significance since they contain no attributes. The different categories have special meaning to the user and to the various tools. There are also some rules on which kind of attributes the categories may contain:

- `terminals`: The attributes defined in this category define the interface of the model. Only components which are descendants of the superclass Terminal are accepted.
- `parameters`: These attributes are model variables which are not varying in time and may be set by a user. Variable attributes or components which are descendants of the superclass `Parameter` may be defined in this category. Since parameters and terminals are part of the model interface special scope rules applies to attributes defined in these categories. These scope rules will be discussed below.
- `variables`: These are model variables which may be time varying in a model instance during simulation. This category may contain variable attributes or components which are descendants of the class Variable.
- `realizations`: These are components defining the behaviour of the model. The components must be descendants of the predefined class Realization. The attribute `primary_realization` names one of the realizations to be used by default.
- `constraints`: These are equations used for propagating parameter values. Constraints will be discussed more in Chapter 6.

The Terminal class

A terminal is a model component used for defining interaction with other models. A terminal attribute is special because it may be related to a terminal in another model by a *connection*. The class `Terminal` is empty with the trivial Omola definition:

```
Terminal ISA Class;
```

It will serve as a common superclass to three other predefined terminal classes: `SimpleTerminal`, `RecordTerminal` and `VectorTerminal`. Their Omola definitions are given below. For a more elaborate discussion on terminal semantics, see [Mattson, 1988 and 1989c].

SimpleTerminals represent single quantities. A simple terminal has a number of attributes making it possible to specify its physical quantity, unit of measure, causality with respect to the model, value limits, etc. However, the most important attribute is `value`. The value of a terminal may be bound to other terminals through a connection or an equation. The class has the following Omola definition:

```
SimpleTerminal ISA Terminal WITH
  value TYPE Real;
  default TYPE Real;
  direction TYPE (Across, In, Out);
  causality TYPE (Read, Write);
  variability TYPE
    (TimeVarying, Parameter, Constant) := TimeVarying;
  low_limit TYPE Real;
  high_limit TYPE Real;
  unit TYPE String;
  quantity TYPE Symbol;
END;
```

RecordTerminals are similar to records in Pascal or structures in C. The fields in the record are defined as component attributes in the the category `components:`. The components of a record terminal are terminals themselves. The code is:

```
RecordTerminal ISA Terminal WITH
  components:
END;
```

VectorTerminals have a number of identical components. They can be used as column vectors in matrix equations. The `VectorTerminal` class defines two attributes: `length` which is positive integer determining the number of components, and `comptype` which is a terminal component.

```
VectorTerminal ISA Terminal WITH
  length TYPE Cardinal;
  comptype ISA Terminal;
END;
```

The Realization class

Realizations are used to define model behaviour. The class `Realization` is a trivial empty class definition:

```
Realization ISA Class;
```

There are two important subclasses called `SetOfDAE` and `Structure` which will be discussed here. When Omola is used in a multi tool environment other kinds of realizations are useful, such as transfer function or linear state-space realization.

SetOfDAE realizations define model behaviour as a set of differential or algebraic equations. A realization may define local parameter attributes and variable attributes with the same meaning as in models. Further more, a SetOfDAE realization may define a number of equations in the category `equations:`. Equations are written in a special syntax that is not actually part of the basic Omola:

expression = *expression*;

How this special syntax relates to the basic Omola definition will be discussed in a following section.

The equations of a primitive realization can refer to variables and parameters of the realization as well as variables, parameters, terminals and terminal components of the model of which the realization is a component. This will be discussed in more detail below.

Structure realizations define model behaviour as a set of submodels and connections. The following category tags are special:

- `submodels:` All attributes in this category must be components which are models.

- `connections`: A connection attribute is a relation between two terminals and it is written in a special syntax using the key word `AT`.
- `constraints`: are actually equations between parameter attributes. Equation syntax is used for defining constraints.

Scope rules of equations and connections

Some fundamental scope rules of class and attribute names were stated for Omola on the basic level. These rules were liberal in the sense that any attribute in any class was accessible by dot notation. When Omola is used for model representation is is desirable to restrict attribute access in order to make reuse of models secure. The restrictions are based on the idea that the terminals and the parameters constitute the model interface. Interaction between models should be limited to terminal connections and parameter constraint equations.

The following rules state all the permitted types of references. A connection defined in a realization may refer to

1a terminals of the model,

1b components of terminals of the model, or

1c terminals of submodels defined in the realization.

An equation or a binding expression in a model may refer to

2a variable attributes of the model,

2b terminals of the model, or

2c attributes of terminals of the model, or

2d variables of realizations.

An equation or a binding expression in a realization may refer to

3a variable attributes of the realization or the model,

3b terminals of the model,

3c attributes of terminals of the model, or

3d parameters of submodels defined in the realization.

5.4 Discussion – representation of equations and connections

In the last section we introduced some new special syntax for defining equations and connections in Omola. The formats were used because they are convenient and natural for the user. The `AT` key word to indicate a connection between two models is used in Dymola [Elmqvist, 1978]. The equal sign "=" is chosen to indicate equality between two expression because it is the standard mathematical notation (though in Matlab, for example, the equal sign is used for assignment which is different).

The concepts of equation and connection may be introduced in Omola in either of two ways:

1. The concepts are added as new primitives in the basic Omola definition, or
2. equations and connections are translated from the suggested form into basic Omola structures by the Omola parser or by a pre-processor.

The first alternative is simple and straight forward but it is against the desire to keep the basic Omola language definition as small and simple as possible. If we choose the second alternative we have to come up with a representation of equations and connections based on existing Omola primitives. This is not so difficult; if we introduce the following predefined classes in the model representation level, they can be used as component superclasses to represent equations and connections in models:

```
Equation ISA Class WITH
  left_hand_side  TYPE Number;
  right_hand_side TYPE Number;
END;

Connection ISA Class WITH
  terminal1 TYPE Name;
  terminal2 TYPE Name;
END;
```

Then for example, the equation '`dot(x) = -x`' and a connection between the `outflow` of `Tank1` to the `inflow` of `Tank2` may be represented by the following two components declared in proper categories of their models:

```
E1 ISAN Equation WITH
  left_hand_side := dot(x);
```

```
    right_hand_side := -x;
  END;

  C1 ISA Connection WITH
    terminal1 := Tank1.outflow;
    terminal2 := Tank2.inflow;
  END;
```

This kind of Omola representation of equations and connections may be hidden for the user since they are entered and displayed in short form. However, there is one advantage with the full Omola representation: connections and equations are named classes which can be specialized or redefined in subclass models. If only the short forms are used, a subclass model may only add more equations and connections or replace the whole set by redefining its realization. Individual connections and equations may not be redefined.

In the current Omola definition, only the short form of connections and equations is supported. This means as far as the user's concern alternative 1 is used but the actual implementation may use alternative 2.

5.5 Interpretation of Omola

Omola is a *declarative* language for model representation. This means that Omola code states the facts about a model rather than saying how things should be computed. Declarative models are universal and may be used for different purposes – not just simulation. The act of traversing a model or a set of model objects represented in Omola is called interpretation. Omola models may be interpreted differently by different tools depending on the purpose. The subdivision of attributes into categories may be used to separate out the aspects that are important for a specific tool. For example, a simulation tool that interprets a model in order to generate efficient simulation code is mainly interested in the equations and the connections of the model while it is ignoring many other attributes. The model representation level of Omola, as it is presented in this chapter, is mainly intended to support the needs of a simulation tool. However, it is also sufficient for control design and similar tasks, especially if it is augmented with realizations for linear models like transfer functions and state space descriptions. In contrast to this, if Omola

is used in a graphical tool for manipulating and viewing models on block diagram form, the connection and submodel attributes are not sufficient as they are presented. We need to add some additional attributes to the models dealing with their appearance as graphical objects. The point is that it is perfectly alright to add these attributes in a new category, say it is called `graphics:`, without disturbing the interpretation of the models in other tools. The simulator will simply ignore any attribute defined in the `graphics:` category.

Model instantiation

One of the more important ways to interpret an Omola model is called instantiation. An Omola class is a description of a real object rather than a representation of the object itself. This is an advantage when models are defined because objects that are separate but identical in the real world may share the same description. However, when a model is going to be simulated we need separate representation for all objects. On class level a tank model represents the concept of a tank, i.e., what is true for all tanks at all times. In a simulation on the other hand, a tank *instance* represents a specific tank at a specific moment in time. The procedure of interpreting Omola code in order to create instances is called *instantiation*. The instantiation procedure will be discussed in the next chapter.

6. Model Operations

Models represented in Omola can be used for various purposes like simulation, control design and documentation. Omola can be viewed as a textual image of a model database which is central in an environment of different special purpose applications or tools. An application interrogates the model database about structure and behaviour. Some applications may derive and insert new information into the database while others are just using it as input source code.

Most operations on models are performed by applications, part of an environment for simulation and design, which are not discussed in any detail in this thesis. However, some model operations, like model consistency check and instantiation, are fundamental to the database itself. These model operations will be the topic of this chapter.

Model correctness may be studied on several different levels. On the lowest level we have the Omola syntax. Omola has a grammer definition and every sentence that is parsed by the grammer is syntactically correct. The second level concerns semantics of models as it was discussed in Section 5.3. Correctness on this level is based on the scope rules of variable references and on terminal connections. If every variable reference and connection is resolvable according to the scope rules, the model is correct on this level. The third level of correctness is called model consistency. Variable values may be derived through bindings, equations and connections which may form a complex web of dependencies. If all derived variable values are consistent, the the model is consistent. Some values are static, which means, their values are the same for the whole class, that is for all instances at all times. Consistency of static variables may be checked at class level.

It is desirable to catch model inconsistencies as early as possible, preferably before instantiation. Model consistency check at class level requires that static variable values are derived from instantiated models. This will be the topic of the following sections. First there will be some introductory discussions and then an algorithm is presented. More details about the algorithm are given in Appendix C.

