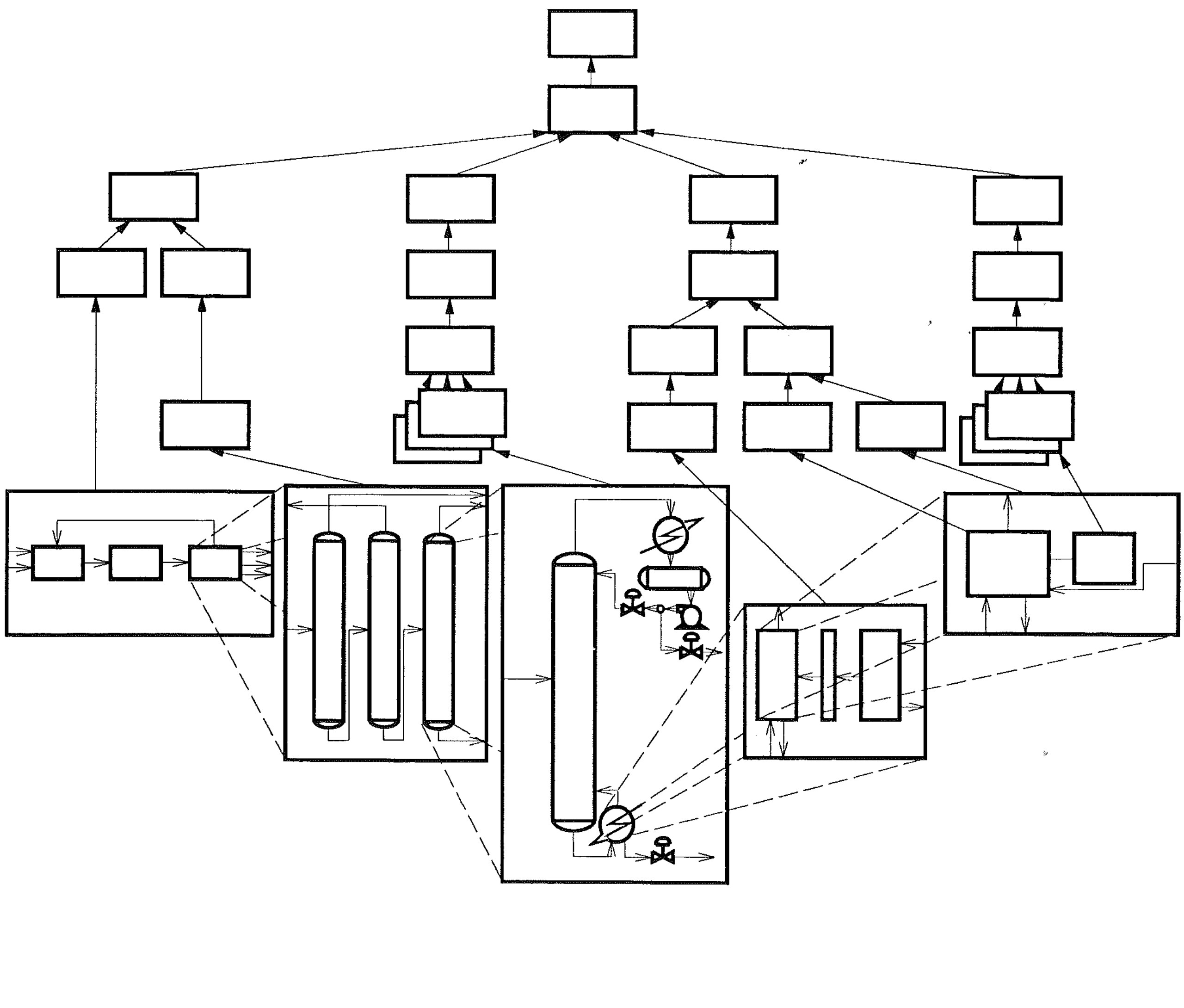
Object-Oriented Modeling of Chemical Processes

Bernt Nilsson



[illegible]

Department of Automatic Control, Lund Institute of Technology

Ryggtitel som på Bo Bernhardsson's avhandling

Bernt Nilsson Object-Oriented Modeling of Chemical Processes

ISSN 0280–5316
ISRN LUTFD2/TFRT–1041–SE

15 cm/m

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* DOCTORAL DISSERTATION *Date of issue* August 1993 *Document Number* ISRN LUTFD2/TFRT--1041--SE
Author(s) Bernt Nilsson	*Supervisor* Karl Johan Åström and Sven Erik Mattsson *Sponsoring organisation* Swedish Board of Technical Development under contract 87-02503 and 89-2740

Title and subtitle

Object-Oriented Modeling of Chemical Processes

Abstract

Models are important for almost all engineering activities. This thesis presents an object-oriented methodology for development of models for complex chemical processes. Although we primarily deal with chemical processes much of the methodology can also be applied to other complex technical systems. An object-oriented modeling language, Omola, is used as a tool throughout the thesis.

The main results of the thesis are guidelines for structure and class hierarchy decomposition. A chemical plant model is decomposed into smaller and smaller pieces following the guidelines for model structure decomposition. The libraries of predefined models are organized following the guidelines for class decomposition. These guidelines cooperate to create well defined model modules that are easy to reuse in new applications.

Other important results are particular modeling methods. Medium and machine decomposition is a method to separate the description of the process media and the processing unit machine. Methods for control system abstraction are also presented together with parameterization methods for reusable models with complex interiors. The guidelines and these methods are used in a chemical plant application and they are shown to be successful in an Omola implementation.

Some modeling methods need extensions in current Omola. Examples are concepts for regular structures as well as abstract and parameterized classes. A method for batch process modeling is also suggested that uses automatic switching between different internal descriptions.

Key words

Modeling; process models; object-oriented modeling; systems representations; hierarchical systems; computer simulation; simulation languages; computer-aided design; software tools.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280–5316		*ISBN* 0280–5316
Language English	*Number of pages* 166	*Recipient's notes*
Security classification		

(166 sid) 01

blank
02

Object-Oriented Modeling of Chemical Processes

The illustration on the front page shows the structure and class hierarchies of a chemical plant example. The front cover example is discussed in more detail in Appendix C. The hierarchy decompositions are discussed in Chapter 4 and 5.

Object-Oriented Modeling of Chemical Processes

Bernt Nilsson

Department of Automatic Control, Lund Institute of Technology
Lund, August 1993

5

To Åsa and Cecilia

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

ISSN 0280-5316
ISRN LUTFD2/TFRT–1041–SE

4

Contents

Preface

The following thesis belong somewhere in the borderland of Computer Science, Automatic Control and Chemical Engineering. The thesis studies a new modeling language, called Omola, in modeling chemical processes for control analysis and design. Writing an interdisciplinary thesis has many potential pitfalls. The scope necessary becomes broad and the audience, with experience from all fields, is small. This thesis is primarily written for a reader with knowledge of automatic control and chemical engineering. It is broad in the sense that it touches upon several different areas that each, in its own right, could be the topic of further research. The intention behind the work has been to explore the use of Omola in a particular application and to study the use of traditional chemical modeling concepts in Omola. Another pitfall with an interdisciplinary thesis is that it run the risk of being considered inappropriate for all fields. Hopefully, this is not the case here. Instead my hope is that the work may generate a new modeling methodology and new modeling concepts.

Acknowledgements

Many people have helped me in producing this thesis and it is a great pleasure to express my gratitude to the following good friends.

I am very glad to express my sincere gratitude to Karl Johan Åström. He has created the necessary conditions for a stimulating research atmosphere and has been a consistent source of enthusiasm. His great knowledge of people and broad overviews of the state of different research areas have been of great help. He has greatly improved the manuscript by constructive criticism on several chapters.

I thank Sven Erik Mattsson. He has supervised me throughout this work and has been an important source of inspiration particularly in the first phase of this work. As the leader of the CACE group he has always pointed out the importance of application driven interaction with the development of a software. I am thankful for the opportunity to interact with the development of OmSim. Sven Erik Mattsson has read several versions of the manuscript and I am grateful for his comments.

I also would like to thank Mats Andersson, a key person in the CACE group and the inventor of Omola, for patiently answering my questions about Omola, taking my criticisms of OmSim with calm and for always beating me on the 10 km track at Skrylle. Mats Andersson has also made constructive comments on several chapters of the manuscript.

I am also very glad to express my sincere gratitude to Karl Erik Årzén. He was a very important person during the "hard years", when I was working on another project. I would particularly like to thank him for always letting me share his deep insight to the fantastic world of G2. I also would like to thank him for reading several versions of the manuscript and for constructive suggestions and comments that have greatly improved the quality of the thesis.

My colleagues at the Department form a friendly and inspiring community and I am grateful for their encouragement and help. I specially would like to thank the following: Tim Berg for correcting my English under time pressure, Leif Andersson for maintaining excellent computer and text generating facilities and for patiently answering all my TEX-questions, Dag Brück and Tomas Schönthal for the great work of implementing the OmSim environment and always promising me that my bugs will be corrected in the next version, and Per Hagander for many long talks during my first years at the department.

I also would like to thank Björn Tyreus for several important discussions in the beginning of this work.

The work has been partly supported by the Swedish Board of Technical Development.

Finally I thank Åsa, my beloved wife, and Cecilia, our lovely daughter, for their encouragement and their support in everyday life.

B.N.

1

Introduction

Models are important for almost all engineering activities. They are used in design and analysis of new and existing systems and they are an increasingly important ingredient in control systems. Requirements on increased productivity and quality make the systems more complex which means increased engineering effort in design. At the same time there are demands for shorter development time to a lower cost. Tools that increase the engineering efficiency are therefore important. This thesis presents some new ideas for development of models for complex chemical processes. Although we primarily deal with chemical processes much of the methodology can also be applied to other complex technical systems.

The requirements on future chemical processes will increase. Competition on the international market together with government regulations on the production, impose challenging constraints on process design and operation. Recycling of both energy and material is required in order to satisfy environmental regulations and economic demands. This makes the process complex. Flexible production, both regarding quality and quantity, due to changing market demands and just-in-time production, is another important requirement. Safety is yet another vital consideration. Hazardous release of chemicals involves enormous costs. There will also be increasingly hard regulations on transport of chemicals which will force the production sites to be near the consumers. Such processes will be small, customized, miniaturized processes. These requirements make the process design complex and the process operation difficult. The process control system must handle not only the classical regulation problem but also rapid operational changes. The system must have facilities for discontinuous operational changes, e.g. start up, planned and emergency shut downs, reduced production, etc. Process diagnosis of failure situations together with qualitative operational diagnostics for improved quality will be common ingredients in future process information systems.

1.1 Engineering Design

Engineering design of systems, particularly large and complex systems, involves some basic principles to handle complexity. Here are a few general principles that are useful when dealing with complex systems: modularization, abstraction and reuse. A design methodology of chemical processes that uses these principles is found in [Douglas, 1988].

Modularization

Breaking a large system into smaller subsystems, called modules, is a natural way to handle complexity. The basic idea is that the subsystems are easier to deal with than the total system. It is important that the modules are isolated with well defined interaction. Modularization can be done hierarchically at a number of levels where modules are decomposed into submodules, which are decomposed into subsubmodules, etc. This creates a hierarchy of modules. Each decomposition is made to make design of each individual system easier. There are no direct rules for modularization. The physical structure of components is a common guideline for modularization in process industry.

Abstraction

Another aspect of modularization is abstraction. A good decomposition gives modules with small interaction with surrounding modules and with its super module. A well abstracted module is easy to use in a larger structure because the user does not have to know about the internal details of the module.

Reuse

In design of large systems it is often possible perform the decomposition so that the modules have a well known design. It is then possible to build a library of modules that can be used for different purposes. The common use of unit operations is a typical example. It should therefore be possible to reuse previously defined modules. Reuse is the driving force in modularization. Large systems are often decomposed until well defined submodules, often with a known design, are found. This increases the importance of abstraction. A well abstracted module can be reused in many different applications. The possibility of module reuse also forces the modularization to use previous described modules.

Modeling

Design is a decision making process. These decisions can used to create a model. A model is therefore often a representation of the decisions made in the design process and the decisions implications on the behavior. The most common way to use this model is in the analysis of the design and one common analysis method is simulation. This analysis may result in a redesign. Modeling and the model therefore play an important role in the design of systems.

Object-Orientation

The object-oriented methodology is based on decomposition and reuse. The problem is decomposed into objects. An object is a structuring entity containing a number of attributes. These attributes describe different properties of the object. The objects in a problem description can interact with each other.

Reuse is facilitated through inheritance in object-oriented methodology. One object can inherit attributes from another object. The first object is called subclass of the second one, which is called super class. This creates a class hierarchy, where object classes can be reused to describe more specialized objects. Object-oriented methodology is presented in [Stefik and Bobrow, 1984].

Object-oriented methods for modeling are investigated in a number of different research projects. Omola is one object-oriented modeling language that supports modularization, abstraction and reuse. It is developed at the department and an interactive environment is under implementation. Omola is used as the modeling tool throughout this thesis. Other approaches are briefly discussed in Chapter 3.

1.2 Context of the Thesis

Although structuring has been used for a long time in mechanical, chemical and electrical engineering these structuring studies, which are hardware based, have taken a long time to emerge. Because they are tied to hardware they are also less amenable to experimentation. The advances in Computer Sciences have provided a good platform to experiment with system structuring. This is the theme of this thesis.

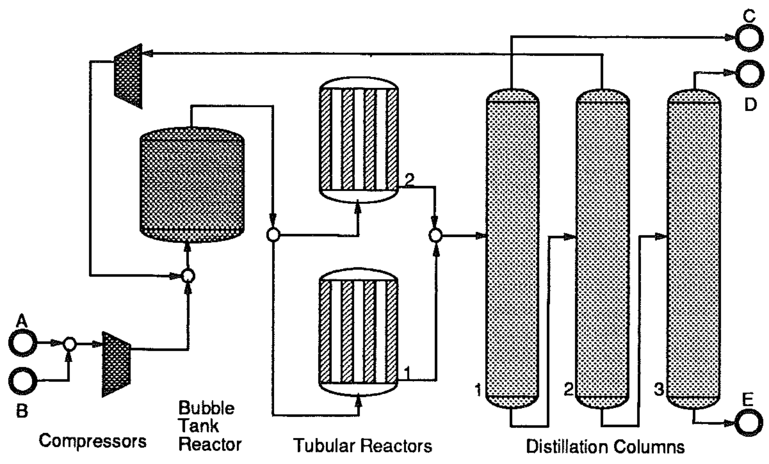


Figure 1.1 A typical flowsheet of a chemical plant.

A method of modeling and simulation called *flowsheeting* is frequently used in the design and redesign of chemical plants, see [Westerberg and Benjamin, 1985]. It usually only involves static simulation of the performance due to changes of design parameters. In flowsheeting, the process topology is described by connections between units as seen in Figure 1.1. The units are predefined and found in a library. The process medium can be described separately, independent of the units and the topology. The process description, where topology, units and medium are separated, is very user oriented. The underlying mathematical problem contains a large, nonlinear equation system which is solved through static simulation or optimization. The process dynamics and the control system design are often not considered in this phase of construction, due to the lack of tools and methodology.

The dynamic studies, together with control system construction, are done later in the design phase, sometimes even after the plant has been completed and it is too late to change the process design. The control engineering is of tradition therefore concentrated on existing plants, available process data and tools that handle local phenomena. In control system descriptions, the natural process unit representation is destroyed. The tools are often not capable of handling large systems. They are, on the other hand, "open-systems" with a model description language which allows user defined models, see [Cellier, 1991]

The flowsheeting packages use abstraction for handling large problems but this has resulted in "closed-softwares". The dynamic simulation packages have limited abstraction capabilities but they are open and support the user with model description languages.