Instantiation is another fundamental operation on model classes. Instantiation is needed when a model is going to be simulated. Model

instantiation and simulation are discussed in the two last sections of this chapter.

6.1 Variable and parameter expressions

In this section we will discuss the meaning and the use of variables in models. It will serve as an introduction and as a motivation for the following section devoted to derivation of variable values.

Variables play three different roles in a model, they are representing

1. time varying properties (states) of the model,
2. time constant properties, i.e., model parameters, and
3. structure parameters.

Variables are declared in models as typed variable attributes or as components that are descendants of any of the predefined classes `SimpleTerminal`, `Variable`, or `Parameter`. By default, a variable is considered to represent a time varying property of the model, that means it belongs the the first type. A variable defined in the parameter category of a class is considered to represent a static property, that means it belongs to the second or third type. Also a simple terminal having its `variability` attribute bound to `parameter` or `constant` belongs to this type.

Parameters are considered to be part of the interface between the model and the model user and they provide the means of adapting the description of behaviour of the model. Whether a variable is a parameter or not, effects mostly its accessibility from other classes. This was discussed in the previous chapter in the section about scope rules. Parameters may also be treated specially by the user interface.

A *structure parameter* is a parameter that affects the model representation itself and its value must be known at class level. A structure parameter usually affects the dimension, i.e., the number of equations and variables, of the model. Examples are found in [Nilsson, 1989] where some models have an array of submodels of similar type. The length of such an array is a typical structure parameter which affects the instantiation of the model.

Variables may be related through bindings and equations. A binding or an equation may refer to variables of other models or model components according to the scope rules discussed in Section 5.3.

Variable bindings

A variable *binding* is an expression that can be used to compute the value of the variable. A binding that is constant, i.e., it is a number literal or an expression containing only variables with known values, may be evaluated at class level. The value of a variable with such a binding is the same for all instances at all times. In a model instance, the variable is considered to be a constant.

Here are a few examples of variables with bindings:

```
M ISA Model WITH
  v1 TYPE Real := 0.5;
  v2 TYPE Real := 2*v1;
  v3 TYPE Integer;
 parameters:
  p1 ISA Parameter;
  p2 TYPE Integer := v3;
END;
```

The model contains five variables, two of which are considered to be parameters. The values of `v1` and `v2` are known because `v1` has a binding to a constant and `v2` has a binding which can be evaluated to a known value. All the other values are unknown in this model at class level; `v3` and `p1` because they have no bindings and `p2` because it is bounded to a variable with an unknown value. (Recall that since `p1` is a subclass of parameter, its value is defined by the value of the inherited `value` attribute, which has no binding.)

Variables with unknown values may be given values in a subclass, by a constraint equation in a supermodel or in a model instance.

Equations and constraints

Equations are mathematical expressions defining relations between variables. Some equations are static, i.e, they contain only variables which are time invariant like parameters. Static equations are sometimes called constraints or parameter equations. A constraint containing only variables that are known may be evaluated and checked for consistency. A model with variable values that are not consistent with the constraints is said to be inconsistent. As an example of an inconsistent model consider a model where two simple terminals are connected. One terminal has the `quantity` attribute bound to `Voltage` while the other terminal has the same attribute bound to `Current`. The connection defines an

implicit equation between the quantity attributes. Since the values of the attributes are not the same we have an inconsistency in the model due to the connection.

From a model designer's point of view, parameter bindings and constraints are ways of defining parameter propagation. Parameter values may propagate up and down in a hierarchy of models and submodels, and they may propagate via terminals and connections. Regard the following example of a structured model using the model M defined above, as submodels.

```
MM ISA Model WITH
 parameter:
  p ISA Parameter;
 realization:
  MM_behaviour ISA Structure WITH
   submodels:
    M1 ISA M;
    M2 ISA M;
   constraints:
    M1.p = p;
    M2.p = 2*p;
  END;
END;
```

The two constraint equations in the realization relates the p parameters in the submodels with the p parameter of MM. This means that if any of the tree parameters gets a binding the other two will also be bound. In particular, if p of the supermodel MM is left unbound in the class it will be a free parameter in all instances. When the user assigns a value to the parameter in an instance, this value will propagate down to the submodels as well.

From a formal point of view, all bindings and constraints in a model, define a system of equations. On class level, this system of equations is typically under determined, i.e., there are more variables than equations. However, the system of equations can usually be partitioned such that some groups of variables can be solved from a group of equations while some other equations are used to test for consistency. In general, constraints and bindings are nonlinear expressions. Using them for deriving parameter values involves solving nonlinear systems of equations. It is the limitations of the nonlinear equation solver that restricts what type of constraint expressions that may be used for deriving variable values.

It is desirable to derive variable values and to check for model inconsistency at class level. For this reason, we will in the following section describe an algorithm for deriving variable values by examining the variable bindings, constraints and connections defined in the model.

Variable consistency

A variable may get values from a bindings as well as from equations or constraints. These values have to be consistent; if they are not, we say that the model is inconsistent. The following is an example of an inconsistent model.

```
BadModel ISA Model WITH
    X TYPE Real := 0.0;
    Y TYPE REAL := 1.0;
  constraint:
    X=Y;
END;
```

Inconsistencies may also result from connections indicating that a submodel is used in an incorrect manner. Models should be checked for consistency before they are instantiated for simulation or any other purpose. If models are created interactively, consistency can be checked whenever new attributes are added. The model designer will then get immediate feedback if he adds a connection or an equation that leads to an inconsistent model.

In order to check if a model is consistent, it must be possible to derive variable values at class level. This procedure is given below.

6.2 Derivation of variable values

As mentioned above, a variable may get its value through a binding or through a constraint. Before we present the algorithms for deriving variable values we have to introduce a new concept called *context*.

Context

A *context* is a chain of component attributes from a root class to a particular attribute. A context can also be viewed as an instantiation of a model hierarchy. Contexts are needed in order to reason about models at the class level as if they were model instances. Contexts are not part

of the Omola language; it is introduced here as a tool for reasoning about models and it will be used in the algorithms discussed later.

Regard the following example with a simple and a structured model with some constraint equations:

```
M1 ISA Model WITH
  p TYPE Real;
END;

M2 ISA Model WITH
 realization:
  M2struct ISA Structure WITH
   submodels:
    S1 ISA M1;
    S2 ISA M1;
   constraints:
    S1.p = 1;
    S2.p = 2;
  END;
END;
```

The variable attribute p defined in model M1 is a class objects, i.e., it represents the properties of the p attribute of *all* models of type M1 including all subclasses of it. If we want to reason about model M2 and its equations we need to distinguish between the p of S1 and the p of S2, however, at class level they refer to the same object. One solution would be to actually instantiate the model and then reason about the instances which will be unique. This would be unnecessary work since we do not need the instance objects themselves, we only need to reason about them. Therefore, the context concept is useful.

Contexts are similar to dot-notation and we can use dot-notation to represent a context in written form in this report. For example the context "X.Y.Z" means the Z attribute of the Y component of the class called X and it represents an instance of Z given any instance of X. A single element context, for example "X", should be interpreted as any given instance of X. In the example above we can imagine three different contexts for p; they are written as "M1.p", "M2.M2struct.S1.p" and "M2.M2struct.S2.p".

Operations on contexts are defined and used in Appendix C.

Bindings and constraints

A binding for variable v may formally be written as the equation

$$v = f(v_1, ..., v_n)$$

where f is a known function. From this equation we can compute the value of v if the values of v_1 to v_n are known.

Any constraint equation may, by simple manipulations, be transformed into the equation:

$$g(v_1, ..., v_n) = 0$$

From this we can compute the value of a variable $v \in \{v_1, ..., v_n\}$ if the equation is *solvable* for v and if all the other variables appearing in the equation are known. What is solvable depends on the function g and on the solution algorithm. A minimum level of ambition is to be able to solve for all variables that appear linearly in the equation.

Algorithms

The main algorithm for deriving a variable value in a certain context will be given as a function written in a pseudo language.

The main function is called `find_value` and it takes a context (of a variable) and tries to derive a value. If the variable has no binding and if there is no equation which can be solved for the variable then `nil` is returned.

```
find_value(v: Context): Value;
BEGIN
  IF binding(last(v)) THEN
    RETURN eval(v);
  ELSE
    FOR EACH e IN find_equations(v) DO
      IF x:=solve(e,v) THEN
        RETURN x;
    END
END
```

The expression `last(v)` refers to the variable itself and the function `binding` is a simple database lookup that returns true if the variable attribute has a binding. The function `eval` evaluates the binding of the

variable in its context. It will call `find_value` recursively to get values of variables appearing in the expression. The function `find_equations` returns the set of all equations in which the variable v appears while `solve` tries to solve an equation with respect to the variable in its environment.

The function `find_equations` is non-trivial and its implementation is given in Appendix C. The basic idea is to search the environment for equations and connections referring to the particular variable. The scope rules given in Section 5.3 give some ideas of how this search may be limited to a small set of models and realizations.

6.3 Instantiation of Omola classes

An Omola instance is an object created based on an Omola class. The instance has the same attributes as the class and their values are references to other instances. An instance of a variable attribute is an object capable of storing a value of the variable type. Any number of unique instances may be created from one class.

Variable resolve

Expressions in equations, connections, attributes and bindings contain symbolic references to other model attributes. During model instantiation these references have to be resolved, i.e., they have to be turned into references to attribute instances. Instantiation of expressions is a function that takes an unresolved expression and a class instance and returns an expression with all variables resolved. Since resolve requires that all instances of referenced variables are available resolve must be done *after* instantiation of each attribute.