- One of the purposes of this thesis is the unification of the tools for flowsheeting and control system construction.
- Another is to explore the use of object-oriented methodologies in the mathematical modeling of chemical processes. Although object-

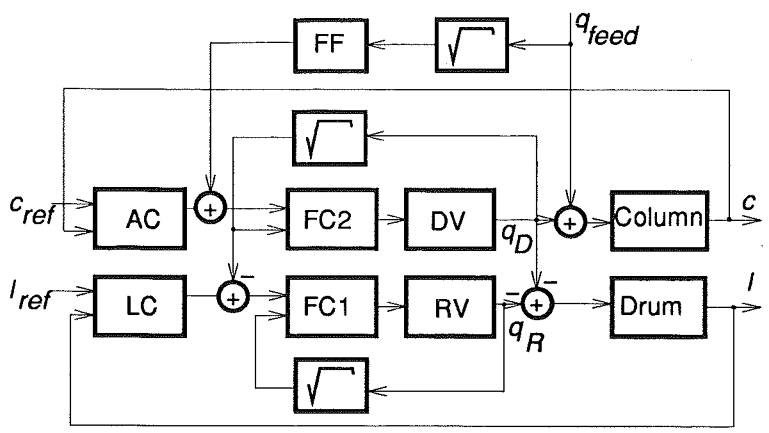


Figure 1.2 A typical block diagram of a part of a distillation column control system. For controlling of the level in the reflux drum and the composition control in the distillate product.

orientation has been used for a long time in engineering the concepts was formalized in Computer Science, where tools for dealing with it was developed.

The work has been a part of the Computer Aided Control Engineering, CACE, research program at the department of Automatic Control in Lund.

1.3 Scope of the Thesis

This thesis discusses the use of new techniques for representation, model structuring and reuse, which are based on an object-oriented approach. The object-oriented modeling language, Omola, is therefore used as a tool to demonstrate the techniques. An overview of the Omola project is found in [Mattsson *et al.*, 1993]. Primarily the models are used for dynamic simulation studies, but the aim is to develop general techniques which can also be used for other problems. The main contributions in the thesis are the use of the following concepts in dynamic flowsheeting and control system simulation:

- A methodology for system decomposition through use of submodels with well defined interfaces. A model is decomposed into a number of submodels, e. g., a chemical plant is decomposed into processing units, units into components and so on. Guidelines for structure decomposition are presented in Chapter 4 and in Chapter 7. Primitive behavior decomposition in general, and medium and machine decomposition in particular, is discussed in Chapter 5. Models with multiple interior descriptions are discussed in Chapter 8. The proposed batch process modeling concept is of particular importance.
- A methodology for building model libraries of components that can be used. Inheritance is used to facilitate reuse. A model can be reused

by the inheritance of a model from a library. A special kind of reuse allow the user to modify a library model. This flexible way to use models allows the reuse of empty models, half developed models and parts of models. Guidelines for model class inheritance are presented in Chapter 4 and Chapter 7.

- Advanced parameterization methods are used to increase the abstraction of complex submodels. Parameterization of a model means that a property of the model can be changed in order to adapt the model to different applications. A parameter can therefore be many different things, from a single variable to a whole structure of submodels. Parameterization is the subject in Chapter 6 and the discussion result in the demand of Omola extensions.
- The methodologies are applied to a nontrivial example which is successfully modeled and simulated using the concepts listed above. The example is a small but complete process plant application and it is presented in Chapter 9.

The work presented in this thesis is a mixture of demands from the user and an exploration of the new object-oriented methodology in chemical process modeling.

1.4 Thesis Outline

The thesis is divided into ten chapters. Some background and fundamental aspects of modeling and simulation are presented in Chapter 2. Chapter 3 introduces the notation of object-oriented modeling and the language Omola which is used as a tool throughout this thesis. General model structuring concepts are also discussed in that chapter. which also surveys related works Decomposition of process topologies and inheritance of process models are discussed in Chapter 4. This is continued in Chapter 5 where unit and medium separation and behavior decomposition are discussed in detail. Chapter 6 focuses on parameterization which is important for abstraction and reuse of models. Control system representation and abstraction are discussed in Chapter 7. Multi-facet models and the model database are discussed in Chapter 8. A chemical plant example, modeled and simulated in Omola, is presented and discussed in Chapter 9. Chapter 10 gives concluding remarks.

2 Modeling and Simulation

Models can be of many different kinds, from rigorous mathematical models to geographical descriptions and logical representations. A mathematical model can describe the static relations between design parameters or the dynamic behavior of the system. The mathematical model contains constraints from the physical world and from the engineering design. A pure model of a system is often mathematically incomplete, e. g., the numbers of variables and parameters are larger than the number of equations. The problem formulation describes the use of the model and makes the description complete by adding data. The required accuracy and complexity of the model are problem dependent. The problem solving method, the model and the problem formulation generate a solution with a given accuracy. The engineer uses different models, formulations and solvers for different purposes depending on the demands on the solution.

Simulation is one problem type which is important in engineering. The mathematical problem is to solve an equation system, a system of algebraic and/or differential equations. The designer uses simulation to learn how a complex process behaves and to get insight in process constraints. Simulation is also used in operation for operator training, prediction of control actions and safety studies.

Optimization is another problem formulation which is important, particularly in design. The problem is to optimize a criterion under the constraints given by the model. The model is used together with a loss function description which is optimized by the problem solving tool. To do this optimization the tool may invoke simulation.

Models are the key to both design and simulation. To create good models rapidly is an important part of engineering. In this chapter the roles of models are discussed. In the following sections the use of simulation in process and control engineering are also discussed further.

2.1 Models

Every engineering activity has its own model descriptions and its own usage of models. The separation of model, problem formulation, and solution procedure, i. e., problem solver, is more or less explicit. This is discussed in this section.

What is a Model?

A model is a chunk of knowledge that describes a system. A model is a simplified description of the system behavior from a certain point of view. It can be the dynamic time response in a specific time scale or the steady state relation between process variations and parameters. When a model is developed, it is done in a context. The model describes the system for a particular reason and with particular inputs or as Marvin Minsky [Minsky, 1965] puts it:

A model (M) for a system (S) and an experiment (E) is anything to which E can be applied in order to answer questions about S.

To violate this is both the potential and the danger in the use of models. To assume that the model M is valid for a class of experiments, (CE), makes it possible to study other experiments than the one that was used to develop the model. This can of course lead to misuse where models are used in experiments where they do not fit.

A model can be quantitative or qualitative. It can describe the behavior or the function of the system during normal operation or under fault conditions. In control engineering, quantitative description of the dynamic properties of a system is the most widely used explicit model description. This is often expressed as ordinary differential equations, ODEs. Static descriptions, often algebraic equations, AEs, is the dominating form of models in process engineering. There is clearly a good incentive to unify these view points.

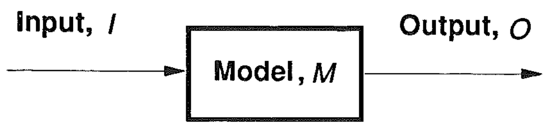


Figure 2.1 A model can be used in a number of different problem formulations.

Use of Models

Models are widely used in engineering for explicit knowledge representation in problem solving activity. Assume a model M with an input I and an output O, see Figure 2.1. Depending on what is known or specified, other missing quantities can be found.

- The so called *direct problem* is to apply I on M and study O. This is done in simulation and analysis.
- The reverse use of M is to apply O and study the input I. It can be called the *inverse problem* and it is also sometimes called the control problem.

If both the input, I, and the output, O, are specified then they can be used to find properties in M. This is used in the identification and the estimation problems in control engineering and in the design problem in process engineering.

- In the *identification problem* it is attempted to find the structure and parameters of M.
- In the *estimation problem* one can find the internal states in M if the structure of M is known.
- In process engineering this is the *design problem* where the input and the output are used to find given parameters in M, often through optimization.

A similar discussion is found in [Cellier, 1991].

Models can be used for many different purposes. In this perspective it seems natural to have problem independent models stored in libraries and use them together with different problem solving tools. Today this is not the case. Each problem solving tool has its own model description and the models are very different for different problems.

Simulation problem formulations use models in a direct and explicit way. Developing problem independent models is therefore often misunderstood as making simulation studies.

EXAMPLE 2.1—A Tank Reactor Model
A simple tank reactor is modeled by a mole balance of one of its chemical components. The reactor is assumed to have constant volume, flow, pressure and temperature. It can then be described by the following equivalence statement.

$$\begin{aligned} \text{In} + \text{Production} &= \text{Out} + \text{Accumulation} \\ qc_{in} + (-kc)V &= qc \quad + \frac{d(Vc)}{dt} \end{aligned} \tag{2.1}$$

This is a general model that describes the constraints between a number of process variables and parameters. □

EXAMPLE 2.2—Simulation of the tank reactor dynamics
To simulate the concentration dynamics in Example 2.1 one needs to define all the parameters and variables that are constant during a dynamic simulation. Examples are the process parameters, flow q, volume V and the reaction coefficient k. To make a dynamic simulation the initial value, $c(0)$, must also be given together with the input signal $c_{in}(t)$. The simulation problem is then well defined. We have the following differential equation,

$$\frac{dc(t)}{dt} = -\left(\frac{q}{V} + k\right) c(t) + \frac{q}{V} c_{in}(t), \tag{2.2}$$

which can be integrated by an ordinary ODE solver. □

EXAMPLE 2.3—The inverse problem
In the inverse problem to Example 2.1, the concentration profile, $c(t)$ is defined and the input signal $c_{in}(t)$ should be determined. The concentration profile must be differentiated in order to find $\frac{dc}{dt}$ which is used to calculate c_{in} in the following equation.

$$c_{in}(t) = \left(1 + k\frac{V}{q}\right) c(t) + \frac{V}{q}\frac{dc(t)}{dt} \tag{2.3}$$

□

EXAMPLE 2.4—Static simulation of the tank reactor
The steady-state simulation problem defines the concentration derivative to zero in Example 2.2, $\dot{c} = 0$. The system then becomes

$$c = \frac{q}{(q + kV)} c_{in} \tag{2.4}$$

and the problem can now be handled by an equation solver. □

EXAMPLE 2.5—Reactor model in a design problem
In process design one often wants to find a process parameter, for example the volume, based on a specification on the process performance. In this case the incoming and outcoming concentrations can be defined and the volume is calculated by an equation solver.

$$V = \frac{q}{k}\left(\frac{c_{in}}{c} - 1\right) \tag{2.5}$$

This problem is mathematically identical to the steady-state simulation problem above, in Example 2.4. The design problem has only one unknown, the volume. This is often not the case. A number of variables are unknown and the problem is underdetermined. Optimization is therefore used to solve the problem. To do this additional cost functions must be defined. One example is to optimize the flow and the volume based on the constraint above in Example 2.1. This means that the problem has two variables and one equation. A loss function, $F_{loss} = f(q, V)$, must then be added. The problem is now to minimize F_{loss} based on the model (constraint) 2.1. □

In Examples 2.2 to 2.5, the model of the process, found in Example 2.1, is general and it is used in a number of different problem formulations. In each problem additional information are defined, which specialize the model to a specific problem formulation.

2.2 Modeling

Modeling is an activity that can mean different thing for different engineers. A rough classification of modeler is as follows:

- The research model developer tries to describe small scale systems with complex behavior or unknown phenomena.
- The production model developer often studies large scale systems and is mainly interested in the problem solution. The elements of the model are often predefined and well known.

In a classification like this a control engineer works often as a research developer while a process designer often works as a production developer.

Modeling in Control Engineering

Control engineering relies heavily on models, both models of the control system and the process to be controlled. A model of the process is created and in the control design a controller description is developed which fulfills some specifications. This control system is then implemented in the

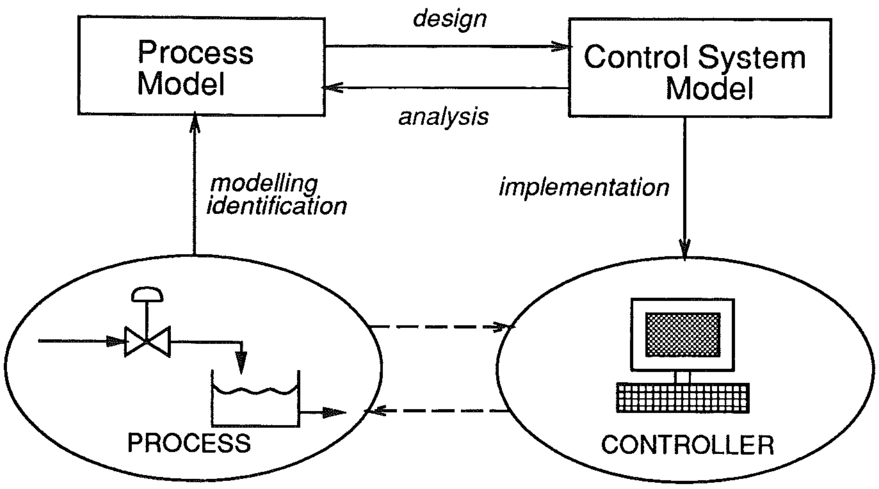


Figure 2.2 A conceptual description of activities in control engineering.

physical control system. A conceptual description of the role of process models in control engineering is illustrated in Figure 2.2.

Modeling in this context means to describe the behavior of an existing process. Since control engineering often is applied to existing processes, modeling and identification are based on some kind of experiments. The model is often input/output oriented with disturbance descriptions. Small scale models are common.

Modeling in Process Engineering

There are many different uses of models in chemical engineering. A research model developers develop new models of a particular phenomenon or models of new process components. The structure of the model is known or partly known and it is often of low order and may include complex behavior descriptions. The model developer works directly with the mathematical description to fit the model to data. The user of the developed model is often the developer himself. A control engineer and a research process model developer work like this. Problem solving software in this domain is therefore mathematically oriented. This is called microscopic modeling in Figure 2.3, which illustrate the activities in process engineering.

In chemical engineering there is also the production model developer who uses predefined component models to create models of different process designs. This kind of model user does not work on the mathematical level. Instead the developer uses a domain related description language often related to the application. An electrical engineer uses circuits and circuit elements to describe phenomena. A chemical engineer uses often common process components, like pumps, valves, reactors etc. These modeling problems are so large that there can be many model developers and many more users. This is called flowsheet modeling in Figure 2.3.

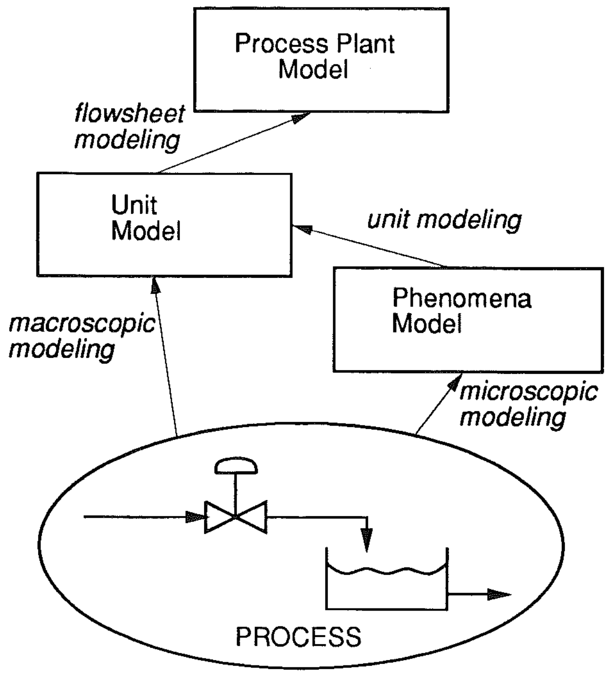


Figure 2.3 A conceptual description of activities in process engineering modeling.

The Modeling Process

Model development is more of an art than a science. How a system can be structured and mathematically described depends on the problem formulation and the problem solver. A model of a system is composed of the structure description and the behavior of the entities. Modeling is therefore equivalent to defining structural entities and their interrelations and defining behavioral descriptions of the entities.

The structure of the modeling process is illustrated in Figure 2.4. To develop a model the user has a purpose and demands on the solution. The laws of physics are also constraints on the modeling process. But the problem complexity may force the model developer to make approximations and violate physical principles. Purpose, demand and physics are the inputs to the modeling process. The modeling process can be decomposed into six levels. First the model is decomposed into minor submodels with well defined interactions, called the structuring phase. When the model is structured the submodels are described. These submodels may be recognized as standard components which can be reused; this is called the model reuse phase. If the submodel is not known the model developer has to develop a new submodel. This is done in the new description phase. Both in this phase and in the reuse phase the developer can realize that the structure must be changed which means that the process is iterative. A problem independent model is described after these three phases.

When the model is developed it can be used for problem solving. First the problem dependent information must be added which is done in the

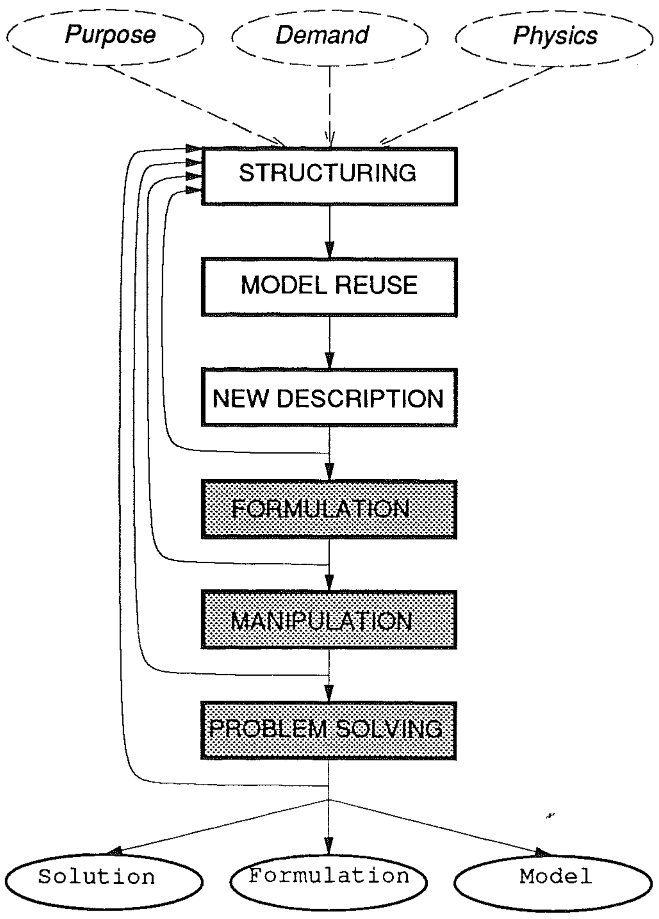


Figure 2.4 A conceptual description of the modeling process. Purpose, demand and physics are the input to the iterative process of modeling and out come the problem solution, the problem formulation and the model.

problem formulation phase. This problem formulation must often be manipulated to fit the problem solving tool. The manipulation can be a minor change of the model, e. g., sorting of explicit equations, or a major change, e. g., solving implicit equations and adding problem solving equations. The aim is automatic transformation of the original problem formulation into a problem solving tool description. The resulting description can now be used in a problem solving tool. New insight to the problem can force a new iteration in the modeling process, both in the model manipulation and problem solving phases. The output from the modeling process is not only the solution of the problem generated by the solving tool but also the problem formulation and the actual model. This is clearly illustrated in Figure 2.4.

2.3 Simulation in Control Engineering

Simulation is an important tool in control engineering. It is used for analysis of process dynamics and control system behavior. Control systems are often designed using linear models which are simplifications of the physical world. Simulation is therefore used to study the control system behavior on a more general model, e. g., a nonlinear model with different kinds of disturbances.

Continuous System Simulation Language

Most of the general-purpose continuous time system simulation languages which are commercially available are derivatives of the Continuous System Simulation Language, CSSL, that was specified by the CSSL committee in 1967 [Augustin *et al.*, 1967]. CSSL packages are all based on the same basic system representation, namely *state-space descriptions* of system equations. This is a set of first-order ordinary differential equations (ODE),

$$\begin{aligned} \frac{dx(t)}{dt} &= f(x,t,u) \\ y &= g(x,t,u) \end{aligned} \tag{2.6}$$

where t is time, x is internal state, u is input and y is output. In CSSL this description is declarative and not executive, which means that the equations are sorted into an executable sequence and solved simultaneously by an ODE solver. Surveys on general-purpose simulation softwares are given in [Kreutzer, 1986] and in [Kheir, 1988]. Use of CSSL tools is discussed and well exemplified in [Cellier, 1991]. One example of a widely used CSSL tool is ACSL [Mitchell and Gauthier, 1986].

Other commercial systems are Simnon [Elmqvist *et al.*, 1990] and SIMULINK [MathWorks, 1991]. These systems can handle composite systems, where a number of submodels are connected to each other. Simnon has a system description language for definition of systems. SIMULINK has a nice graphical interface with a block diagram editor and graphical interaction.

The standard formulation of the problem makes it possible to make a separation of the system model and the problem solver. The key idea is to allow the user to specify the problem and leave the problem solving to the simulator.

Equation Oriented Simulation

The state-space description has some major drawbacks. It can be difficult to explicitly describe derivatives in a physical based model. So called algebraic loops are another problem. This means that even if each submodel is defined by explicit calculations it may be necessary to solve an algebraic equation to find the variables of the interconnected system. Therefore some efforts have been done on leaving the explicit descriptions to allow equation oriented model descriptions such as,

$$F(\dot{x}, x, t, u) = 0 \tag{2.7}$$

This is an implicit ODE description. The mathematical formulation can be rewritten as $B(x)\dot{x} = f(x, t, u)$. If $B(x)$ is invertible then it can be turned into an explicit ODE, $\dot{x} = B(x)^{-1}f(x, t, u) = h(x, t, u)$. If it is not invertible then it is a differential-algebraic equation system, a DAE system. DAEs are categorized into so called index problems. An index zero problem can be turned into an explicit ODE description, which means that $B(x)$ is invertible. An index one problem means that the DAE is composed of one differential part, dynamics, and one algebraic part, statics. An index one problem can be solved by a DAE solver. The most well-known DAE solver is the DASSL by Petzold [Petzold, 1982]. For higher index problems some equations must be differentiated a number of times which lowers the index. The inverse problem of the tank reactor in Example 2.3 is an index two problem. The original problem formulation must be differentiated once in order to make it an index one problem that can be solved by a static equation solver.

One example of a language that handles equations is Dymola, Dynamic Modeling Language, developed in [Elmqvist, 1978]. Dymola sorts all equations and generates an explicit ODE description. It works like a preprocessor in front of an ordinary CSSL, like ACSL or Simnon. This makes it possible to handle the invertible case, index zero problem.

Discussion

Control engineering is often focused on local, small scale control problems with complex behavior. Modeling languages are often good at allowing new model descriptions but they have poor structuring facilities. The CSSL standard has established the basic idea of separating problem formulation and problem solving by the use of a manipulation phase. This is not only true in simulation but also in other parts of computer aided control engineering. It is therefore quite odd that equation oriented representations are not so common. There are a number of reasons like the lack of DAE theory and computer power, but the main reason is probably that control engineers deal with small scale problems.

2.4 Simulation in Process Engineering

Simulation has been used for more then 30 years in Chemical Engineering. Steady state simulation is the most common application. There are many packages for steady state simulation, which is also called flowsheeting. Dynamic simulation was rare and almost no commercial dynamic flowsheeting packages were available until the mid-eighties. Already in the beginning of the sixties two different paradigms in chemical process flowsheeting were developed, *sequential modular* and *equation oriented.* A lot of surveys can be found in the literature, [Westerberg and Benjamin, 1985], [Perkins, 1986], [Biegler, 1989] and [Marquardt, 1991].

Sequential Modular Flowsheeting

FORTRAN subroutines were used to make steady state calculations of unit operations in the end of the fifties. This approach was very successful and a number of so called *flowsheeting packages* that are used today rely on this approach. The name derives from the term flowsheet, which is a graphical abstract representation of the topology of chemical processes. Flowsheeting packages are unit oriented, which means that the software modules describe the behavior of a unit operation, such as a reactor, a distillation column etc. The process stream represents interactions between the unit modules which are connected to each other in a flowsheet description. The sequence of units in the flowsheet describe the process. There are packages available which were mostly developed during the seventies, like ASPEN PLUS [Aspen, 1982]. Actually, most of the major chemical companies have their own flowsheeting package. The major drawback of the sequential modular description is that the models and problem solver are tightly coupled in the software modules, i. e., FORTRAN subroutines. It is only in recent years that user friendly interfaces have become available for user defined module descriptions. Another drawback is the problem with recycle loops in the flowsheet. The modules are calculated in a sequence and when the recycle appear, an iteration of the sequence must be done which must be repeated until it converges. In flowsheeting the process model description is separated into three parts: the process topology, the process units and the medium property. A small application in ASPEN PLUS is found in Appendix A.

Equation Oriented Flowsheeting

The problem discussed above with recycle loops was recognized already in the beginning of the sixties, [Sargent and Westerberg, 1964]. An alternative approach was developed, equation oriented flowsheeting. In this approach the user defines the equations, which are solved once and for all. The advantage of equation oriented flowsheeting is a well defined separation of model and problem solving. Recycle loops do not create any problems in equation oriented flowsheeting. The main drawback is that many models are not described by simple equations. Typical examples are physical properties. These are instead described by implicit equations which require iteration. Tailormade solvers have been developed for the sequential modular flowsheeting packages.

Dynamic flowsheeting is often based on the equation oriented approach. SPEEDUP, developed at Imperial College in London, one of the first equation oriented flowsheeting packages [Sargent and Westerberg, 1964], has an equation oriented but FORTRAN like syntax for system representation, see [Perkins and Sargent, 1982]. Today, SPEEDUP has a DAE solver for index one problems, see [Pantelides, 1988]. SPEEDUP has much in common with Dymola.

In recent years some efforts have been made to combine the two approaches into what is called the *simultaneous modular* method. The equation oriented approach is used to describe the structure and the sequential modular approach is used to describe the physics, see [Fagley and Carnahan, 1990].

Discussion

The two paradigms, sequential modular and equation oriented, share some common ideas, which are based on the user demands on flowsheeting and the user needs to evaluate and analyze a process design. The process design is based on structures of unit operations where the problem is to size the units. This means that the user wants to work with unit operation modules and change well defined parameters in these modules. The unit operation modules must also be medium independent and the flowsheeting package must support a library with physical property descriptions for common chemicals. This results in unit orientation of the flowsheet structure and a separation of units and physical properties. Good structuring facilities have been developed in flowsheeting. They are problem specific, and use large data bases for units and medium descriptions. It is on the other hand difficult to include new submodel descriptions.

2.5 Conclusions

Modeling and simulation have been discussed in this chapter. These two activities are sometimes interwined because simulation problems require an explicit model description. Models are used in all engineering activities and tools for developing general models are therefore important. The modeling process was discussed together with the practice of simulation in control engineering and in process engineering. This showed that the different areas have much in common. An unification should allow the power of an open modeling language together with concepts for abstraction and reuse developed in flowsheeting.

3

Object-Oriented Modeling

Object-oriented methodology has been a subject of research in computer aided engineering in the last years. Object-oriented programming was developed in the Computer Science already in the end of the sixties and got a lot of attention in the eighties. The basic idea is to group properties of an entity into an object. An important concept is inheritance. One object class is a subclass of another class and inherit its properties. A number of different research groups are working on applying object orientation on modeling.

Omola is an object-oriented modeling language. It is a modeling language that captures the modeling structuring concepts developed in the Computer-Aided Control Engineering research program, CACE, at the department of Automatic Control at Lund Institute of Technology, [mat, 1989]. Omola is the third generation of modeling software developed at the department. Simnon is CSSL inspired with state space model description language. It was developed in the early seventies, see [Elmqvist, 1975]. Simnon is commercially available and is still widely used, see [Elmqvist *et al.*, 1990]. Dymola, a dynamic modeling language for large continuous systems, was presented in Elmqvist's thesis in the late seventies, [Elmqvist, 1978]. It is based on structured modeling and has powerful structuring concepts. It works as a preprocessor where Dymola models are

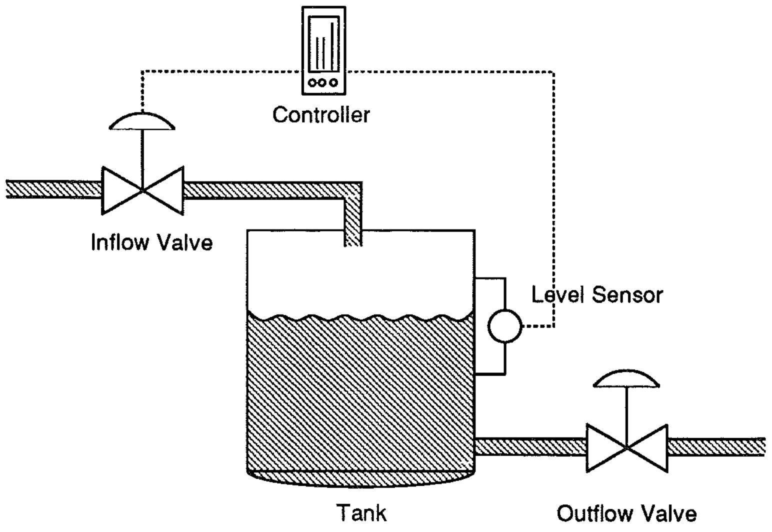


Figure 3.1 The introductory tank system example.

manipulated and fed to other packages for simulation. A renewed interest in Dymola is reported in [Cellier and Elmqvist, 1993] and [Elmqvist, 1993] and today Dymola is a commercially available. The object-oriented modeling language, Omola, is presented in [Andersson, 1989] and in Andersson's thesis [Andersson, 1990]. A nice overview of Omola is found in [Mattsson and Andersson, 1992] and a more detailed discussion is found in [Mattsson *et al.*, 1993].

In this chapter we first introduce Omola by an introductory example in Section 3.1. A discussion of the modeling concepts in Omola is presented in Section 3.2 and a more detailed presentation of the Omola modeling language is then given in Section 3.3. The concepts are based on ideas from structured and object-oriented programming. The Omola Simulation Environment, OmSim, is briefly presented together with a discussion about engineering problem solving environments of tomorrow in Section 3.4. Related projects in object oriented modeling are briefly discussed in the last section.

3.1 The Tank Example

A small example which illustrates some of the modeling concepts in Omola is now given. The example is a water tank with one inlet and one outlet. The level of the tank is controlled by the inflow.

The tank system is composed of a water tank with an inflow valve, an outflow valve and a controller, see Figure 3.1. The level of the water tank is controlled by the controller which uses the inflow valve as an actuator. The outflow is changed by an external consumer.

To study the dynamic behavior of this system we need a dynamic model that describes the mass balance of the tank. It is also assumed that

```
TankSystem ISA Model WITH
terminals:
  In  ISA PipeInTerminal;
  Out ISA PipeOutTerminal;
submodels:
  Tank         ISA TankModel;
  InflowValve  ISA ContValveModel;
  OutflowValve ISA ValveModel;
  PID          ISA PIDControllerModel;
connections:
  In              AT InflowValve.In;
  InflowValve.Out AT Tank.In;
  Tank.Out        AT OutflowValve.In;
  OutflowValve    AT Out;
  Tank.Level      AT PID.Measure;
  PID.Control     AT InflowValve.Control;
END;
```

Listing 3.1 The tank system model expressed as Omola code which is a textual description of the graphical interpretation in Figure 3.1.

the valves can be modeled statically both with respect to mass and momentum balances. The flow through a valve is assumed to be proportional to the velocity which in its turn is a nonlinear function of the pressure drop. The controller is an ordinary PID-controller.

A top-down description begins with the composite model describing the tank system with its process components. This is given in Listing 3.1. The tank system class has twelve attributes: the in and out terminals, four submodels and six connections. `TankSystem` uses four reserved word: `ISA`, `WITH...END` and `AT`. `ISA` describes the classification of the model or the model component and `WITH...END` indicate the beginning and the end of the model body. The `AT` operator is interpreted as a connection between two terminal. All attributes are grouped together under a so called category tag, such as `terminals` and `submodels`. These tags are optional and are used more like comments than as a part of the language. The actual mathematical description is found further down in the structure hierarchy. The four submodels are process components and use the description of predefined model types or model classes, like `TankModel`.

The interaction between the composite model components, the submodels, are described by connections. A connection is a relation between two terminals with identical internal structure. The water flow from one component to another is described by the `PipeInTerminal` given in Listing 3.2. The terminal is a record terminal composed of two subterminals, `Flow` and `Pressure`. These terminals represent the interaction variables.