The instantiation of a an Omola class is a simple procedure that is recursive in the class attributes. The procedure is outlined in the following:

```
Instantiate class:
  Create a class instance object.
  For each attribute of the class:
    Instantiate the attribute and add it as attribute
    of the instance object.
  For each expression in the class:
    Insert the resolved expression in the instance.
```

```
Instantiate variable:
  Create a variable instance.
```

Structure parameters

All structure parameters of the model have to have known values before the instantiation can be done. Recall that a structure parameter by definition affects the instantiation. A typical example is a parameter defining the dimension of a vector terminal.

Any change in the model class affecting the structure of the model will make all existing instances of that class invalid. It might be possible to develop an instance updating procedure that updates all instances according to the changes in the class but this may be rather complicated. The simplest solution to the updating problem is probably to disregard the old instances and create new ones. However, it might be possible to allow some minor changes, not affecting the basic structure, in a model class and to propagate these changes to all existing instances. For example, minor changes to the equations of a primitive realization are easy to propagate to all instances.

6.4 Simulation

A simulator is one of the most important tools in an environment for modeling and process design. In this section we will outline the basic procedure of setting up a simulation based on a model represented as an Omola class in the model database.

Setting up a simulation of a model represented as an Omola class starts by instantiation, as it was outlined above. Then the model instance structure is traversed and every resolved equation is collected. Equations must also be generated from every connection between simple terminals. A connection between two structured terminals will then result in a number of equations. When connections are turned into equations one have to consider if the terminals are across or through and if conversion factors between different units of measure has to be introduced.

From the equations collected from the model instance we can now, by simple manipulations, turn the problem into the differential/algebraic (DAE) form

$$g(t, \dot{x}, x, v, p, c) = 0$$

where t is the independent variable (usually time), x is a vector of variables that appear derivated, v is a vector of other unknown variables, p are known parameters and c are known constants. This type of problem can be solved by standard DAE solvers; DASSL [Petzold, 1982] is an example of such a routine. The DAE solver needs a routine for computing the residual:

$$\Delta = g(t, \dot{x}, x, v, p, c).$$

The residual can be computed by interpreting and evaluating the instantiated equations or, in order to increase the efficiency, the equations can be coded in some programming language like Fortran or C which is then compiled.

In most cases it is also possible to increase the efficiency by applying simple symbolic manipulations. Many intermediate variables and trivial equations like $A = B$ are generated from connection equations. These variables may be eliminated thus reducing the size of the problem. The equations and variables are also sorted so that the the problem is turned into a block triangular form. This may detect deficiencies like too many or too few equations. More about this is found in [Elmqvist, 1978] and [Mattsson, 1989b].

7. An Example

In this chapter we will discuss a process model represented in Omola. The process is a chemical reactor, chosen as an example of a reasonably complex system, so that the structuring facilities of Omola are demonstrated. First the process is presented and then the Omola model is discussed and some of the interesting points are highlighted. Finally, we will see how a totally new model representation — a transfer function — can be introduced in Omola.

7.1 The Process and its model

The process to be modeled is a continuous, well stirred, tank reactor for an exothermic reaction. The reactor can be used as a component in a larger chemical process with other components processing the raw materials and the product and with systems for recirculation of energy and materials.

The process is described in [Nilsson, 1989b] which also presents a hierarchical decomposition of the system. The same model decomposition is used in this example with only some minor changes.

A schematic picture of the process can be seen in Figure 7.1. The main components of the reactor is the tank, which is supposed to be homogeneous in concentration and temperature. The tank has an input flow of mixed raw materials and an output flow of reaction products. The input flow can be controlled by a valve while the output flow depends on the following stages in the process. The reaction dynamics is supposed to be fast compared with the residence time in the tank.

Since the reaction is exothermic, heat has to be dissipated from the tank which is done by a cooling jacket. The flow of cold water through the jacket can be controlled by a valve. During startup of the reactor, steam or hot water can also be fed through the cooling jacket in order to rise the temperature to the desired operating point.

The reactor system is also supplied with devices for measuring the temperature and the level in the tank and the output flow.

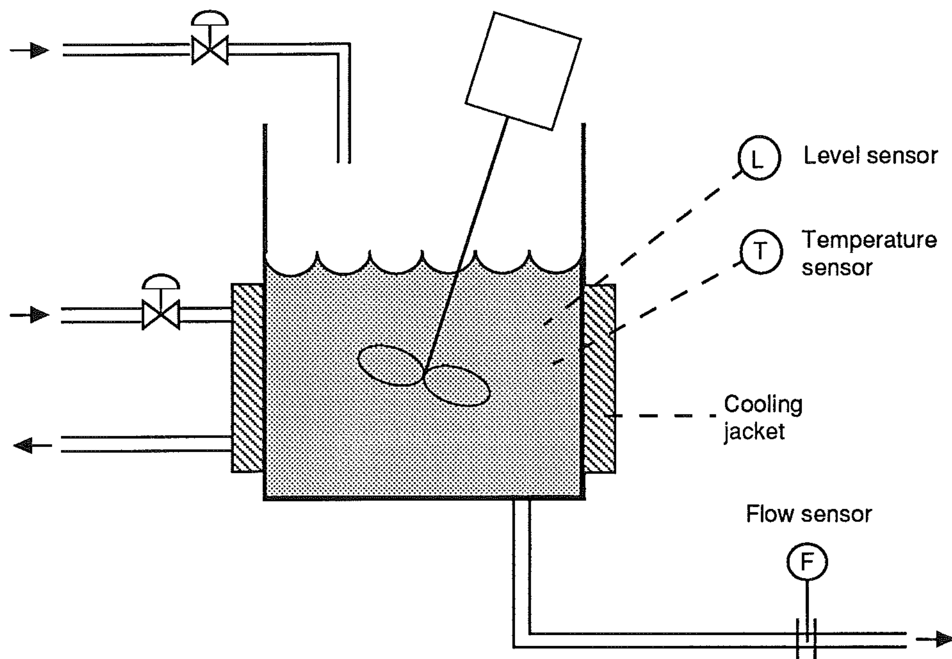

Figure 7.1 The chemical reactor system.

The model

The Omola code of the model is given in Appendix C. We will here give a brief top-down description of the model structure.

The model of the complete reactor is called `ReactorSystem` and it is decomposed into components representing the reactor vessel, the cooling jacket, and the valves and measurement devices; see Figure 7.2. The reactor system has terminals representing the control and measurement signals and the flows of cooling medium, raw materials and product. The flows are represented by structured terminals which will be discussed below. The interaction between the cooling jacket and the reactor vessel is modeled by a connection between two heat flux terminals representing the temperature and the energy flux through the walls of the vessel.

The valves, the cooling jacket and the measurement devices are primitive models. The cooling jacket model is a simple heat transfer through a wall with a given heat transfer coefficient. The cooling medium is supposed to have a linear temperature profile through the jacket.

The reactor vessel has an internal structure based on a media–machine decomposition [Nilsson , 1989]; see Figure 7.3. The machine model contains mass, energy, and component balance equations. The media model describes the chemical reactions and the properties of the involved substances. The interaction between the machine and the media model is represented by a connection between two media data terminals.

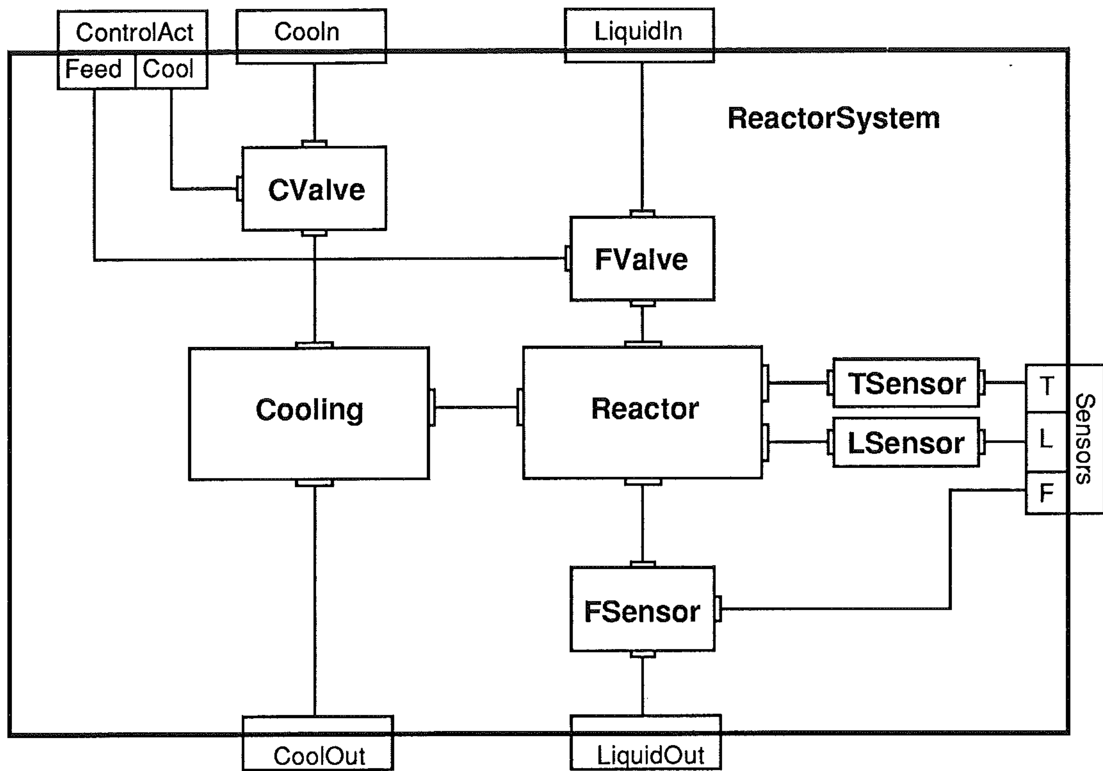

Figure 7.2 The structure of the reactor system model.

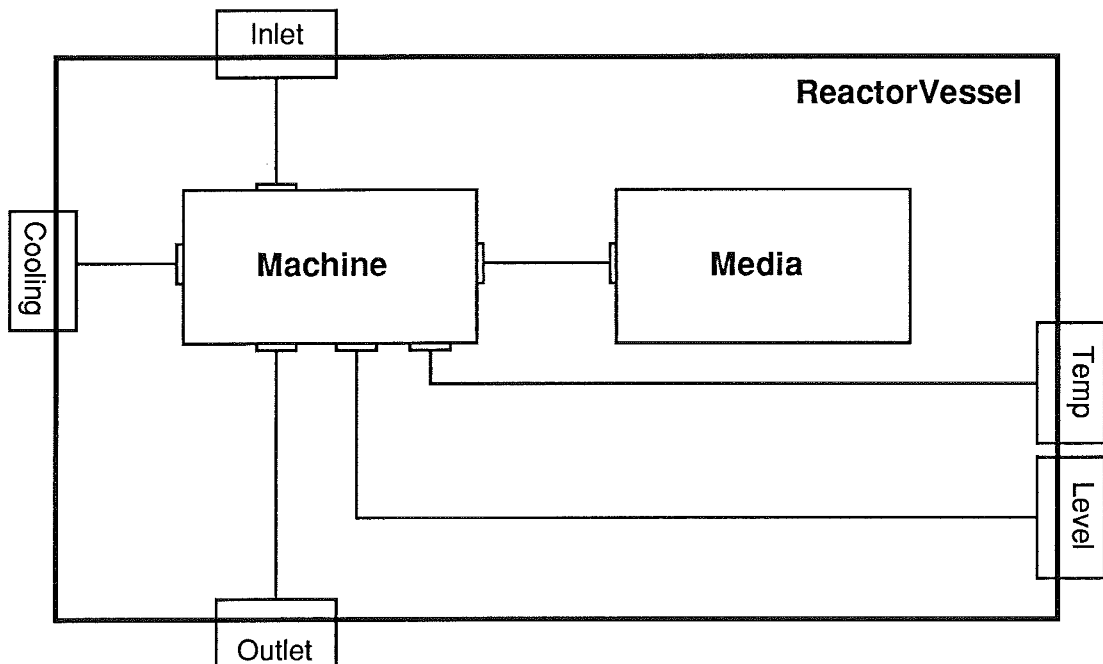

Figure 7.3 The structure of the reactor vessel model.

The terminal classes defined in the example are typical for this kind of application. The terminals could have been taken from a library of

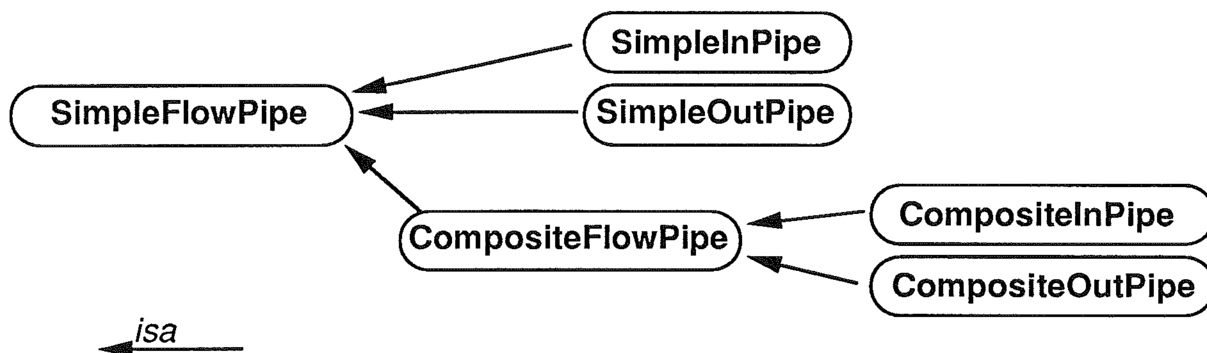

Figure 7.4 The inheritance structure for the flow pipe terminals

general terminal types used for chemical process modeling.

Inheritance is used for several terminal definitions. For example, `SimpleFlowPipe` is a record terminal with three components: flow, temperature and pressure and it is not intended to be used directly but as a general superclass for a number of specializations. The flow component (called `F`) is just included in `SimpleFlowPipe` as a dummy; it is specialized in the descendants `SimpleInPipe` and `SimpleOutPipe`, used for the flow of cooling liquid, to be a `FlowInTerminal` and a `FlowOutTerminal` respectively. `FlowInTerminal` and `FlowOutTerminal` are simple terminals where the quantity attribute is bound to `volumetric_flow`, the unit of measure is set to m^3/s, and the direction attribute is bound to `in` and `out` respectively, indicating that the components are through terminals.

Another specialization of `SimpleFlowPipe`, called `CompositeFlowPipe` and used in the product circuit, adds a fourth component which is a `CompositionTerminal` — a vector containing the concentrations of the chemical components of the flow. The dimension of the vector, i.e., the number of flow components, is not specified in the terminal but derived from the models where it is used. The `CompositeFlowPipe` is also specialized in the same way as `SimpleFlowPipe` giving the inheritance hierarchy shown in Figure 7.4.

Inheritance is also used in the valve models. The reactor system has two valves: one is controlling the flow of cooling liquid and one is controlling the input flow of the vessel. We want to use the same model for both valves but the two flows are represented by different types of terminals; the product flow has a composition component not present in the cooling flow.

We start by defining a simple valve model, called `Valve`, that has `SimpleInPipe` and `SimpleOutPipe` as flow terminals. These are the same types of terminals that are used for the cooling circuit so we can use `Valve`

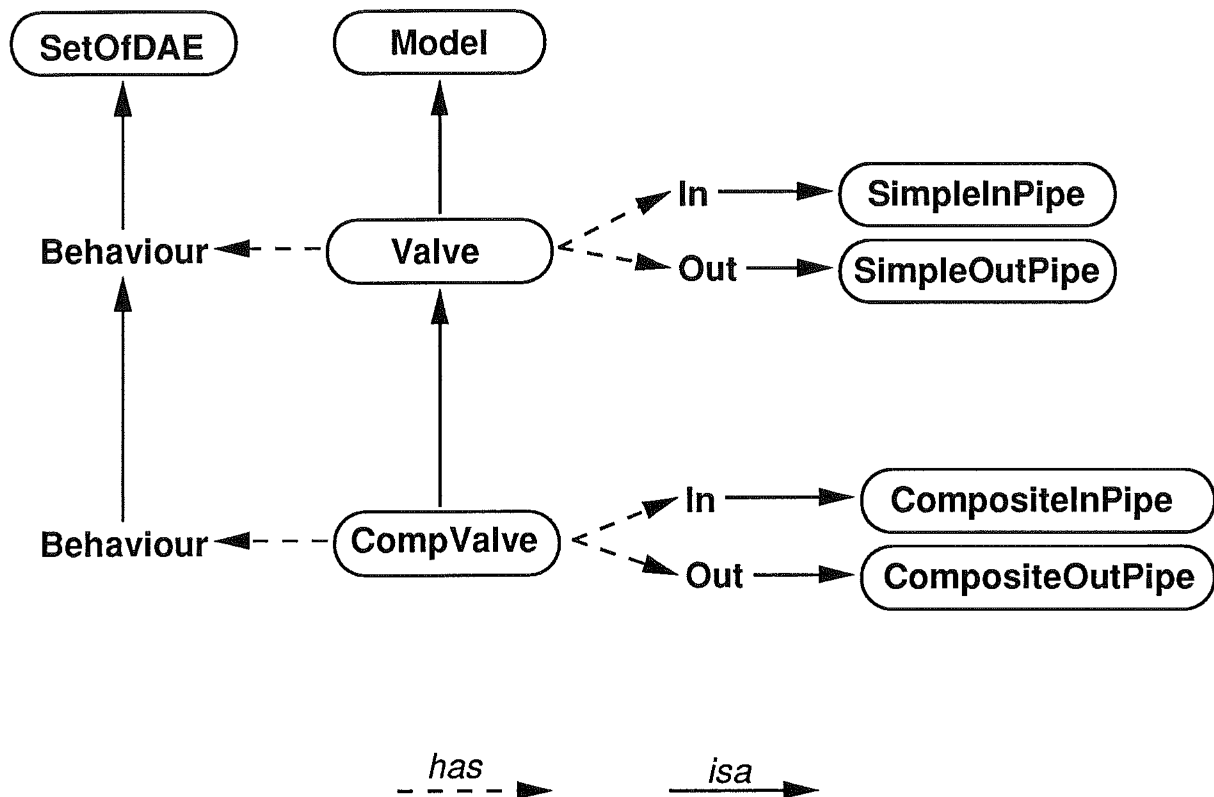

Figure 7.5 Some objects of the valve models.

to control the flow of cooling liquid. The model has a realization containing equations for the pressure drop depending on the position of the valve and of the flow. There are also two equations defining preservation of flow rate and temperature.

Now we want to define a valve model useful for composite flows in the product circuit. The only difference between this new valve and the previous one is the type of the flow terminals and an additional equation. We can use inheritance and define a valve model, called `CompValve`, as a subclass of `Valve` that redefines the flow terminals to be `CompositeInPipe` and `CompositeOutPipe`. The behaviour of this model is identical to the previous one, except that an equation defining preservation of flow composition has to be added. This is done be overriding the inherited realization with a specialization of it adding the new equation. This is done by using the *super* prefix for the superclass of the new realization. See Appendix C.2 for the definition of the `CompValve`. The structure diagram with the important components and their relations are shown in Figure 7.5.

The sensor models used in the reactor system are of two types: one type, called `SimpleSensor`, is used for typical across quantities like the

level and the temperature of the reactor, and one type, called `FlowSensor`, is used for composite flows.