Listing 3.3 shows the Omola model for the water tank which is a so

```
PipeInTerminal ISA RecordTerminal WITH
  Flow ISA ZeroSumTerminal WITH
    direction := 'in;
  END;
  Pressure ISA SimpleTerminal;
END;
```

Listing 3.2 Omola description of the terminal describing flow interaction between process components in the simple water tank example.

called primitive model. The `TankModel` has eight attributes. Three terminals describe inflow and outflow and one measurement. The tank also has four parameter attributes which are time invariant variables. One internal variable, `mass`, is also defined in the tank model. The behavior is described by four equations: one mass balance, one constant density assumption, and one mechanical energy balance, a Bernoulli equation. The tank is also assumed to have atmospheric pressure. The dot notation is used to refer to an attribute of a class. In the mass balance equation the flow descriptions are subterminals of record terminals, `In` and `Out`. Note the derivative operator, `mass'`, in the mass balance which is the first equation.

Omola can describe structure hierarchies and hierarchical terminals as shown above. Omola also has taxonomy concepts from object-oriented programming, like classes and inheritance. The use of inheritance is illus-

```
TankModel ISA Model WITH
terminals:
  In    ISA PipeInTerminal;
  Out   ISA PipeOutTerminal;
  Level ISA SimpleTerminal;
parameters:
  Density   ISA Parameter;
  GravAcc   ISA Parameter;
  TankArea  ISA Parameter;
  PipeArea  ISA Parameter;
variable:
  mass ISA Variable;
equations:
  mass' = Density*(In.Flow - Out.Flow);
  mass = Density*TankArea*Level;
  Density*GravAcc*Level =
    Out.Pressure + Density*(Out.Flow/PipeArea)^2/2;
  In.Pressure = 0.0;
END;
```

Listing 3.3 Omola models of the tank class which is used as a component in the tank system model.

```
ValveModel ISA Model WITH          ContValveModel ISA ValveModel WITH
terminals:                         terminal:
  In  ISA PipeInTerminal;            Control ISA SimpleTerminal;
  Out ISA PipeOutTerminal;         parameter:
parameters:                          ValvePar ISA Parameter;
  PressDrop ISA Parameter:         variables:
  Density   ISA Parameter:           PressDrop,Position ISA Variable;
  PipeArea  ISA Parameter:         equations:
equations:                           PressDrop =
  In.Flow = Out.Flow;                  ValvePar*Position;
  PressDrop *                        Position =
    (In.Pressure-Out.Pressure)=        IF Control<0 THEN 0 ELSE
    Density *                          IF Control>1 THEN 1 ELSE
    (In.Flow/PipeArea)^2/2;            Control;
END;                               END;
```

Listing 3.4 Omola models of the valve *(left)* and its subclass control valve *(right).*

trated in the valve model classes in Listing 3.4. The valve model, ValveModel, describes a static valve with seven attributes for the inflow and outflow terminals, three valve parameters and two equation attributes. The equations describe the static mass balance and a Bernoulli equation. The PressDrop parameter describes how much of the pressure difference over the valve is used to generate the flow through the valve.

The control valve model, ContValveModel, is a specialization of the valve model. The valve model attributes are inherited and the inherited PressDrop parameter is overwritten and replaced by a variable which changes with the valve position. The valve position is proportional to the control signal and its range is limited to be between 0 and 1.

Description Structure

Two different hierarchies are discussed above, the structure hierarchy for abstraction and the class hierarchy for inheritance and reuse. These hierarchies are also called *HAS-A* and *IS-A* hierarchies. These two hierarchies are orthogonal to each other and create a net structure. A small part of the network of objects for the tank system example is shown in Figure 3.2. Every item is an object class. The super classes for model, record terminal, zero sum terminal and simple terminal are all predefined in Omola, see Figure 3.2. The record terminal class has one subclass which also has one subclass. The PipeInTerminal has two subterminals which are local subclasses of predefined Omola classes, ZeroSumTerminal and SimpleTerminal. These two terminal subclasses PipeInTerminal and PipeOutTerminal are globally defined in the model database. The model

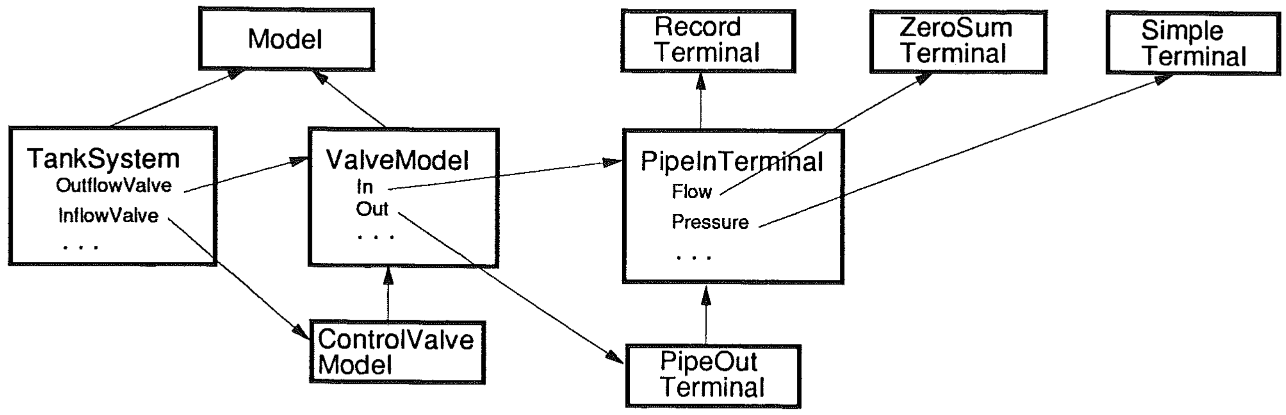


Figure 3.2 A part of model and class hierarchies in the tank example.

root class has two subclasses, namely `TankSystem` and `ValveModel`. The valve model has two local attributes that are subclasses to the globally defined terminal classes. These local terminal classes are inherited by the control valve model. The tank system class has two locally defined valve classes which inherit attributes from their super classes. These local classes are only accessible inside the owner object and its subclasses. Notice that local attribute classes can be specialized.

3.2 Modeling Concepts in Omola

Omola is based on the methodology from structured and object-oriented programming. A brief discussion is presented in this section.

Structured Modeling in Omola

The concepts for structuring models have strong relations to structured programming. Creation of a hierarchy of encapsulated models with well defined interfaces have corresponding constructs in programming. The model must have an interface for communication with the surrounding models. This makes it possible to abstract the internal structure of the model. Local complexity inside a model is hidden in the model interior. This makes it easy to handle a structure of abstracted models. To summarize the abstraction concepts a model is composed of two parts, interface and interior.

- The *interface* of a model describes its interaction. It is composed of two types of model components.
  - A *terminal* is a model component class which describes interaction with a connected model.
  - A *parameter* is a model component class that allows the user to interact with the model. It is used to adapt the behavior of the model to a new application.

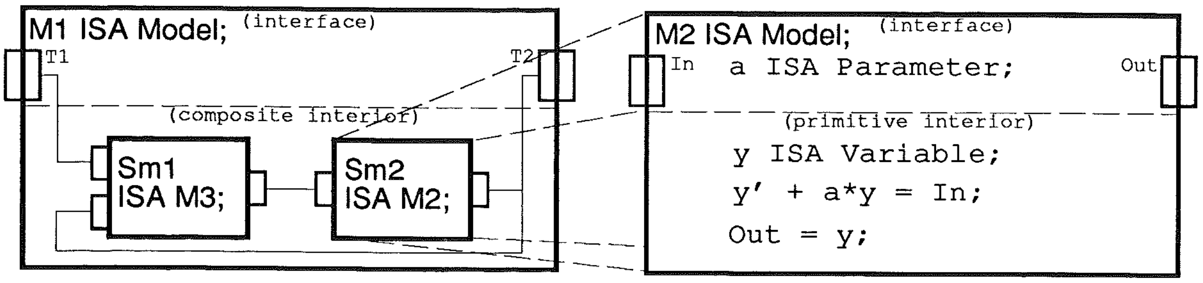


Figure 3.3 The concept of structured modeling pictured graphically. Models have well defined interfaces and interior descriptions.

- The *interior* is a description of the model's internal behavior. It can be primitive or composite. A primitive interior is described symbolically with equations and variables. A composite interior is described as a structure of connected submodels.
  - An *equation* describes the relation between variables, simple terminals, their derivatives and parameters.
  - A *variable* describes an internal state that is not seen in the model interface.
  - ○ A *submodel* is a structuring entity describing a part of the behavior of a composite model.
  - ○ A *connection* is a relation between model terminals.

EXAMPLE 3.1—Encapsulation
The abstraction concepts are illustrated in Figure 3.3. Model M1 has an interface consisting of two terminals. It also has a composite interior of two submodels, Sm1 and Sm2. The Sm2 submodel is a local subclass of the global model class M2, which has an interface composed of two terminals and one parameter. It has a primitive interior described by one internal variable and two equations. □

Object-Oriented Modeling in Omola

Object-oriented modeling is strongly influenced by the ideas and methodology developed in object-oriented programming. Classes and instances are used in the Omola object-oriented approach to modeling.

A *class* is a type definition which is composed of attributes. Attributes can be inherited and local. Inherited attributes are inherited from the super class. A super class can inherit attributes from its super class and so on. This creates a *class hierarchy* with *single inheritance* where a class only can have one super class. If it can have many super classes then it is called *multiple inheritance* which creates a net structure. Omola supports single inheritance. The local attributes of a class are used to make the class more specialized than its super class. The most common rule describing inheritance is that a local attribute that has the same name as an inherited attribute overwrites the inherited one. In other words the last defined attributes in the class hierarchy are the valid ones. There can be other rules, like deletion of inherited attributes and other overwriting rules in multiple inheritance, but they are not so common and are not discussed here.

Another concept is the *instance* in object-oriented methodology. An *instance* is the actual data structure described by a class and used by the executing program. Omola does not use instances on the model representation level. Instances are used in the model compilation. They are created in the process of translating a model into simulation code.

Inheritance facilitates *reuse*, *specialization* and *polymorphism*. The reuse is directly seen in the tank example in the first section in this chapter. Local classes in a composite model reuse the class definition of global classes. Specialization means that a subclass in the inheritance hierarchy has local attributes specializing the subclass making it more special for its purpose. Polymorphism is a concept closely related to reuse. Polymorphic models can be used in the same context and the surrounding models in a composite model cannot see any difference between the models. Polymorphic models can be exchanged in a composite model without any modifications which means that the structure is reusable. A number of objects with different internal descriptions can have the same super class. This super class contains the interface definitions. In object-oriented programming polymorphic objects have the same names for their *methods* and the *message passing* cannot see any difference. A message is the specification of an operation to be performed on an object. Similar to a procedure call. A method is a function that implements the response when a message is sent to an object. The Omola version of object-oriented modeling does not have *methods* and *message passing*. Instead it has equations and connections. Connections are like message passing and equations are like methods. As long as the connections between the surrounding models and the polymorphic models are valid the internal descriptions of equations are polymorphic.

EXAMPLE 3.2—Inheritance

The use of inheritance is illustrated in the Figure 3.4. M2a1 is a subclass of M2a which directly reuses its inherited description. It only has one additional parameter binding. M2a is a specialization of its super class, M2ic, which contains an interface description. Since M2b has the same interface and super class as M2a they are polymorphic. In the composite model M1 the submodel Sm2 is a subclass of M2a. A new model which uses the polymorphic model instead is easy to define. M1b is a subclass

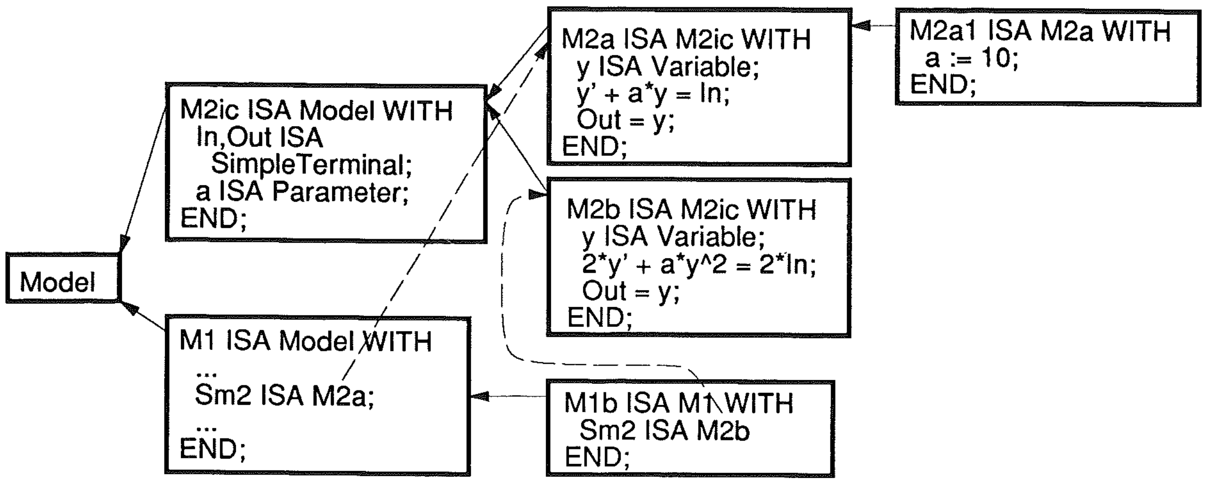


Figure 3.4 An illustration of the use of inheritance for direct reuse, specialization and polymorphic models

of M1 and it inherits its attributes except for the Sm2 definition which is overwritten. The new definition of Sm2 has a new super class, M2b, a polymorphic model to the original one. □

3.3 The Omola Modeling Language

Omola is a general object-oriented data representation language. It can be viewed at two different levels, one basic level and one model representation level. Omola is presented in Andersson's thesis, [Andersson, 1990]. A nice description of Omola is also found in [Mattsson and Andersson, 1992]. Extensions to combined discrete events and continuous time systems are published in [Andersson, 1992] and [Andersson, 1993a]. A user's manual of Omola and OmSim environment is found in [Andersson, 1993b]

Basic Omola

The class is the most important entity in Omola. Every class has a name and a set of attributes. The set of attributes is defined in the class body. A class definition is seen below.

```
{name} ISA {name of super class} WITH
  {class body}
END;
```

The class body contains the attributes of the class. Attributes can be other class definitions, variable type definitions, assignments and equations. These class definitions create a class tree hierarchy where the root of all classes is called `Class`. A class inherits all attributes from its super class. If the class has a local attribute that has the same name as an inherited attribute, the local overwrites the inherited.

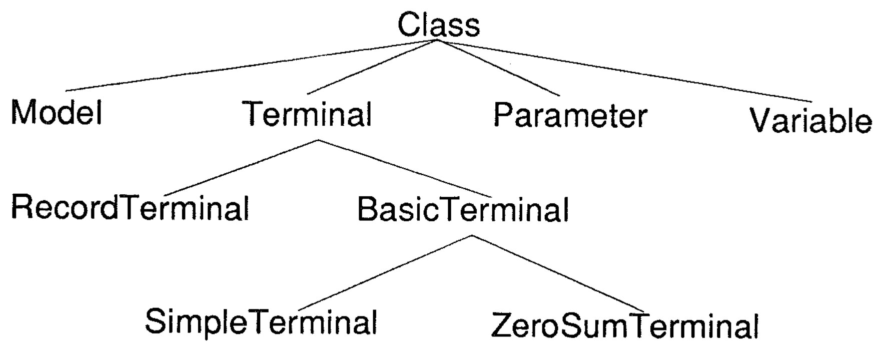


Figure 3.5 The predefined classes for model representation in Omola.

Omola Model Representation

A set of classes are predefined in Omola. They have special meaning to the system and in some cases rules limiting what kind of attributes they can have. The most important predefined Omola classes are the `Model` class and three model component classes `Terminal`, `Parameter` and `Variable`. The predefined class hierarchy is illustrated in Figure 3.5.

- The `Model` class is the root class for all user defined models. It can contain model component attributes.
- The `Terminal` class is the root class for all model interaction classes. It has subclasses for record terminals and for basic terminals. The basic terminal class has subclasses for simple terminals and for zero sum terminals. This is discussed in more detail in the next subsection.
- The `Parameter` class is a time invariant variable used by the model user to adapt the model behavior to new applications.
- The `Variable` class is the super class for user defined variables used as internal variables in models.

User defined classes are defined as subclasses of these predefined classes. New attributes are added to these user defined models. The attributes can be other defined subclasses, equations, connections and events. This is clearly illustrated in the inheritance example 3.2.

Submodel Interaction

The interaction between two submodels is described by connections between terminals. A connection is valid if the two terminals connected to each other have the same internal structure. A record terminal is used to describe a set of interactions and a connection between record terminals means that each individual subterminal pair is connected.

A *basic terminal* represents a single quantity of interaction. That quantity is represented by an attribute called value. It also has three attributes called causality, quantity and unit. These attributes must have

```
AModel ISA Model WITH
terminals:
  T1 ISA RecordTerminal WITH
    x ISA SimpleTerminal;
    y ISA SimpleTerminal  WITH causality:='input; END;
    z ISA ZeroSumTerminal WITH direction:='in;    END;
  END;
  T2 ISA this::T1 WITH
    y ISA SimpleTerminal WITH causality:='output; END;
  END;
connection:
  T1 AT T2;
END;
```
Listing 3.5 A model containing two connected record terminals

compatible values in order for a connection to be valid. The causality of a terminal can be undefined, input or output. The causality defines the calculation order of the terminal variables. A connection between terminals with undefined causality is interpreted as an equation between the terminal variables. A connection with a well defined causality is interpreted as an assignment and cannot change direction. The quantity attribute specifies the name of the physical quantity which the terminal represents and two terminals with different quantity cannot be connected. The unit attribute describes the unit of the quantity and is used for automatic scaling. This makes it possible to use models from different libraries developed in different units.

The *zero-sum-terminal* is used for describing flows and has one additional attribute describing the positive direction of the value. The direction attribute can be defined as `in` or `out` and the default direction is `in`. It is used for describing flows. Submodel interaction in Omola is discussed in [Mattsson, 1988] and also in [Mattsson, 1989].

An example containing a number of different connection interpretation is illustrated in Listing 3.5. `AModel` contains two terminal attributes and one connection. The first terminal, T1, is a record terminal with three subterminals. They are one simple terminal, one simple terminal with a given causality and one zero sum terminal with a given direction. The second record terminal, T2, is a subclass of T1 with a new definition of the y subterminal with the opposite causality. The result of the connection is interpreted as follows.

$$\begin{aligned} T1.x &= T2.x \\ T2.y &:= T1.y \\ T1.z + T2.z &= 0 \end{aligned}$$

The x terminal connection is translated into an equation. The y terminal connection is interpreted as an assignment where the output terminal assigns the value of the input terminal. The third z terminal connection is described by an equation where the values are summed to zero. The sign in front of the value is defined by the direction attribute.

Primitive descriptions in Omola

A model can have behavior described by equations. Such a model is sometimes referred to as a *primitive model* since it is not further decomposed. The equations may contain common mathematical expression. The elements of the equations are variables, basic terminals and parameters which can be of the types integer, real, boolean and matrix. Variables can also be defined directly using types. The variable can have an optional binding to an expression.

```
{name} TYPE {type} := {expression};
```

The time derivatives of variables can be expressed as x' or x''. Higher order derivatives are expressed by the DOT operator like DOT(x,3). An example of a primitive model description is the one of the tank model in the previous section, see Listing 3.3.

```
mass' = Density*(In.Flow - Out.Flow);
mass  = Density*TankArea*Level;
```

The derivative of the mass variable is equal to the density times the difference between the inflow and the outflow. The next equation describes the relation between mass, density, tank cross area and tank water level. The density and the tank area are parameters. If the inflow and outflow are known and the parameters have proper values, the mass is a state variable and the level is an output variable. The level becomes a function of the mass after the equation manipulation phase.

Assignments can be used to force a binding of an expression to a variable. A variable or parameter that is assigned a numerical value becomes a constant. When the causality of a module is given and the behavior is of state space form, assignments can be used instead of general equations.

Another model component used in primitive models is the event. Events can be used to describe discrete systems. Events can be time dependent events or state dependent events and may have explicit names. Time events are scheduled by a schedule operator and state events are caused by conditions. The use of events is discussed in more detail in Chapter 7.

Summary

The Omola language is based both on structured and object-oriented modeling. It has concepts for model encapsulation with model components describing its interface and its interior. Models and model components are classes in Omola. Even local model components that are attributes of a model class are classes, so called local classes.

3.4 Omola Simulation Environment

The Omola Simulation Environment, OmSim, is the implementation of a kernel for object-oriented modeling. This kernel can represent Omola models with a one to one mapping. In other words, Omola is the textual description of the models in the OmSim kernel.

OmSim Architecture and Implementation

The OmSim architecture is illustrated in Figure 3.6. The architecture is based on a central model database. This database can be accessed by a number of tools. The so called model database in current OmSim is not a proper database since it does not support permanent storage of its contents. The user must explicitly store the contents on files. There are tools for model development and for simulation. A browser is used to show the classes in the database and there are a number of graphical tools to display the contents of the model database. A graphical object diagram

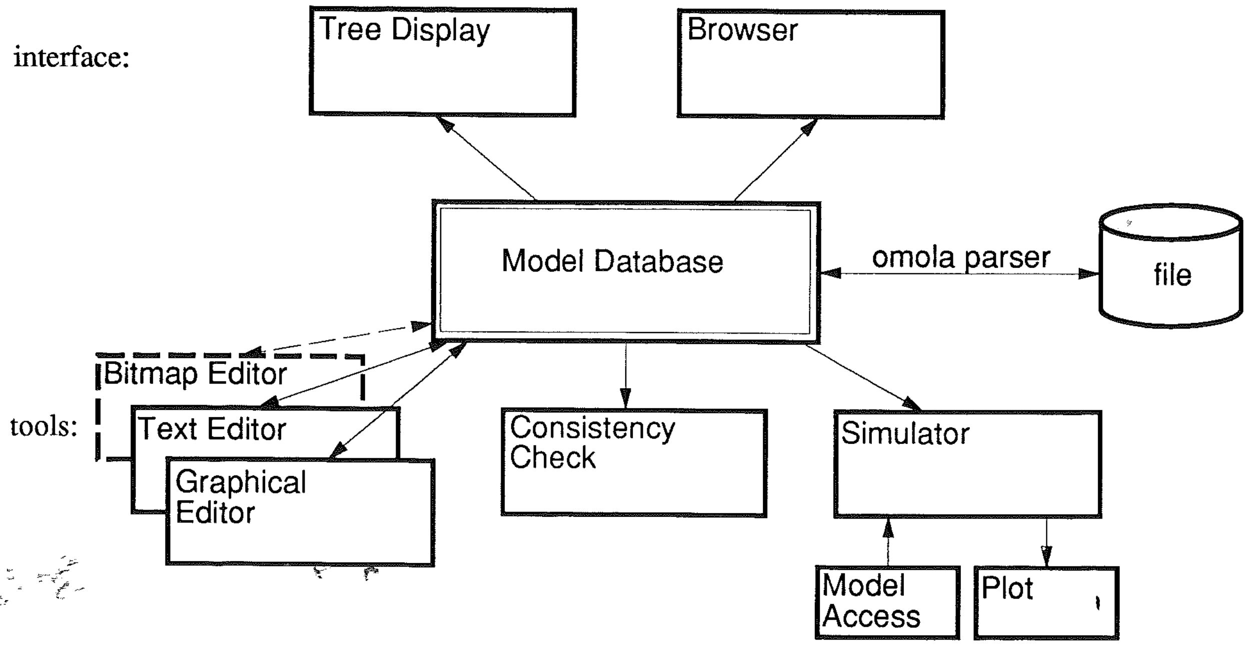


Figure 3.6 The OmSim architecture with the model database in the center surrounded by a number of different tools.

editor, called MED, is used to develop composite models and an ordinary text editor, Emacs, to develop primitive models. The OmSim simulator is only a problem solving tool in the current version. There is a supporting tool for checking of the model consistency.

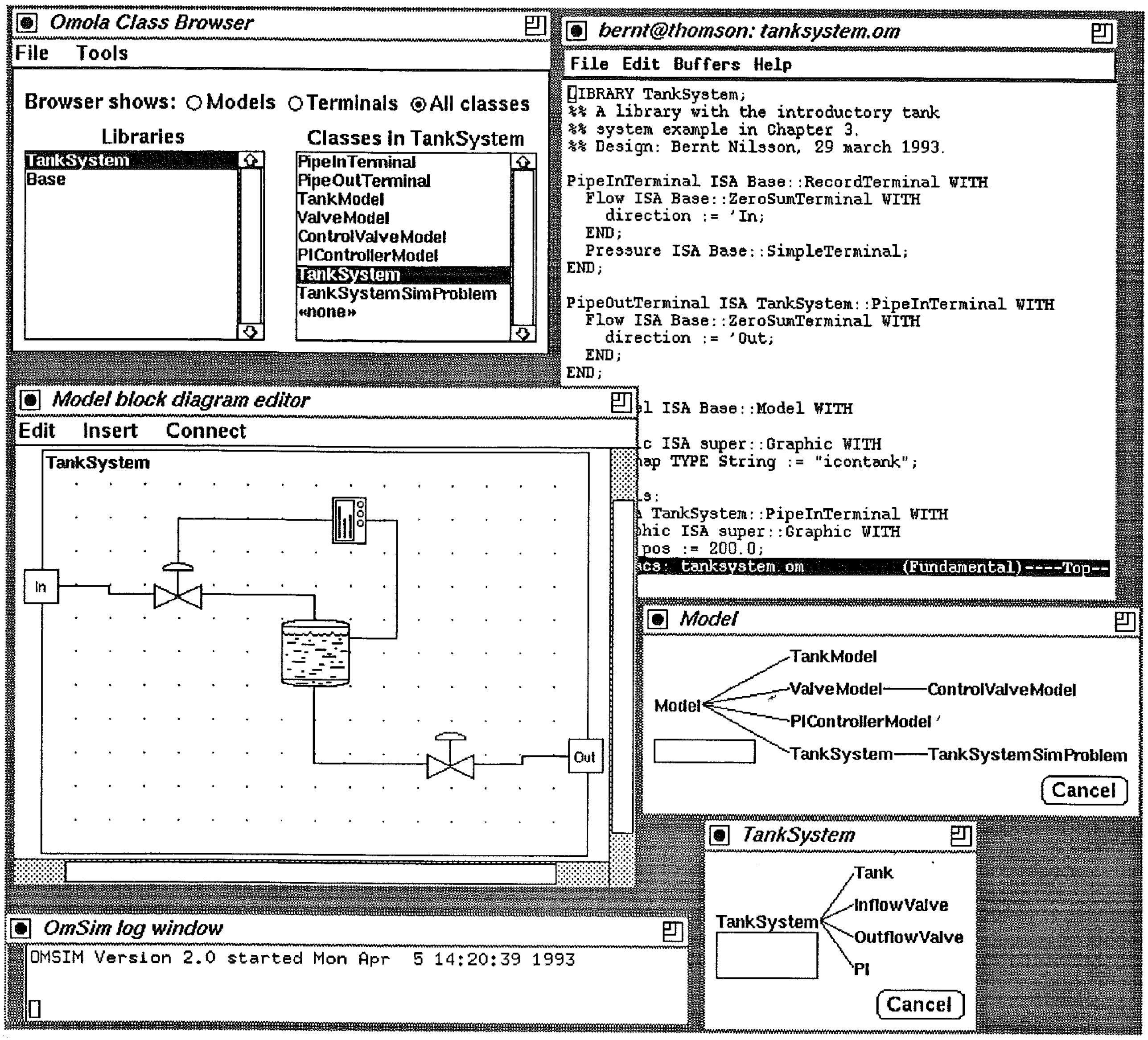
A prototype called SEE, System Engineering Environment, was developed by Andersson [Andersson, 1989]. SEE was implemented in KEE [IntelliCorp, 1987] and CommonLisp [Winston and Horn, 1984]. KEE and CommonLisp need powerful computers. If SEE was to be spread among a larger community of users, it probably would have to be implemented in a less resource demanding environment. Therefore it was decided that an implementation in a compiled language should be developed and the name was changed to OmSim. The current version of OmSim is implemented in C++, an object-oriented programming language [Stroustrup, 1986]. OmSim is developed under Unix using X-windows. Interviews, a public domain graphics library, is used for developing the graphical interface. It is important to note that Omola model classes cannot be represented as classes in the implementation language. This is because C++ classes are pure compile-time entities which do not exist at run-time. In the modeling environment, however, model classes are created and modified dynamically. A dynamic object-oriented environment is therefore developed in OmSim using C++.

Model Development in OmSim

Models can be developed outside OmSim using an ordinary text editor, for instance Emacs. These models, stored in files, are parsed into the OmSim model database. An alternative way is to develop models inside OmSim using Emacs and a graphical object diagram editor called MED. It is possible to send models from the OmSim database to Emacs, edit and send the model back using temporary files.

The graphical model editor is an object diagram editor where submodels can be defined and connections between terminals can be drawn. Submodels are represented by user defined icons or by standard rectangles A model with an undefined icon attribute automatically gets a rectangle icon. A composite model then looks like a block diagram. An icon is developed using a bitmap editor outside OmSim. The OmSim architecture is illustrated in Figure 3.6 with the three editors on the left.

An example of how OmSim might look for model development is shown in Figure 3.7. The OmSim model browser is seen at the top left. Except for the predefined library it contains only one library, the tank system library. It is composed of two terminal classes and six model classes. The composite tank system model is seen in the graphical editor below the browser. The object icons are developed and stored outside of OmSim.



```
LIBRARY TankSystem;
%% A library with the introductory tank
%% system example in Chapter 3.
%% Design: Bernt Nilsson, 29 march 1993.

PipeInTerminal ISA Base::RecordTerminal WITH
  Flow ISA Base::ZeroSumTerminal WITH
    direction := 'In;
  END;
  Pressure ISA Base::SimpleTerminal;
END;

PipeOutTerminal ISA TankSystem::PipeInTerminal WITH
  Flow ISA Base::ZeroSumTerminal WITH
    direction := 'Out;
  END;
END;

l ISA Base::Model WITH

c ISA super::Graphic WITH
ap TYPE String := "icontank";

s:
 TankSystem::PipeInTerminal WITH
hic ISA super::Graphic WITH
 pos := 200.0;
cs: tanksystem.om          (Fundamental)----Top--
```

```
OMSIM Version 2.0 started Mon Apr  5 14:20:39 1993
```

Figure 3.7 The tank system example illustrated in OmSim.

An ordinary text editor, Emacs, is used to develop the primitive models. The Emacs window on the right in Figure 3.7 is loaded with the tank system library. Two tree display windows are found below the text editor. One shows the class tree, i. e., the inheritance relations between the six model classes. The other shows the structure hierarchy of the tank system, which is composed of four submodels.

For long time storage, models are stored on files. A library concept has therefore been developed. Omola models are grouped in libraries and one practical way of storing models is one library per file.

Simulation in OmSim

The OmSim simulator can be divided into two parts, the model compilation part and the simulation part.

The procedure for transforming an Omola model into a representation that can be handled by the simulator is quite complex. The procedure, which includes analysis of consistency and completeness, is called model compilation for simulation. It is composed of two steps, one model syntactical and semantical analysis and one mathematical consistency check and transformation.

Model syntactical and semantical analysis is used to control the model description in Omola:

A Lexical and syntactical analysis. This is done already when a model is loaded into the model database.

B Semantical check involves scope rules for name and type consistency of expression.

C Connection consistency check.