A simple sensor has an input terminal for the physical quantity and an output terminal for the measured value. The terminals have their causality attributes defined to be either input or output. The reason for having defined causalities for the measurements is to prevent misuse of the model by attempt to affect the system through its sensors.

The flow sensor model has two composite flow terminals — an inflow and an outflow — and a terminal for the measurement value with a defined causality. Both types of sensor models have the most trivial kind of behaviour in this example. More complex sensor models, e.g., with dynamics, flow resistance or measurement noise, can easily be defined by using inheritance.

Parameter constraints are used in a few places in the reactor model to propagate parameter values between different model components. One example is in the media model where media parameters are bound to parameters in the media data terminal and thus propagated to the machine model. Another example is the number of components involved in the composite flow terminals and in the reaction model. This parameter is defined explicitly in the media model where it also bound to the value of 2. The value propagates to the `length` attribute of twelve different vector terminals in different places in the model. Propagation is in most of the cases defined implicitly through connections but there are also two explicit constraint expressions: one in the media and one in the machine model.

7.2 A transfer function model representation

The heat exchanger model of the cooling jacket may be quite critical for the simulation and for the tuning of the control system. For that reason it may be wise to try to identify a more accurate model based on measurements from the real process. Assume we have estimated an ARMAX model that we want to use for the cooling jacket instead of the one used previously.

From an identified ARMAX model for a single-input single-output system we can get the deterministic part of the model represented by two polynomials, $a(q)$ and $b(q)$ in the time delay operator q such that

$$a(q)y(k) = b(q)u(k)$$

where $u(k)$ is the sampled input and $y(k)$ is the sampled output. A model with n inputs and m outputs can then be represented by two m by n matrices of polynomials, $A(q)$ and $B(q)$, such that

$$A_{i,j}(q)y_i(k) = B_{i,j}(q)u_j(k)$$

In order to represent this in Omola we have to invent a new type of realization that we can call `DiscreteTransfer`:

```
DiscreteTransfer ISA Realization WITH
  attributes:
    NoInputs   TYPE Cardinal;
    NoOutputs  TYPE Cardinal;
    SampleTime TYPE Real;
    U TYPE Column[NoInputs];
    Y TYPE Column[NoOutputs];
    A TYPE Matrix[NoInputs,NoOutputs] OF Polynomial;
    B TYPE Matrix[NoInputs,NoOutputs] OF Polynomial;
END;
```

Once this new realization type is defined we can extend our simulator and all other tools that we want to use discrete transfer functions. It is the names and the meanings of the attributes defined in `DiscreteTransfer` that works as a common framework for representing transfer function models. A particular model based on a discrete transfer function defines a realization component that is a subclass of `DiscreteTransfer` and assigns values to the attributes. A tool accessing such a model has to know the names of the relevant attributes.

A transfer function for the cooling jacket

We can now use the new type of realization in a new version of the cooling jacket model. The new cooling jacket model will inherit all attributes from the old model.

The cooling jacket has three input signals and three output signals. This makes all together nine transfer functions but we can assume that one of the outputs, the flow rate of the cooling outlet, is identical to the flow rate of the cooling inlet and not effected by the other inputs, so we have actually six transfer functions.

In the transfer function realization of the new cooling jacket we have to assign values to the number of inputs and number of outputs attributes and we have to define the assumed ordering of the inputs and outputs by

assigning the U and Y vectors. In this example we have chosen to define the actual transfer function polynomials as parameters so that they can easily be changed.

```
CoolingJacket2 ISA CoolingJacket WITH
  realization:
    TransferFunc ISA DiscreteTransfer WITH
      parameters:
        A11,A12,A13,A21,A22,A23 TYPE Polynomial;
        B11,B12,B13,B21,B22,B23 TYPE Polynomial;
      attributes:
        NoInputs := 3;
        NoOutputs := 3;
        U:=[HeatIn.T, CoolIn.T, CoolIn.F];
        Y:=[HeatIn.Q, CoolOut.T, CoolOut.F];
        A:=[A11,A12,A13; A21,A22,A23; 0,0,1];
        B:=[B11,B12,B13; B21,B22,B23; 0,0,1];
  END;
END;
```

With a new cooling jacket model we can now define a new reactor system. This is easy since the interface of the cooling jacket model is the same as before. By using inheritance from ReactorSystem as well as from its realization ReactorStructure, the new reactor model becomes:

```
ReactorSystem2 ISA ReactorSystem WITH
  realization:
    ReactorStructure ISA Super.ReactorStructure WITH
      submodels:
        Cooling ISA CoolingJacket2;
    END;
END;
```

8. Conclusions

Models are essential in all kinds of control and process engineering. Since it is difficult to develop good models, and since models of real processes tends to become very complex, it is important to have advanced computer resources supporting development and reuse. Process and control design usually involve the use of a number of different tools supporting analysis, synthesis and simulation. All these tools are normally based on some kind of process model represented in some formal way. The problem is that the tools are intended to be used as stand-alone programs and therefore normally communicates with the user and not with other programs.

It is the goal of this thesis to define a standard model representation to be used as a basis in an integrated environment of cooperating CACE tools. Such a model representation can be thought of — and formalized — as a modeling language. In the introduction, the essential requirements of a new modeling language were identified. In short, the language must support a number of mathematical and logical frameworks for representing model behaviour, it must include concepts for structuring complex models and it should facilitate reuse of model components. Further, the language must be general and extendible in order to represent new modeling concepts in the future and it must be suitable for graphical manipulations and incremental model development.

Abstraction and decomposition are the foundations of structured modeling. This was discussed in Chapter 3 where also the basic concepts and the different types of model components were presented. The main structure entity is the *model*. A model has terminals, parameters and realizations. A realization is a description of behaviour and a model may have more than one. The behaviour of a model may be defined by a set of submodels and connections, thus forming the basis for hierarchical submodel decomposition. Differential and algebraic equations are the most general mathematical framework considered for model behaviour discriptions but also more special types of descriptions are discussed. Parameters that propagate over terminals and connections make it possible to check model consistency and prevent misuse of models. This is specially important when models are saved in libraries and reused by other persons.

The main contribution of this thesis is the specification of a modeling language to meet the requirements mentioned above. The language is called Omola and it was presented in Chapter 5. Omola extends the notion of structured modeling with concepts adopted from object-oriented programming. Omola can be viewed on two different levels: the basic level which is a general, object-oriented data representation language and the model representation level which defines model structuring components like model, terminal, realization, etc. The basic level of Omola is based on the class concept. A class may have attributes which may be other class definitions or typed variables. Attributes may be grouped into categories to indicate their different roles. Classes may be related by subclass relations thus forming an inheritance hierarchy. While the basic level of Omola is firmly set by the syntax the model representation level is more flexible; it can be extended with new concepts simply as agreements between different users and tools.

In Chapter 6 some operations on Omola models were discussed. In particular an algorithm for parameter propagation and consistency checking was outlined. Also model instantiation and simulation were discussed briefly.

Omola has been used in an application project studying structured modeling of chemical processes and found to meet most expectations [Nilsson, 1989]. The language has been supported by a prototype implementation in KEE and Lisp of an interactive modeling environment. It also serves as a basis in an on-going implementation project in C++ aiming at a kernel system for modeling, simulation and design.

8.1 Future work

Omola is designed to be extendible and many new modeling concepts can probably be represented within the current framework without substantial modifications. However, there are extensions to Omola that might be interesting to include in the future. Some of the extensions are of notational nature and quite easily adopted while others are more fundamental and require further research.

Regular structure notation

In [Nilsson, 1989] a notation for regular structures is suggested. A regular structure is a set of submodels of the same type connected in a regular fashion. This kind of structure appears sometimes in chemical process applications, in electrical circuits, and as approximations of distributed parameter models (partial differential equations).

It is suggested that connections of regular structures can be made by a matrix notation. For example, with an array of submodels declared by

```
Element[1..N] ISAN ElementModel;
```

where N is a structure parameter, we can connect these submodels by a single connection statement:

```
Element[1..N-1].Out  AT  Element[2..N].In;
```

It is easy to see how this can be translated into an equivalent set of $N-1$ single connection statements.

Multiple inheritance

Many object-oriented programming languages and frame based systems allow multiple inheritance, i.e., a class may inherit from more than one direct superclass.

Multiple inheritance is natural when a class can be considered to be a specialization of several separate concepts. From the example in the previous chapter, a chemical reactor may, from the first impression, be defined as a subclass of a tank model, containing the mass and energy balance equations, and as a subclass of a chemical reaction model. However, in reality it turns out to be not so simple. The main problem is due to lack of methods for defining interaction between the inherited parts. This may possibly be solved by naming conventions, i.e., attributes with identical names in the superclasses are assumed to refer to the same attribute in a descendant class. However, this is beyond standard object-oriented methodology and have to be investigated further.

In [Nilsson, 1989] and in the previous chapter, the chemical reactor model is instead structured by a media–machine submodel decomposition. The author also points out that he has not found any good examples where multiple inheritance could be used effectively. It is my impression that in most cases where multiple inheritance seems natural, submodel decomposition can be used just as well.

If multiple inheritance turns out to be useful in modeling it can be included in Omola without too much effort. Basically it requires the definition of the rules governing resolution of name conflicts, that can appear when attributes from more than one class are inherited.

Procedural specification

The version of Omola presented in this thesis is a purely declarative language and there are no concepts of procedural or functional knowledge. In many cases model behaviour is most naturally represented by functions or procedures in some traditional programming language or in a special language like Matlab [Moler et al., 1987]. The possibility to define procedural behaviour in other languages is particularly useful for development of control systems. The control algorithms may be evaluated in a simulation environment and then down-loaded into the real control system computer.