Mathematical consistency checking and transformation is the second step. The mathematical properties of the model are analyzed and a proper model is transformed into simulation code:

1 Check of structural defects.

2 Order equation into a sequence of subproblems.

3 Sort out time-invariant parts.

4 Derive the differential index-one problem.

5 Sort equations in computational order.

6 Check causality.

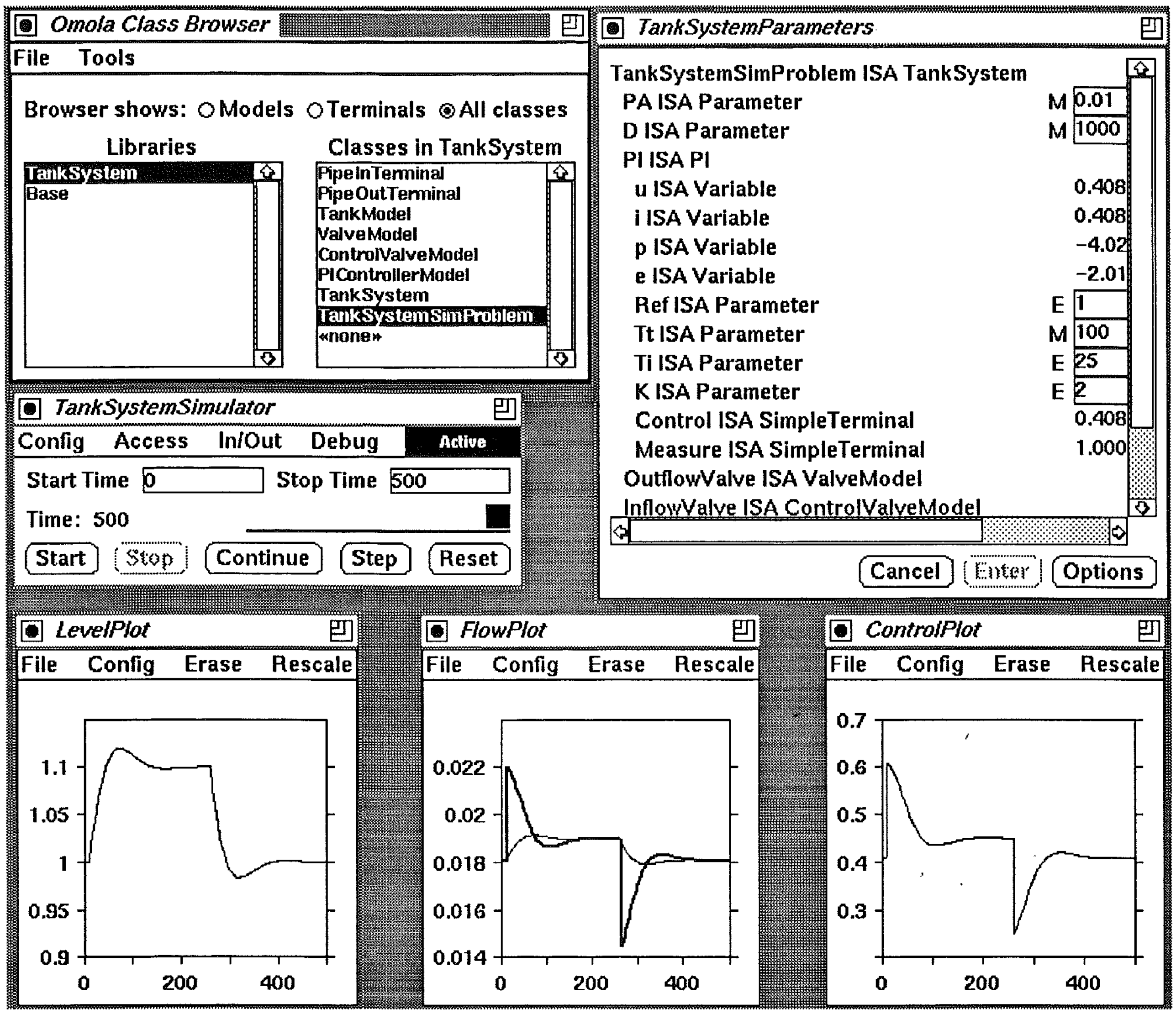
7 Reduce higher index.

8 Partitioning and symbolic manipulations.

9 Output a result suitable for simulation.

A model can be simulated after a successful model compilation. The result from the model compilation is the simulation code that can be used by the numerical solvers. The OmSim simulator has a number of different numerical solvers. For differential-algebraic problems there are DASSL and Radau5. DASSL is a multi-step method and Radau5 is an implicit Runge-Kutta method. There are also three solvers for ordinary differential equations, Dopri45r is an explicit Runge-Kutta method with modified step size control, one pure Euler method and finally RKsuite, a state-of-the-art explicit Runge-Kutta method. A more rigorous discussion about the OmSim simulator is found in [Mattsson *et al.*, 1993].

It is possible to interact with the simulator and change the numerical solver, the error tolerances on the solution etc. It is also possible to interact with the model and change parameter settings and initial conditions.



Libraries	Classes in TankSystem
TankSystem	PipeInTerminal
Base	PipeOutTerminal
	TankModel
	ValveModel
	ControlValveModel
	PIControllerModel
	TankSystem
	TankSystemSimProblem
	«none»

Start Time 0 Stop Time 500

Time: 500

TankSystemSimProblem ISA TankSystem		
PA ISA Parameter	M	0.01
D ISA Parameter	M	1000
PI ISA PI		
u ISA Variable		0.408
i ISA Variable		0.408
p ISA Variable		-4.02
e ISA Variable		-2.01
Ref ISA Parameter	E	1
Tt ISA Parameter	M	100
Ti ISA Parameter	E	25
K ISA Parameter	E	2
Control ISA SimpleTerminal		0.408
Measure ISA SimpleTerminal		1.000
OutflowValve ISA ValveModel		
InflowValve ISA ControlValveModel		

Figure 3.8 The tank system example simulated in OmSim.

The result of a simulation can be displayed in plot windows or stored in files. The user can run the simulation interactively, manipulating different windows with buttons and menus. The Omola command language, OCL, is a language that can describe a sequence of simulator commands.

A simulation of the tank system example is seen in Figure 3.8. The model database browser is found top left and below this the main simulator panel is found. On the right, a model access window shows the parameters and variables of the tank system model. The presentation is automatically expandable and by clicking on an object the internal variables are listed. Parameters are interactively changeable in this window. Three plot windows, in the bottom of Figure 3.8, show the level, inflow and outflow and finally the control signal. The simulation example shows what happens when the reference value is changed from 1.0 to 1.1 at time 10 and back again at time 250. The tank system simulation problem is a DAE problem of index one.

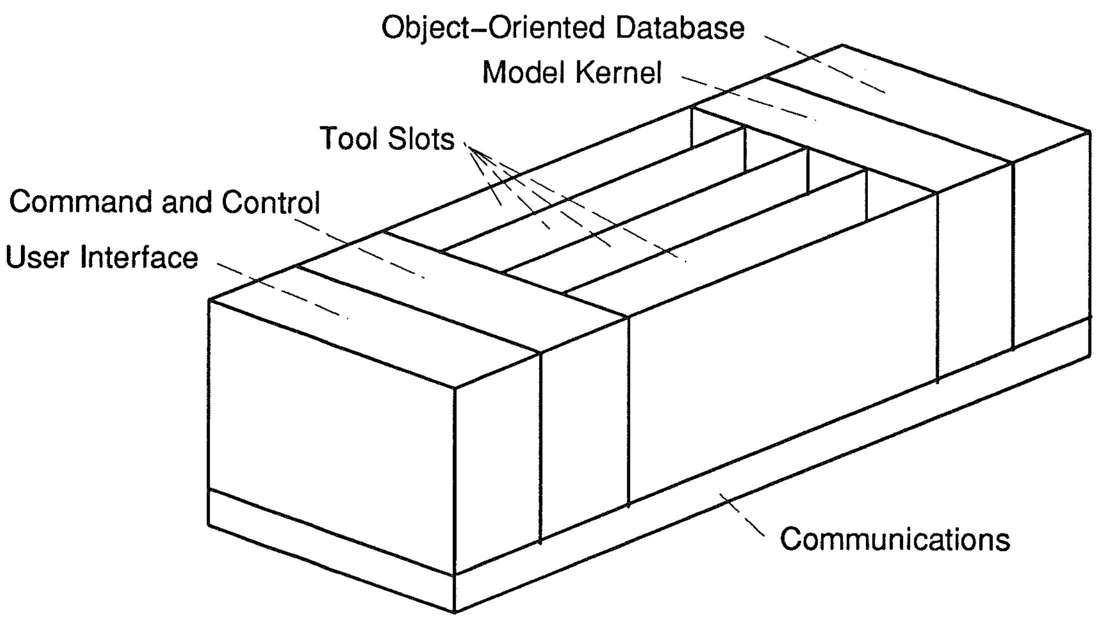


Figure 3.9 CACE reference model suggested in [Barker *et al.*, 1993].

3.5 Multiple Problem Solving Environment

A direct continuation of OmSim is to add new tools for other types of problem solving. There is currently a discussion on how an architecture like this should look. A reference model for CACE is suggested in [Barker *et al.*, 1993]. This is closely related to the ideas in Computer Aided Software Engineering, CASE. The environment model in Figure 3.9 is a modification of the ECMA CASE reference model. ECMA stands for European Computer Manufacturer's Association.

The vision is to have an environment with a common model kernel based on an object-oriented database. These models can be used in different problem solving activities by the tools in the tool slots. The tools use a common interface which supports customized problem specific user interfaces. This software architecture resembles a hardware architecture, where processing cards can easily be changed without any integration problems. In [Barker *et al.*, 1993], Omola is suggested as the model kernel description language.

Multi-Purpose Models in Omola

Omola is a general modeling language with equation oriented descriptions. This means that it is possible to construct general models that can be used for many different problem formulations. One example of how a multi-purpose model is used in a number of different applications is given in Chapter 2. The example is a simple tank reactor and it is described further in the following examples.

The tank reactor example from Chapter 2 has constant flow and volume. The dynamics are described by a dynamic mole balance of one chem-

```
TankReactorModel ISA Model WITH
terminals:
  In ISA RecordTerminal WITH
    Flow ISA ZeroSumTerminal;
    Conc ISA SimpleTerminal;
  END;
  Out ISA In WITH
    Flow ISA ZeroSumTerminal WITH direction:='out; END;
  END;
parameters:
  Volume, ReacCoeff ISA Parameter;
variables:
  Conc, ReacVelo ISA Variable;
equations:
  In.Flow = Out.Flow;
  In.Flow*In.Conc + ReacVelo*Volume =
    Out.Flow*Out.Conc + Volume*Conc';
  ReacVelo = -ReacCoeff*Conc;
  Out.Conc = Conc;
END;
```

Listing 3.6 A simple multi-purpose tank reactor model.

ical component. The inflow and outflow are described by two record terminals. The internal behavior of the model is composed of four equations, one static volume balance, one dynamic mole balance , one reaction velocity equation, and finally one dummy equation declaring that the internal concentration is the same as the outflow concentration. The Omola code describing the tank reactor is found in Listing 3.6.

This simple model can be reused in four different problem formulations as illustrated below. First the dynamic and static simulation prob-

```
DynamicSimProblem ISA TankReactorModel WITH
parameters:
  Volume := 10;
  ReacCoeff := 0.1;
equations:
  In.Flow = 1;
  In.Conc = 1;
  Conc.initial := 0;
END;

StaticSimProblem ISA DynamicSimProblem WITH
equation:
  Conc' = 0;
END;
```

Listing 3.7 The dynamic and the static simulation problem formulation of the tank reactor model.

```
InverseProblem ISA TankReactorModel WITH
parameters:
  Volume := 10;
  ReacCoeff := 0.1;
equations:
  In.Flow = 1;
  Out.Conc = (1 - exp(-0.2*Base::Time));
END;
```

Listing 3.8 The inverse problem formulation of the tank reactor model.

lems and then the inverse problem followed by the design problem. They all reuse the tank reactor model.

In a dynamic simulation problem the tank reactor model is reused and we just specify some parameters and an initial state condition. The mathematical problem is now a well defined differential equation problem and it is found in Listing 3.7. The static simulation problem formulation is a further specialization of the dynamic problem, where also the derivative of the state is set equal to zero. The mathematical problem becomes one algebraic equation with Conc as the only unknown. It is also found in Listing 3.7.

In the inverse problem we use the tank reactor model with the outflow concentration profile specified together with the parameter values. The mathematical problem is one differential equation with a well defined output, which result in the need for differentiation of the output to find the input. The inverse problem formulation of the tank reactor is found in Listing 3.8.

The inflow and the outflow concentrations are specified in the tank reactor design problem formulation. The unknown variable is the volume parameter. Notice that the Out.Conc is a specified constant which means that the equation solver must set the derivative of Conc to zero. The only unknown is the volume which is calculated by an equation solver. The Omola representation of the design problem is listed in Listing 3.9. This problem formulation is mathematically identical to the static simulation

```
DesignProblem ISA TankReactorModel WITH
parameters:
  ReacCoeff := 0.1;
equations:
  In.Flow = 1;
  In.Conc = 1;
  Out.Conc = 0.3;
END;
```

Listing 3.9 The design problem formulation of the tank reactor model.

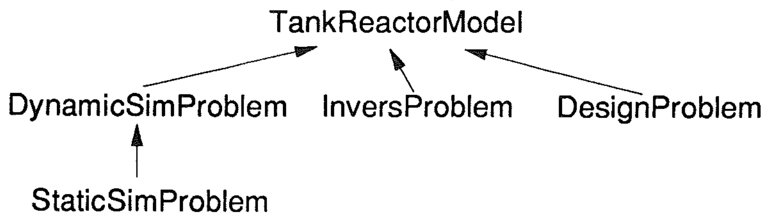


Figure 3.10 The tank reactor model is the super class to four different problem formulations, dynamic and static simulation, inverse problem and the reactor design problem.

problem formulation in Listing 3.7.

The general tank reactor model is inherited in all these examples. This means that the problem formulations are subclasses of the actual model. To develop models should be problem independent. This class hierarchy is illustrated in Figure 3.10. On the other hand, different problems may require different degree of detail which results in different specializations in different problem formulations.

The discussion above illustrates that a general model can be specialized into a set of problem formulations and it means that an explicit model is not necessarily a simulation problem. However the only problem solving that can be solved by the current OmSim is the dynamic simulation problem.

3.6 Other Object-Oriented Approaches

The object-oriented approach to modeling is a very active research area and a number of different languages have been developed. Two other modeling languages are discussed in more detail in this section, namely ASCEND and MODEL.LA. They are both developed for modeling of chemical engineering systems and were both developed, like Omola during the late eighties. An good overview is found in [Marquardt, 1991].

ASCEND

ASCEND was developed at Carnegie-Mellon University by the group under Westerberg, see [Piela, 1989] and [Piela *et al.*, 1991]. It is a model building environment for complex models consisting of large sets of simultaneous nonlinear algebraic equations. The ASCEND language is a textual description of models which is parsed into the database of the ASCEND environment. The ASCEND environment is based on Design

Systems Laboratory, DS-Lab, which is a programming environment developed at Carnegie-Mellon University.

The model building blocks are models, atoms and types.

1) *Models* are structured types built hierarchically from instances of other models, instances of atoms and types, and relationships between instances.
2) *Atoms* are primitive variables used to represent physical quantities and they are of a particular type.
3) *Types* are elementary types such as real, integer, boolean etc. These types are predefined in the language.

Models and atoms are organized in simple inheritance hierarchies in which every interior type has an unique, immediate parent. ASCEND has single inheritance and has three different concepts for inheritance.

- The *REFINES* operator describes the inheritance between classes.
- The *IS_A* operator describes the instantiation of classes and types.
- The *IS_REFINED_TO* operator changes the type associated with a previously declared instance.

Relationships between models, atoms, and types are declared in statements. Notice that ASCEND has instances and classes. Instances cannot have local attributes. The relation *ARE_ALIKE* is used to group instances together. This relation is similar to making a connection between terminals. The relationship *ARE_THE_SAME* is similar to ARE_ALIKE but merges the operands into a single structure. Mathematical relations can be expressed as equations and assignments. Relations can be named and the syntax is *name : relation.* A reference to an element of a structured

```
MODEL valve REFINE model;
  in      IS_A pipeterminal;
  out     IS_A pipeterminal;
  density       IS_A real;
  valvecoeff    IS_A real;
  pipearea      IS_A real;
  in.flow = out.flow;
  valvecoeff*(in.pressure - out.pressure) =
    density*(in.flow/pipearea)^2/2;
END valve;

MODEL pipeterminal REFINE stream;
  flow      IS_A real;
  pressure IS_A real;
END pipeterminal;
```

Listing 3.10 A simple valve example in ASCEND.

type is done with dot notation.

An ASCEND example of the valve class discussed earlier is illustrated in Listing 3.10. ASCEND has structuring mechanism for building hierarchies but it does not have encapsulation with well defined submodel interfaces. All elements in a model are open and can be referred to using dot notation. This problem, if it is a problem, does not appear in the small example above. Submodels with boundaries and interfaces or open structures with visible internal descriptions is perhaps a philosophic question.

MODEL.LA.

MODEL.LA. is developed at the Massachusetts Institute of Technology by the LISPE group under G. Stephanopoulos, see [Stephanopoulos *et al.*, 1990a] and [Stephanopoulos *et al.*, 1990b]. The structure of process models is depicted by specific digraphs, which are symbolically constructed by algorithmic procedures. These procedures are driven by the context of the modeling activity. MODEL.LA. is implemented in KEE, Knowledge Engineering Environment [IntelliCorp, 1987]. It does not have any textual description like Omola and ASCEND. The semantic network is based on six modeling elements and eleven semantic relationships. The modeling elements are the following:

- *Generic-Unit (GU)* is an encapsulated system that has well defined boundaries. Derived subclasses are plant, plant-section, unit and subunit.
- *Ports* are the entities in a GU which describe the interface. Derived subclasses of port are convective, non-convective and information.
- *Stream* is the entity that describes the connection between ports. Derived subclasses from stream are similar as for port subclasses.
- *Modeling-Scope* is a set of declarative relationships, which apply to all the model components. It has two important subclasses, *Context* and *Model*. In context, assumptions about the model are declared and in model the actual relationships between model elements are described.
- *Constraint* is a declarative relationship between quantities. Relation and assignment are subclasses of constraint.
- *Generic-Variable* is the basic building block for constructing modeling relationships. It encapsulates the following entities: physical significance, value, range, units, trends, etc. It has two subclasses, namely variable and term.

The subclasses of GU are used to create a hierarchical decomposition of the plant model. Ports are the terminal class, and streams are the connection class. Model, constraint, and generic-variable are classes which are used for primitive model descriptions. The context class is a model element that has no equivalence in ASCEND and Omola.

Eleven semantic relationships describe the dependencies between the modeling elements. The relationships can often be grouped two and two because they are the inverses of each other.

- *Is-a* relations build up the class hierarchy. It governs the inheritance.
- *Is-a-member-of* relations describe generic/individual links between object and instantiations.
- *Is-composed-of* and *Is-part-of* links build up the model hierarchy.
- *Is-attached-to* and *Is-connected-by* are the links between streams and generic-units to ports.
- *Is-described-by* and *Is-describing* are links between objects of generic-units, streams and ports to the mathematical description objects like generic-variables and constraints.
- *Is-disaggregated-in* and *Is-abstracting* are links between elements in different contexts.
- *Is-characterized-as* is a special semantic link that describe the relation between an element and an attribute.

The first semantic relation is explicit in both Omola and ASCEND and the second relation in ASCEND. The other semantic relations are all implicit in Omola and ASCEND.

Comparison of Omola, ASCEND and MODEL.LA.

The modeling languages and the implementations of the modeling systems Omola, ASCEND and MODEL.LA. are very different but the languages share a number of common concepts. Most of the model building elements are equivalent in Omola, ASCEND and MODEL.LA., capturing the structured modeling concepts discussed earlier. The only major difference is that ASCEND has open submodels while the others have models with boundaries and interfaces. The object-orientation mechanisms are also similar. Omola and ASCEND have single inheritance and MODEL.LA. has multiple inheritance. Omola does not have instances as a modeling concept which the other two have. Finally both Omola and ASCEND are textual languages while MODEL.LA. is an user application on top of KEE. The implementations of the three and the problem solving tools are very different. Omola is the only one currently implemented in a compiled language.

Other Modeling Languages

The three languages discussed above are similar and there are other languages are under development in the same spirit.

DYMON developed by Lund [Lund, 1992], is a prototype of a dynamic process modeling environment, similar to the three discussed above. It is based on a textual description, closely related to Omola and ASCEND. The main difference is that DYMON uses a dynamic sequential modular simulation tool [Hillestad and Hertzberg, 1988].

gPROMS is a general process modeling system used in combined discrete event and continuous time simulation. It is under development at Imperial Collage, see [Barton and Pantelides, 1991] and [Pantelides and Barton, 1992]. gProms supports both structure and class hierarchies. The actual description language is a further developement of SPEEDUP, [Pantelides, 1988]. The continuous part, called model, can handle DAEs. It also has a task concept which is similar to events in Omola. A task can generate other tasks and it can have actions. Models and tasks are combined in a single entity called process. A solver that also can handle partial differential equations is under development.

VeDa stands for "verfahrenstechnisches datamodell" and is under development at Stuttgart University [Marquardt, 1993]. It also supports structure and class hierarchies. VeDa has something called "phenomenological modeling objects" creating a primitive model hierarchy, e. g., an equation system is composed of equations which are composed of variables which can be expressed by other variables and so on. VeDa is supposed to be implemented on top of an object-oriented database management system. An automatic code generator transforms the models into a procedural representation, i. e., FORTRAN code. This code can be used in the simulation environment Diva, see [Marquardt *et al.*, 1987] or [Kröner *et al.*, 1990]. Diva can handle DAEs with discontinuous events. One DAE solver is modified to use sparse matrix techniques.

Modeling Assistants

Modeling assistant is another active research area in computer aided modeling. The idea is to use expert system techniques to develop intelligent help systems that assist the model developer. These techniques make it possible to have advanced user interfaces with customized model development tools for certain applications.

DESIGN-KIT is a modeling assistant from MIT, [Stephanopoulos *et al.*, 1987], and it is a precursor to MODEL.LA. Like MODEL.LA., it is an user application on top of KEE and is heavily based on object-oriented programming. The use of multiple inheritance is intersting in DESIGN-KIT. A process object inherits attributes from a number of super classes. This means that a process object can be described directly by selecting super classes from a library or, as in DESIGN-KIT, from a menu.

A mixing drum example in DESIGN-KIT, from [Stephanopoulos *et al.*, 1987], is shortly presented below to illustrate the ideas. A mixing drum inherits attributes from six super classes. From MIXING-DRUM-ICON it inherits the icon and from PHYSICAL-VERTICAL-DRUM it inherits methods for designing and sizing. The super class COMPONENT-CONSERVATION has attributes describing *n* number of material balances and that *no* reactions occur. The properties inhertited from TWO-INPUTS-ONE-OUTPUT-TOPOLOGY super class together with the previous inherited properties exactly describe the material balances. ADIABATIC super class together with the two previous super classes describe the energy balance. The ISOBARIC super class describes that there is no pressure dependency. The mixing drum example has six super classes and the inherited attributes are interpreted as a set of equations.

ModAss is an intelligent modeling assistant, developed by Sørle [Sørlie, 1990] at the Norwegian Institute of Technology. It is based on another approach compared to DESIGN-KIT, namely the use of expert systems. ModAss is capable of supporting both process knowledge and general modeling knowledge to the user. A prototype is implemented in Knowledge-Craft, an interactive programming environment, similar to KEE. A separate program, called the ModelAss supervisor, operates in the background. It is only visible when the user makes obvious mistakes or asks for expert advice. The blackboard concept is used to organize different knowledge sources which are monitoring the modeling process. Each knowledge source submits a bid if it is trigged and the best bid is allowed to give its expert advice to the user.

PROFIT and HPT are model assistants for process modeling from first principles and they are also developed at the Norwegian Institute of Technology by Telnes, [Telnes, 1992], and Woods,[Woods, 1993], respectively. PROFIT and HPT are implemented in LISP with the object-oriented enhancement CLOS. PROFIT has a graphical interface for displaying three dimensional process pictures. The process equipment geometry is described interactively by the use of menus. Description of phases and reactions are done in a similar way. These descriptions are then used to generate the model resulting in a mathematical description of the equipment. An inference engine is used to identify phase interactions, conservation equations, and reaction influences. HPT stands for Hybrid Phenomena Theory and is based on the ideas from Qualitative Process Theory, QPT. The basic idea is that phenomenon and topological descriptions are orthogonal. The user build a *topological model* which is used together which predefined phenomena definitions to generate a qualitative model, a *phenomenological model.* By adding activity conditions, a *quantitative model* on state space form can be generated. HPT can express a consistent relation between the qualitative and quantitative description of a system.

3.7 Conclusions

Object-oriented modeling in general and the modeling language Omola in particular are presented in this chapter. Omola is shown to capture the basic concepts in structured and object-oriented modeling. Structured modeling concepts are modularization of large models into an hierarchy of modules with well defined interfaces. Omola clearly describe the model interface and model interior. It is possible to develop composite models with hierarchical decomposition of the internal structure into submodels. The models have a well defined interface with terminals and parameters. Object-oriented modeling concepts are focused on reuse. Models are defined as classes in a single inheritance class hierarchy. A model class inherits model components from the super class. A primitive model has an internal behavior expressed by variables, equations, and events. General models can be described by Omola which then can be used in a multi-tool environment.

OmSim is the Omola simulation environment where Omola models can be developed and simulated. It is implemented in C++ using Unix, X-Windows and Interviews. The OmSim simulator can handle differential and algebraic equation systems combined with discrete events.

Two other object-oriented languages, ASCEND and MODEL.LA., are also presented and they are found to have similar properties as Omola.

4 Abstraction and Reuse

Chemical plants are composed of many process units connected in more or less complex structures. Descriptions of systems like this gain a lot by the use of abstraction. Abstraction is created by the use of decomposition of a large model into well defined submodels with minimal interfaces. This decomposition can be done recursively to create a model structure hierarchy. The development of complex systems is also facilitated by the use of common unit operations. The plant development and the model development are very similar. In traditional flowsheet packages the plant is described by a flowsheet composed of predefined unit modules, see Appendix A. The unit modules are typical unit operations commonly used in chemical processes.

In Omola it is possible to develop hierarchical flowsheets by the use of composite models. The predefined unit modules are classes in the model database. The user can also develop new unit classes by specializing an old unit class. Reuse is accomplished by the use of inheritance in object-oriented modeling. A class inherits properties from its super class, which inherits from its super class, and so on. This creates a class hierarchy.

Process structure hierarchy is discussed in the first section which gives guidelines for process decomposition. Section 4.2 discusses process model interaction in detail. The use of inheritance and the class hierarchy categorization is discussed in Section 4.3 and it is also ended with process class hierarchy guidelines.

4.1 Process Structure Hierarchy

The topology of a chemical plant is described in a flowsheet. Recall that a flowsheet is a two dimensional topology map describing how units in the plant are physically connected to each other. The degree of detail varies depending on the purpose of the flowsheet description. In a flowsheeting package the elements in the flowsheet, the unit modules, are unit operations and similar components. This means that the complexities of the unit modules are very different, from simple mixing and splitting drums to multi-fraction distillation columns. In order to have single layer flowsheets with well defined modules, some modules must necessarily be more complex than others. The user wants to have well defined unit modules. The topology of the process flowsheet is described in a single layer which means that it is not possible to develop subflowsheets of parts of the process that are particularly complex. Examples of the use of flowsheeting packages are found in [Aspen, 1982] and [Perkins and Sargent, 1982]. An ASPEN PLUS example is also found in Appendix A.

A natural extension is to allow hierarchical structures. A flowsheet can be composed of submodels that are subflowsheets, and so on. A unit in a flowsheet can have an internal structure of other units. This is the same thing as the composite model description discussed earlier. By the decomposition of one unit into components the internal details of the unit are abstracted and the structure becomes more comprehensible.

Process Structure Decomposition Guidelines

One suggestion for a structure hierarchy is the following: plant - plant section - unit - subunit.

- *Plant* is the super object describing the whole processing system. It is composed of a number of plant sections for feed treatment, reaction, separation, recovery etc.
- *Plant section* is composed of units and is a part of the plant object.
- *Unit* is the basic predefined building block in the flowsheet and it corresponds to the unit operations concept, like reactor, distillation column etc. A unit can have an internal structure of subunits. More complex units can have an internal structure of other units.
- *Subunit* objects are used to abstract the internal behavior of an unit and to increase the reuse of parts of the unit behavior.

The decomposition guidelines sometimes include too many levels and sometimes too few. In a small plant the plant section level becomes unnecessary. For some units the subunit are uninteresting. On the other hand, there is a need for more than one plant section level in very complex struc-

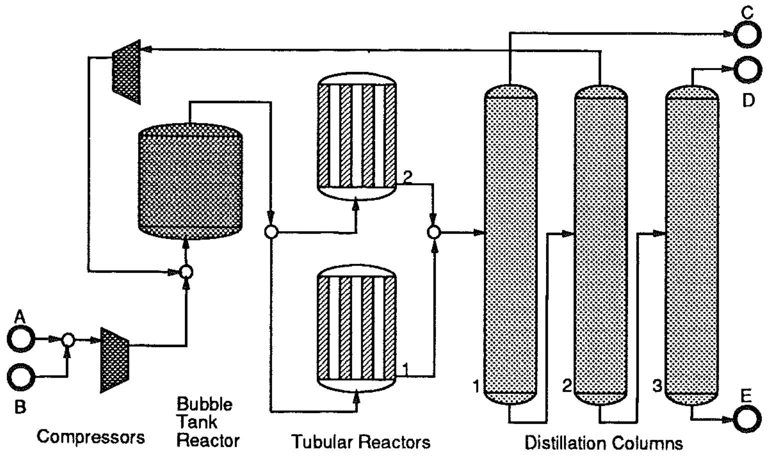


Figure 4.1 A typical flowsheet of a minor chemical plant.

tures , e. g., a separation section is composed of liquid and vapor recovery sections and a product fraction section. Complex units, such as distillation columns, need perhaps more then two levels of abstraction for the internal description. A similar decomposition is found in [Stephanopoulos *et al.*, 1990a], where the following decomposition levels are supported: Plant - plant section - augmented unit - unit - subunit. The augmented unit is a unit composed of other units.

A Plant Example

The structure hierarchy guideline above is exemplified by a chemical plant, see Figure 1.1. The feed is mixed with a recycle stream and fed into a bubble reactor, which is the first reaction step. In the second reaction step the stream enters two parallel tubular reactors. The product stream then goes to the separation section which is composed of three distillation columns. The example is taken from [Nilsson, 1989a]. With the use of the decomposition guidelines above the plant is decomposed into three plant sections: feed preparation, reaction and separation. The decomposition is seen in Figure 4.2. The separation plant section is composed of three distillation column units connected in series. A distillation unit has an internal structure of components, like column, condenser, reflux drum, reboiler etc. The reboiler component also has an internal structure of subcomponents: heating side, boiler side, and heat transfer wall. The heat transfer wall model has no internal structure and is therefore described as a primitive model with equations.

In Omola, structure hierarchies can be represented by composite models inside composite models. This creates a structure tree where the leaves are primitive models and the branches are composite models.

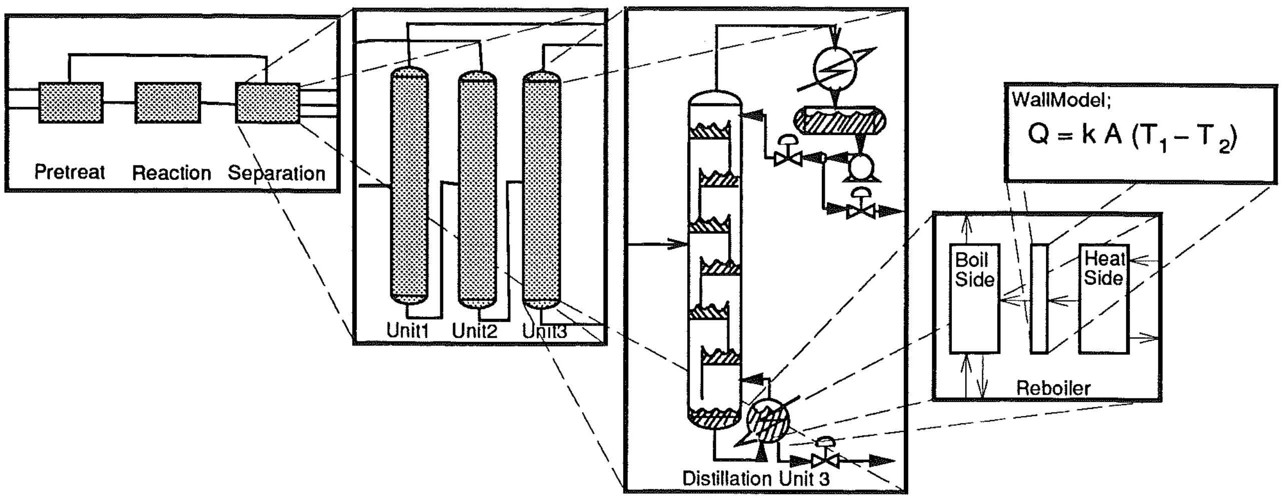


Figure 4.2 An example of hierarchical structure decomposition of a plant flowsheet. The plant is decomposed into plant sections, the separation plant section into distillation units, the distillation unit into other units, the reboiler unit into subunits and the boiler heat transfer wall subunit has a primitive description.