Procedural knowledge in models can be viewed as user defined methods. The methods may represent discrete time model behaviour as mentioned above but they may also be used to manipulate model classes. For example, a non-linear model may have a method for linearization that returns or inserts a new linear realization.

It is desirable that a method concept is introduced in a future development of Omola. The definition and invocation of methods will then be a part of Omola while the implementation of the methods is given in some other language. Methods may be implemented in Fortran or C but also in special purpose languages like Matlab for numeric computations, or Macsyma [Macsyma, 1983] for symbolic manipulations. For example, a linearization method may be implemented in Macsyma and also use Macsyma as an external tool to execute the method. In this way, Omola methods may serve as a common framework and the means of communication between tools in the environment.

Discrete event models

Many phenomena in the real world are most naturally represented by discrete event models rather than by continuous time differential equations. Discrete event models where discussed in Chapter 2. Sampled models are also common, specially in computer controlled systems and in models obtained by from measured data by parametric identification.

The extension of Omola towards discrete event and sampled models is closely related to the introduction of procedural specifications discussed

above. If Omola is going to fulfill the intentions of being a universal modeling language it is important that the notion of discrete events is incorporated in future versions. It is my strong belief that also this extension can be made in a graceful way.

9. References

ACSL (1986): *Advanced Continuous Simulation Language (ACSL), Reference Manual*, Mitchel and Gauthier Associates, Concord, Massachusetts.

ALLEN, E. M., R. H. TRIGG and R. J. WOOD (1983): "The Maryland Artificial Intelligence Group Franz Lisp Environment," TR-1226, University of Maryland, College Park, Maryland.

ANDERSSON, M, (1989): "An Object-Oriented Modelling Environment," *Proc. of the 1989 European Simulation Multiconference, Rome, June 7-9.*

ANON (1987): "Challenges to Control: A Collective View," *IEEE Transactions on Automatic Control*, **AC-32**, No. 4, April 1987, 275–285.

ÅRZÉN, K.-E. (1987): *Realization of Expert System Based Feedback Control TFRT-1029, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.*

ÅSTRÖM, K. J. and W. KREUTZER (1986): "System representations," *Proc. IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD).*

BEECH, D. (1987): "Groundwork for an Object Database Model," in B. Shriver and P. Wegner (Ed.): *Research Directions in Object-Oriented Programming*, The MIT Press, Cambridge, Massachusetts.

BIRTWISTLE, G. M., O-J. DAHL, B. MYHRHAUG and K. NYGAARD (1973): *Simula Begin*, Auerbach, Philadelphia, Pa.

BOOCH, G. (1983): *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Menlo Park, California.

BRÜCK, D. M. (1987): "Implementation Languages for CACE Software TFRT-3195, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

BRÜCK, D. M. (1990): "ANSI C++ Committee Meeting — December 15, 1989 TFRT-7421, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

CELLIER, F. E. (1979): *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Dissertation, Swiss Federal Institute of Techology Zürich.

ELMASRI, R. and B. N. SHAMKANT (1989): *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Redwood City, California.

ELMQVIST, H. (1975): *SIMNON — An Interactive Simulation Program for Nonlinear Systems — User's Manual TFRT-3091, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.*

ELMQVIST, H. (1978): *A Structured Model Language for Large Continuous Systems TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.*

ELMQVIST, H. (1986): *LICS — Language for Implementation of Control Systems TFRT-3179, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.*

ELMQVIST, H. and S. E. MATTSSON (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modeling," *IEEE Control Systems Magazine*, **9**, No. 1, January 1989, 53–58.

ELMQUIST, H., K. J. ÅSTRÖM, T. SCHÖNTHAL and B. WITTENMARK (1990): *Simnon — User's Guide for MS-DOS Computers*, SSPA Systems, Göteborg, Sweden.

FAYEK, A. M., R. C. HUNTSINGER and R. E. CROSBIE (1987): "The New Continuous Systems Simulation Environment in the Simscript II.5 Simulation Language," *Preprints of the International Symposium on AI, Expert Systems and Languages in Modelling and Simulation*, IMACS, June 2–4 1987, Barcelona, Spain.

GOLDBERG, A. AND D. ROBSON (1983): *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts.

KEENE, S. E. (1989): *Object-Oriented Programming in Common Lisp*, Addison-Wesley, Reading, Massachusetts.

KORN, G. A. and J. V. WAIT (1978): *Digital Continuous-System*

Simulation, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

KREUTZER, W. (1986): *System Simulation: Programming Styles and Languages.*, Addison-Wesley Publishers Limited.

LIND, M (1987): "Multilevel Flow Modelling – Basic Concepts," Espri Project 96, working paper.

MACSYMA (1983): *MACSYMA Reference Manual*, The Mathlab Group Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

MATTSSON, S. E. (1988): "On Model Structuring Concepts," *Preprints of the 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, August 23–25 1988, P.R. China, pp. 269–274.

MATTSSON, S. E. (ED) (1989a): "New Tools for Model Development and Simulation TFRT-7438, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

MATTSSON, S. E. (1989b): "On Modelling and Differential/Algebraic Systems," *Simulation*, **52**, No. 1, 24–32.

MATTSSON, S. E. (1989c): "Modelling of Interactions between Submodels," *Proc. of the 1989 European Simulation Multiconference, Rome, June 7-9.*

MATTSSON, S. E. and M. ANDERSSON (1989): "An Environment for Model Development and Simulation TFRT-3205, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

MEYER, B. (1988): *Object-oriented Software Construction*, Prentice Hall.

MINSKY, M. (1975): "A framework for representing knowledge," in P. H. Winston (Ed.): *The Psychology of Computer Vision*, McGraw-Hill, New York.

MOLER, C., J. LITTLE and S. BANGERT (1987): *PRO-MATLAB User's Guide*, The MathWorks, Inc., Sherborn, MA.

MOORE, R. L., L. B. HAWKINSON, M. LEVIN, A. G. HOFFMANN, B. L. MATTHEWS and M. H. DAVID (1987): "Expert system methodology for real-time process control," *Proc. 10th IFAC World Congress*, pp. 274–281.

NILSSON, B (1989): *Structured Modelling of Chemical Processes —*

An Object-Oriented Approach TFRT-3203, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B (1989b): "Structured Modelling of Chemical Processes with Control Systems TFRT-7439, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

PETERSON, J. L. (1981): *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

PETZOLD, L. R. (1982): "A Discription of DASSL: A Differential/Algebraic Systems Solver,".

PIELA, P. C. (1989): *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*, Dissertation, Carnegie-Mellon University, Pittsburg, Pennsylvania.

PRITSKER A. A. B. and HURST N. R. (1973): "GASP-IV: A Combined Continuous Continuous/Discrete Fortran-Based Simulation Language," *Simulation*, September 1973, 65–70.

RASMUSSEN, J and M. LIND (1981): "Coping with Complexity," Risø-M-2293, Risø National Laboratory, Roskilde, Denmark.

RIMVALL, C. M. (1986): *Man-Machine Interfaces and Implementational Issues in Computer-Aided Control System Design*, Dissertation, Swiss Federal Institute of Technology Zurich.

SCHENCK, D. (1988): *Express*, ISO TC184/SC4/WG1, Draft N210, February 1988.

SCS (1967): "The SCi Continuous System Simulation Language (CSSL)," *Simulation*, **9**, No. 6, 281–303.

SIMULATION (1988): "Catalog of Simulation Software," *Simulation*, **51**, No. 4, 136–156.

STEFIK, M. and D. G. BOBROW (1986): "Object-Oriented Programming: Themes and variations," *AI Magazine*, **6:4**, pp. 40–62.

STROUSTRUP, B. (1986): *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts.

TAN, C.-Y. and J. M. MACIEJOWSKI (1989): "The GE MEAD Computer-Aided Control Engineering Environment," *1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD)*, December 16, 1989, Tampa, Florida, pp. 72–77.

TAYLOR, J. H., D. K. FREDERICK, C. M. RIMVALL and H. A. SUTHERLAND (1989): "The GE MEAD Computer-Aided Control Engineering Environment," *1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD)*, December 16, 1989, Tampa, Florida, pp. 16–23.

A. Syntax Rules

This appendix presents the formal syntax rules of Omola on extended Backus-Naur (EBNF) form. We have used lower case letters for non-terminal symbols. Terminal symbols are written in capital letters or whithin double quotes. A vertical bar "|" separates alternatives, an expression within square brackets is optional, an expression followed by an asterisk "*" can be taken zero or many times while an expression followed by a plus "+" can be taken one or many times. Operators in expressions are parsed with the following precedence levels; binary operators are left associative except ^ which is right associative:

1.	unary -, ^	2.	*,/
3.	+,-	4.	range operator
5.	relational operators	6.	not
7.	and	8.	or

```
class_definitions ->
  (name class_def)*

class_def ->
  super_class_def ["with" class_body "end"]

super_class_def ->
  isa (IDENT | "this" "." IDENT | "super" "." IDENT)

class_body ->
  body-item* tag_body

body_item ->
  name_list (class_def |
             type_declaration |
             variable_binding) ";"

tag_body ->
  (TAG body-item* | special_tag_body)*
```

```
special_tag_body ->
  equation_tag equation* |
  constraint_tag equation* |
  connection_tag connection*

equation ->
  conditional_expression "=" conditional_expression ";"

connection ->
  terminal AT terminal ";"

type_declaration ->
  "type" type_designator [variable_binding]

type_designator ->
  "(" name_list ")" |
  simple_type_designator |
  struct_type_designator ["of" simple_type_designator]

simple_type_designator ->
  "real" | "integer" | "cardinal" | "number" |
  "complex" | "string" | "generic" | "name" | "symbol"

struct_type_designator ->
  "matrix" "[" expr "," expr "]" |
  "row" "[" expr "]" |
  "column" "[" expr "]" |
  "list" |
  "range"

variable_binding ->
  ":=" conditional_expr

conditional_expr ->
  expr |
  "if" expr "then" expr "else" conditional_expr

expr ->
  expr "or" expr |
```

```
  expr "and" expr |
  "not" expr |
  expr rel_op expr |
  expr ".." expr |
  expr add_op expr |
  expr mul_op expr |
  expr "^" expr |
  "-" expr |
  primary

primary ->
  variable |
  matrix |
  polynomial |
  REAL | INTEGER | STRING |
  "(" expr ")"
  function_designator

matrix ->
  "[" rows "]"

rows ->
  expr_list (";" expr_list)*

expr_list ->
  conditional_expr ("," conditional_expr)*

polynomial ->
  "{" (c_poly | r_poly) "}"

c_poly ->
  expr_list;

r_poly ->
  expr ":" expr_list

function_designator ->
  IDENT "(" [expr_list] ")"
```

```
name ->
  IDENT

name_list ->
  name ("," name)*

variable ->
  IDENT ( "." IDENT | "[" expr_list "]")*

terminal ->
  variable

isa ->
  "isa" | "isan"

equation_tag ->
  "equation:" | "equations:"

constraint_tag ->
  "constraint:" | "constraints:"

connection_tag ->
  "connection:" | "connections:"

rel_op ->
  ">" | "<" | "==" | ">=" | "<="

add_op ->
  "+" | "-"

mul_op ->
  "*" | "/"
```

B. Data Types and Model Classes

In this appendix we will give a complete list of all types currently available in Omola. We will also give the definitions of all predefined model component classes.

B.1 Omola Types

Type	Declaration	Example litteral	Comment
Cardinal	`cardinal`	`1`	
Complex	`complex`	`1i-1`	
Enumeration	`(a,b,c)`	`a`	1.
Generic	`generic`		2.
Integer	`integer`	`-1`	
List	`list`	`[a,b,c]`	
Matrix	`matrix[m,n]`	`[1, 0; 0, 1]`	
	`row[m]`	`[1, 0]`	
	`column[n]`	`[1; 0]`	
Name	`name`	`T.A`	3.
Number	`number`	`3.14`	
Polynomial	`polynomial`	`{1,2,3}` or `{2: 1,1}`	
Range	`range`	`0..10`	
Real	`real`	`3.14`	
String	`string`	`"m/s"`	
Symbol	`symbol`	`Tank`	

Comments

1. The type designator may be any list of symbols.
2. Any type is valid data.

3. A Name is a symbolic reference that can be resolved to a class or a variable.

B.2 Omola Model Classes

```
Model ISA Class WITH
    primary_realization TYPE name;
  parameters:
  terminals:
  variables:
  realizations:
  constraints:
END;

Terminal ISA Class;

SimpleTerminal ISA Terminal WITH
  value TYPE Real;
  default TYPE Real;
  direction TYPE (Across, In, Out);
  causality TYPE (Read, Write);
  variability TYPE
    (TimeVarying, Parameter, Constant) := TimeVarying;
  low_limit TYPE Real;
  high_limit TYPE Real;
  unit TYPE String;
  quantity TYPE Symbol;
END;

RecordTerminal ISA Terminal WITH
  components:
END;

VectorTerminal ISA Terminal WITH
  length TYPE Cardinal;
  comptype ISA Terminal;
END;
```

```
Realization ISA Class;

SetOfDAE ISA Realization WITH
  parameters:
  variables:
  equations:
  constraints:
END;

Structure ISA Realization WITH
  submodels:
  connections:
  constraints:
END;

Parameter ISA Class WITH
  value TYPE Real;
END;

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

C. Algorithms

This appendix gives the algorithms for searching a model context for all equations refering to a particular variable. The algorithms are used in Section 6.2 for deriving variable values. Variables as well as equations are represented as *contexts*, i.e., as a class object with a component chain.

The high level procedures are given in Lisp code below. The algorithms are efficient in the sense that they regard the scope rules and search only those components which may refer to the variable. The scope rules are, quoted from Chapter 5, the following. A connection defined in a realization may refer to

1a terminals of the model,

1b components of terminals of the model, or

1c terminals of submodels defined in the realization.

An equation or a binding expression in a model may refer to

2a variable attributes of the model,

2b terminals of the model, or

2c attributes of terminals of the model, or

2d variables of realizations.

An equation or a binding expression in a realization may refer to

3a variable attributes of the realization or the model,

3b terminals of the model,

3c attributes of terminals of the model, or

3d parameters of submodels defined in the realization.

In the procedures, some operations have comments refering to the particular scope rule numbers.

```
(defun find_equations (v-context)
  ;; Search the context and return a list of equations
  ;; and contexts that refer to a particular variable.
  (cond ((top-terminal? v-context)
```

```
         (find-equs-top-term v-context))
        ((terminal-comp? v-context)
         (find-equs-term-comp v-context))
        ((parameter? v-context)
         (find-equs-parameter v-context))
        ((variable? v-context)
         (find-equs-variable v-context))
        (t
         (error "Not a variable context."))))


(defun find-equs-top-term (c)
  ;; Return all equations refering to the terminal contex c.
  (let* ((m (owner c))            ; the model context
         (r (prime-realization m)) ; the primary realization
         (s (owner m))) ; super model
    (append
     (filter (equations m) (last c))         ; 2b
     (filter (equations r) (last c))         ; 1a, 3b
     (filter (equations s) (last2 c)))))     ; 1c

(defun find-equs-term-comp (c)
  ;; Return all equations refering to terminal component c.
  (let* ((m (owner (top-term c))) ; the model
         (r (prime-realization m)); its prime realization
         (tc (term-comp c)))      ; Context of comp relative
                                  ; top term.
    (append
     (filter (equations m) tc)     ; 2c
     (filter (equations r) tc))))  ; 1b, 3c

(defun find-equs-parameter (c)
  ;; Return all equations refering to parameter context c.
  (let ((s (owner (owner c)))) ; a model or a realization
    (append
     (find-equs-variable c)
     (filter (equations s) (last2 c))))) ; 3d

(defun find-equs-variable (c)
```

```
(let ((o (owner c))) ; model or realization context
  (cond ((model? o)
         (append
          (filter (equations o) (last c))          ; 2a
          (filter (equations (prime-realization o)) ; 3a
                  (last c))))
        ((realization? o)
         (append
          (filter (equations o) (last c))          ; 3a
          (filter (equations (owner o))
                  (last2 c))))))                   ; 2d
```

The algorithms use a number of low-level procedures which are not given as code but listed together with short descriptions below.

First we have some basic context operations. The concept of *context* was introduced in Chapter 6. In the following function descriptions a string notation is used for contexts.

Owner *context* [Function]
: Return the owner's context. For example, `owner("A.B.C")` returns the context `"A.B"` since `B` is the owner of `C`.

Last *context* [Function]
: Return the "last" part context. For example, `last("A.B.C")` returns the context `"C"`.

Last2 *context* [Function]
: Return the "two last" part context. For example, `last("A.B.C")` returns the context `"B.C"`.

Concat *context1 context2* [Function]
: Return the concatenated context. For example, `concat("A.B","C.D")` returns the context `"A.B.C.D"`.

Top-Term *term-context* [Function]
: Return the context of the top terminal given the context of a terminal component. By *top terminal* is meant a terminal directly owned by a model. For example, `topterm("M.T.A.B")` returns the context `"M.T"` if `M` is a model and `T` is one of its terminals.

Term-Comp *term-context* [Function]
: Return the context of a terminal component relative its top terminal. This is the complementary function to previous func-

tion **TopTerm**. For example, `topterm("M.T.A.B")` returns the context `"A.B"` if `M` is a model `T` is one of its terminals.

Here follows the descriptions of the other functions used in the algorithms above.

Equations *context* [Function]
Return all equations and bindings that are defined in *context*. The returned equations are represented with contexts.

Filter *equations var-context* [Function]
Return only those *equations* that have references to *var-context*.

Model? *context* [Predicate]
Return **true** if the *context* is a model context.

Parameter? *context* [Predicate]
Return **true** if *context* is a parameter context.

Prime-realization *model-context* [Function]
Given a *model-context* return the context of its primary realization.

Realization? *context* [Predicate]
Return **true** if the *context* is a realization context.

Terminal-Comp? *context* [Predicate]
Return **true** if *context* is a terminal component context.

Top-Terminal? *context* [Predicate]
Return **true** if *context* is the terminal of a top terminal, i.e., a terminal directly owned by a model.

Variable? *context* [Predicate]
Return **true** if *context* is a variable context.

D. Tank Reactor Example

D.1 The Reactor System and its Submodels

`ReactorSystem` is the top-most model in this example. Its code is given in this section followed by the code of the main submodel `ReactorVessel` and its submodels `ReactorVesselMachineModel` and `MediaModel`. The `CoolingJacket` model is also found in this section.

```
ReactorSystem ISA Model WITH
  %
  % Reactor vessel with cooling jacket,
  % valves and sensors.
  %
  terminals:
    CoolIn  ISA SimpleInPipe;
    CoolOut ISA SimpleOutPipe;
    LiquidIn ISA CompositeInPipe;
    LiquidOut ISA CompositeOutPipe;
    ControlAct ISA RecordTerminal WITH
      Cool ISA SimpelTerminal;
      Feed ISA SimpelTerminal;
    END;
    Sensors ISA RecordTerminal WITH
      L, T, F ISA SimpleTerminal;
    END;
  realization:
    ReactorStructure ISA Structure WITH
      submodels:
        Reactor ISA ReactorVessel;
        Cooling ISA CoolingJacket;
        CValve ISA Valve;
        FValve ISA Valve;
        LSensor ISA LevelSensor;
```

```
        TSensor ISA TemperatureSensor;
        FSensor ISA FlowSensor;
      connections:
        CoolIn AT CValve.IN;
        CValve.Out AT Cooling.CoolIn;
        Cooling.CoolOut AT CoolOut;
        LiquidIn AT FValve.In;
        FValve.Out AT Reactor.LiquidIn;
        LiquidOut AT FSensor.In;
        FSensor.Out AT LiquidOut;
        FSensor.Flow AT Sensors.F;
        LSensor.In AT Reactor.Level
        LSensor.Out AT Sensors.L;
        TSensor.In AT Reactor.T;
        TSensor.Out AT Sensors.T;
        Cooling.HeatIn AT Reactor.HeatOut;
        ControlAct.Cool AT CValve.X;
        ControlAct.Feed AT FValve.X;
      END;
END;

ReactorVessel ISA Model WITH
  %
  % Media -- data decomposition of reactor.
  %
  terminals:
    Inlet   ISA CompositeInPipe;
    Outlet  ISA CompositeOutPipe;
    Cooling ISA HeatFluxOut;
    Level   ISA LevelData;
    Temp    ISA TemperatureTerminal;
  realization:
    MediaMachine ISA Structure WITH
      submodels:
        Machine ISA ReactorVesselMachineModel;
        Media   ISA MediaModel;
      connections:
        Inlet AT Machine.Inlet;
        Outlet AT Machine.Outlet;
```

```
        Cooling AT Machine.Cooling;
        Level AT Machine.Level;
        Temp AT Machine.Temp;
        Machine.MD AT Media.MD;
    END;
END;

ReactorVesselMachineModel  ISA Model WITH
  %
  %  A  liquid tank  with dynamic behaviour.
  %
  terminals:
    Inlet ISA CompositeInPipe;
    Outlet ISA CompositeOutPipe;
    Cooling ISA HeatFluxOut;
    Level ISA  LevelData;
    Temp ISA  TemperatureTerminal;
    MD  ISA  MediaDataTerminal;
  parameters:
    Area ISA Parameter;
  constraints:
    MD.C.length = Outlet.C.length = Inlet.C.length;
  realization:
    Behaviour ISA SetOfDAE WITH
      equations:
        % Mass balance:
        Area*dot(Level) = Inlet.F - Outlet.F;
        V = Area*Level;
        % Component mass balance:
        dot(N) = Inlet.F*Inlet.C - Outlet.F*Outlet.C +
          V*MD.R;
        C=N/V;
        % Energy balance:
        dot(e) = MD.roh*(Inlet.F*Inlet.T*MD.Cp
                         - Outlet.F*Outlet.T*MD.Cp)
                 + MD.Q - Cooling.Q;
        e = MD.roh*V*MD.Cp*Outlet.T;
        % Homogeneity:
        Cooling.T = Outlet.T;
```

```
        Outlet.C = C;
        MD.C = C;
    END;
END;

MediaModel ISA Model WITH
  %
  % Media model of ReactorVessel.
  % Describing the reaction:  A -> B
  %
  terminal:
    MD ISA MediaDataTerminal;
  parameters:
    roh TYPE Real;  % Density
    Cp  TYPE Real;  % Heat capacity
    dH  TYPE Real;  % Reaction entalphy
    Ea  TYPE Real;  % Activation energy
    R   TYPE Real := 8.3143;
    noComp TYPE Cardinal := 2;
  constraints:
    MD.roh = roh;
    MD.Cp = Cp;
    MD.R.length = MD.C.length = noComp;
  realization:
    Behaviour ISA SetOfDAE WITH
      equations:
        MD.R = [rr -rr]';
        rr = k*MD.C(1);
        k = k0*exp(-Ea/R/MD.T);
        MD.Q = dH*rr;
    END;
END;

CoolingJacket ISA Model WITH
  %
  % Cooling jacket of tank reactor.
  %
  terminals:
    CoolIn ISA SimpleInPipe;
```

```
    CoolOut ISA SimpleOutPipe;
    HeatIn ISA HeatTransferIn;
  parameters:
    Area TYPE Real;
    Cv   TYPE Real;  % Heat transfer coeff.
    Cp   TYPE Real;  % Heat capacity of cooling media.
    roh  TYPE Real;  % Density of cooling media.
  realization:
    Equations ISA SetOfDAE WITH
      equations:
        % energy balance:
        HeatIn.Q = deltaT*(Cp*roh*CoolIn.F);
        deltaT = CoolOut.T - CoolIn.T;
        % mass balance:
        CoolIn.F = CoolOut.F;
        % heat transfer:
        HeatIn.Q = Cv*Area*(HeatIn.T - Tav);
        Tav = CoolingIn.T + deltaT/2;
    END;
END;
```

D.2 Standard Component Models

In this section, the code of `Valve`, `CompValve`, `SimpleSensor` and `FlowSensor` is given.

```
Valve ISA Model WITH
  %
  % Valve  model.
  % X is valve position where X=1 is fully open
  %  and X=0 is closed.
  %
  terminals:
    In  ISA SimpleInPipe;
    Out ISA SimpleOutPipe;
    X   ISA ActuatorTerminal;
  parameter:
    k  TYPE Real := 1.0;   % Gain
  realization:
```

```
    Behaviour  ISA SetOfDAE WITH
      equations:
        Out.F*ABS(Out.F) = k*X*deltaP;
        deltaP = In.P - Out.P;
        % Flow and teperature preservation:
        Out.F = In.F;
        Out.T = In.T;
    END;
END;

CompValve ISA Valve WITH
  %
  % Refinement   of  Valve  that takes
  % composite  flows.
  %
  terminals:
    In  ISA CompositeInPipe;
    Out ISA CompositeOutPipe;
  realization:
    Behaviour ISA Super.Behaviour  WITH
      equation:
        In.C = Out.C;
    END;
END;

SimpleSensor ISA Model WITH
  terminals:
    In  ISA  InTerminal;
    Out ISA OutTerminal;
  parameter:
    k TYPE Real := 1.0; % Gain
  realization:
    Behaviour ISA SetOfDAE WITH
      equation:
        Out = k*In;
    END;
END;

FlowSensor ISA Model WITH
```

```
  %
  % Flow sensor for composite flows.
  %
  terminals:
    In  ISA CompositeInPipe;
    Out ISA CompositeOutPipe;
    Flow ISA OutTerminal;
  parameter:
    k TYPE Real := 1;  %   Gain
  realization:
    Behaviour ISA SetOfDAE WITH
      equations:
        In = Out;
        Flow = k*Out.F;
    END;
END;
```

D.3 Terminals

Here follows the code for terminal types used in the reactor example.

Structured Terminals

```
CompositionTerminal ISA VectorTerminal WITH
  length := 2;
  CompType ISA ConcentrationTerminal;
END;

SimpleFlowPipe ISA RecordTerminal WITH
  components:
    F ISA SimpleTerminal;
    T ISA TemperatureTerminal;
    P ISA PressureTerminal;
END;

SimpleInPipe ISA SimpleFlowPipe WITH
  F ISA FlowInTerminal;
END;
```

```
SimpleOutPipe ISA SimpleFlowPipe WITH
  F ISA FlowOutTerminal;
END;

CompositeFlowPipe ISA SimpleFlowPipe WITH
  components:
    C ISA CompositionTerminal;
END;

CompositeInPipe ISA CompositeFlowPipe WITH
    F ISA FlowInTerminal;
END;

CompositeOutPipe ISA CompositeFlowPipe WITH
    F ISA FlowOutTerminal;
END;

HeatTransferIn ISA RecordTerminal WITH
  components:
    F ISA HeatFluxIn;
    T ISA TemperatureTerminal;
END;

HeatTransferOut ISA RecordTerminal WITH
  components:
    F ISA HeatFluxOut;
    T ISA TemperatureTerminal;
END;

FlowData ISA FlowTerminal WITH
  direction := across;
  causality := out;
END;

LevelData ISA SimpleTerminal WITH
  quantity  := position;
  unit      := "m";
  direction := across;
```

```
END;

MediaDataTerminal ISA RecordTerminal WITH
  %
  % Terminal for connecting media and
  % machine model in reactor vessel.
  %
  components:
    % Reaction rate:
    R ISA VectorTerminal WITH
      CompType ISA SimpleTerminal;
    END;
    % Concentration:
    C ISA VectorTerminal WITH
      CompType ISA ConcentrationTerminal;
    END;
    % Temp:
    T ISA TemperatureTerminal;
    % Parameters:
    roh, Cp ISA ParameterTerminal;
END;
```

General, Simple Terminals

```
ParameterTerminal ISA SimpleTerminal WITH
  variability := parameter;
END;

ActuatorTerminal ISA SimpleTerminal WITH
  range := 0.0..1.0;
END;

InTerminal ISA SimpleTerminal WITH
  causality := in;
END;

OutTerminal ISA SimpleTerminal WITH
  causality := out;
END;
```

Physical Quantity Terminals

```
PressureTerminal ISA SimpleTerminal WITH
  quantity  := pressure;
  unit      := "kPa";
  direction := across;
END;

TemperatureTerminal ISA SimpleTerminal WITH
  quantity  := temperature;
  unit      := "K";
  direction := across;
END;

FlowTerminal ISA SimpleTerminal WITH
  quantity := volumetric_flow;
  unit     := "m^3/s";
END;

FlowOutTerminal ISA FlowTerminal WITH
  direction := out;
END;

FlowInTerminal ISA FlowTerminal WITH
  direction := in;
END;

HeatFluxTerminal ISA SimpleTerminal WITH
  quantity := heat_flux;
  unit     := "J/s/m^2";
END;

HeatFluxIn ISA HeatFluxTerminal WITH
  direction := in;
END;

HeatFluxOut ISA HeatFluxTerminal WITH
  direction := out;
END;
```

```
ConcentrationTerminal ISA SimpleTerminal WITH
  quantity  := mole_concentration;
  unit      := "kmole/m^3";
  direction := across;
END;
```