A Tank Reactor Example

The reaction plant section is composed of reactor units, like a continuous stirred tank reactor, CSTR. A unit operation like a CSTR is decomposed into internal subunits associated with certain aspects of the unit. The reactor unit is a *jacketed cooled tank reactor* which is a vessel cooled by a surrounding jacket. The reactor object is therefore decomposed into one cooling jacket object, one reactor vessel object, and one heat transfer object, the wall or boundary. This is seen in Figure 4.3. The chemical reaction occurs in the reactor vessel and thus generates heat. The heat transfer is modeled in a separate model which is connected to the vessel and the jacket, see Figure 4.3.

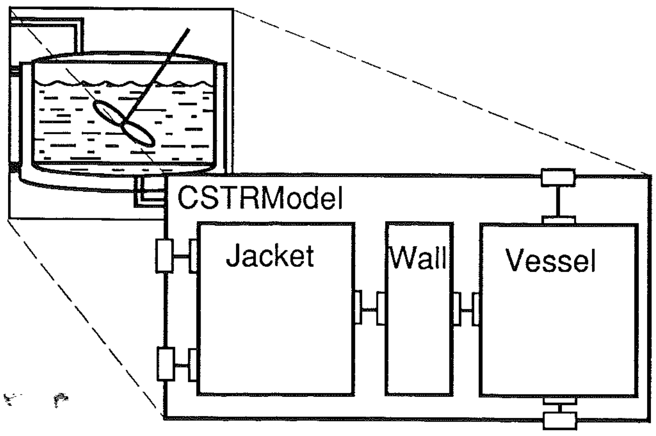


Figure 4.3 The decomposition structure of the CSTRModel in the tank reactor example. The composite model of a CSTR is decomposed into three submodels, namely Jacket, Wall and Vessel.

```
ProcessInTerminal ISA RecordTerminal WITH
  Flow ISA ZeroSumTerminal WITH
    direction := 'in;
  END;
  Composition ISA SimpleTerminal WITH
    n       TYPE Integer;
    value   TYPE column[n];
    default TYPE column[n];
  END;
  Enthalpy ISA SimpleTerminal;
  Pressure ISA SimpleTerminal;
  NoComp TYPE Integer;
  Composition.n := NoComp;
END;
```

Listing 4.1 An example of a process flow pipe terminal. It is composed of four subterminals, one integer variable and one parameter assignment.

4.2 Interaction

The decomposition of processes in the previous section is used to structure the process flowsheets in a natural way. This results in a structure hierarchy of submodels. The submodels interact with each other through terminals which are connected, as discussed in Chapter 3. The interaction description is also abstracted in the same way as the process structure. A number of basic terminals can be grouped together in a record terminal description. Record terminals can also be defined as subterminals in a record terminal hierarchy. This was discussed in more detail in Section 3.2.

A Process Terminal Class

In process applications there are a number of common unit interactions. The most obvious is the flow pipe, where a forced flow of process medium enters or leaves a unit. Other process interactions are, e. g., fluxes of mass and heat.

The process medium is forced to flow from one unit to another in a process flow pipe. This interaction is described by the amount of flowing medium, e. g., mass flow, and the internal state of the medium, e. g., composition, enthalpy and pressure. The composition is a vector terminal with the length, n, equal to the chemical dimension of the medium, e. g., the number of chemical components, NoComp. An example of a process pipe terminal is listed in Listing 4.1. A connection between two process pipe terminals is interpreted as a set of equations. In the example it results in four equations, where one equation is in vector form. The flow terminals are summed to zero because they are derived from a zero sum

```
LiquidInTerminal ISA ProcessInTerminal WITH
  Phase ISA SimpleTerminal WITH
    value TYPE String := "Liquid";
  END;
END;
```

Listing 4.2 A liquid flow terminal is a subclass to the process terminal with an additional `Phase` attribute.

terminal class. The other medium state terminals are set equal to each other because they are simple terminals. The process terminal class name indicates the positive value of the flow component. The terminal class in Listing 4.1 can be used for describing a medium inflow to a unit model.

User Defined Semantics

A connection between two terminals is consistent if each simple terminal pair can be connected to each other, see Section 3.2. Terminals describing a vapor flow and a liquid flow can have exactly the same internal terminal structure. This means that they can be connected to each other which is undesired. One way to prevent this is to add a terminal component describing the phase of the medium in the process terminal. This component is a simple terminal assigned a constant value. A connection between two constant terminals is consistent if they have the same value. This is illustrated in the example in Listing 4.2. The process terminal is specialized by adding a phase attribute. This phase attribute is assigned a string value.

The Tank Reactor Example

To exemplify process interactions the tank reactor example is discussed in more detail. The tank reactor is a continuous stirred tank reactor, CSTR. A chemical reaction is assumed to occur, $A \rightarrow B$, which produces heat. A tank reactor decomposition is illustrated in Figure 4.3. The reactor is decomposed into one jacket submodel, one wall submodel, and one vessel submodel. The internal descriptions of the models together with the internal description of the terminals are seen in Figure 4.4. The figure is intended to illustrate the interaction between the submodels.

`Jacket` is modeled by a dynamic energy balance and a static volume balance. The jacket has two flow terminals, `In` and `Out`, describing the flow and temperature of the flowing cooling medium. It has one terminal describing the heat transfer to the vessel, `Ht`. The terminal is composed of the jacket temperature and the transferred heat. If the transferred heat is known together with `In` terminal, the model describes the temperature in the jacket.

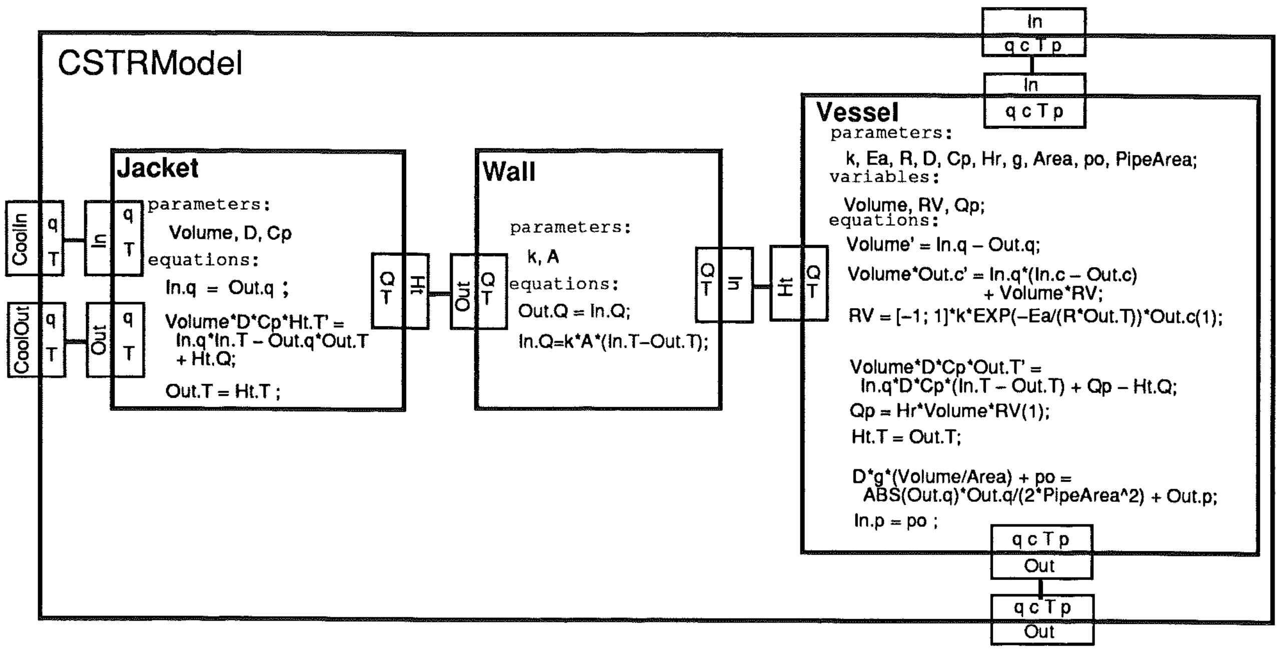


Figure 4.4 The internal details of the primitive models, Jacket, Wall and Vessel, in the composite model, CSTRModel.

The Wall object represents the heat transfer interaction between the jacket and the vessel. The transferred heat is a function of the temperature difference on each side of the heat transfer object, the vessel wall, see Figure 4.4. The heat transfer is described by a simple static equation.

Vessel is modeled by one dynamic volume balance and one dynamic mole balance on vector form of dimension two, A and B. The volume and the composition in the vessel is described by the first two equations in Figure 4.4. The reaction velocity, called RV, is also a vector variable. The fourth equation is the dynamic energy balance and it is followed by a description of the energy production caused by the reaction. The sixth equation in Figure 4.4 defines the equivalence between the temperature subterminal in the outflow and in the heat transfer terminal. Finally there is a Bernoulli equation describing the mechanical energy balance and an equation tying the inflow pressure to a parameter.

A dynamic simulation problem formulation of the CSTRModel will add a number of assignments to the model. The models contain 50 variables, basic terminals and parameters. They also contain 13 explicit equations and 16 equations generated from connections. A problem formulation must add 21 assignments. 15 parameters and 6 subterminals must be assigned values in order to make a well defined simulation problem. The subterminals are the flow and temperature of the CoolIn terminal, the flow, composition and temperature of the In terminal and finally the Out pressure subterminal.

4.3 Process Model Class Hierarchy

Models are represented as classes in a class hierarchy in object-oriented modeling. Through inheritance a model class reuses the description of previously defined class. The first class is a subclass of the second class. This is a generalization of the module reuse that is possible in current modeling environments. In flowsheeting, process unit modules can be selected from a library to form a process topology. The class hierarchy and the inheritance concept can be used for reuse, polymorphism, and organization. Model reuse is achieved in three different ways as discussed in Chapter 3.

- The first way is to use a predefined model as a unit directly in a flowsheet model. In Omola this is the same as making a subclass definition of a global model class as an attribute of a composite model. The global class is a unit model class and the composite model is a flowsheet. This is a direct reuse and it is supported by many systems today, e. g., [Aspen, 1982].
- The second way of reuse is a specialization of a predefined global model class to a specialized global subclass. The specialization consists of the addition of new model components and equations.
- The third way of reuse is the use of polymorphic models. If two submodels are polymorphic they can be used in the same composite model. Polymorphic models have equivalent interface structures which facilitate the reuse of structures.

Reuse, specialization and polymorphism will be illustrated in the tank reactor example in this section.

Reuse

The tank reactor in the example in Section 4.2 is decomposed into three submodels. The components are assumed to be classes in a class hierarchy or, in other words, the components of a composite unit class are local subclasses of global subunit classes found in the model database. The subunits of the tank reactor are the reactor vessel model, the cooling jacket model and the heat transfer wall model, which are local classes inside `CSTRModel`. These local classes can be subclasses of global classes in a class hierarchy. This is an example of direct reuse of predefined classes, which is illustrated in Figure 4.5.

Direct reuse of submodels creates a flat class hierarchy where every model class is a subclass of the predefined `Model` class. The CSTR class is a composite model with attributes that are local subclasses of global classes.

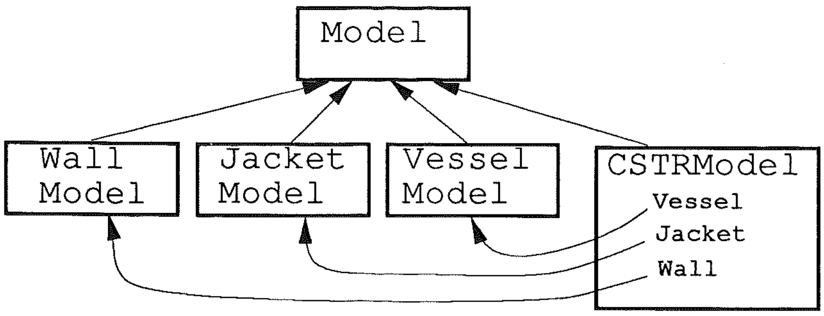


Figure 4.5 An example of a flat class hierarchy with poor reuse and polymorphic possibilities.

Specialization and Polymorphism

Experience has shown that it is rather difficult to make complicated specializations, such as adding equations to an already complete model. The global model class definition should instead be decomposed into classes in a hierarchy. One example is the change of the wall model to a dynamic description, `NewWall`, see Figure 4.6. If the `NewWall` class has the same interface as `Wall` they are polymorphic. The point is to make it possible to reuse the composite `CSTRModel` and just overwrite the local `Wall` definition attribute. It is convenient to create a common super class for the two wall classes. This class is a pure interface definition. `WallIClass` does not have any internal behavior description so it can therefore not be simulated. The postfix `IClass` indicates that the class is an interface class for polymorphic models. The difference between the two `WallIClass` subclasses are the internal primitive description of the behavior.

If the definition of physical objects is spread out in a number of classes in an inheritance hierarchy the reuseability of these classes will increase, particularly for the super classes high up in the hierarchy. The most important result is the polymorphic aspect of the distributed unit descriptions, which makes the reuse of the composite models easy. The described method of class decomposition increases the depth of the class hierarchy with some levels.

The demands on polymorphic models are interface similarities and the same degree of freedom.

- Terminals of polymorphic models must be compatible. This means that the interactions with surrounding models are invariant.
- Parameters that are assigned values by the super model must be identical in polymorphic models. It is possible to have additional parameters that are assigned values locally.
- The degrees of freedom of polymorphic models must be the same. The degree of freedom, d, is the difference between the number of variables, n_v, and the number of constraints, n_c,. Variables are interface

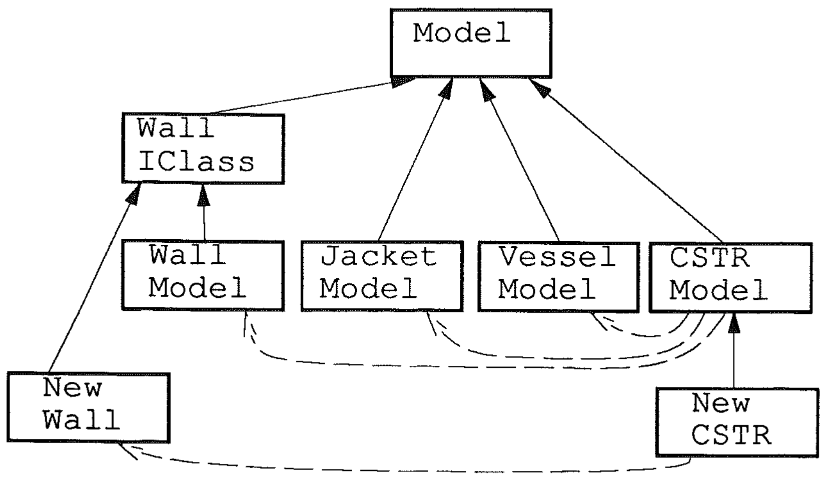


Figure 4.6 The class hierarchy shows how polymorphic wall classes can be used to increase the CSTR structure reuseability. (arrow - *is-a* relation, dashed arrow - *has-a* relation.)

variables, such as terminals and externally assigned parameters, and interior variables, such as locally assigned parameters and internal variables. Constraints are equations and assignments.

$$\begin{aligned} d &= n_v - n_c \\ &= n_{v_{interface}} + n_{v_{interior}} - n_c \\ &= (n_{Term} + n_{ExtPar}) + (n_{IntPar} + n_{IntVar}) - (n_{Equ} + n_{Ass}) \end{aligned}$$

Polymorphic models must have the same interface which means that the number of interface variables are the same. If the number of constraints is increased in one polymorphic model then the number of interior variables, such as internal variables and locally assigned parameters, must also be increased to keep the degree of freedom constant. A good rule is to defined the interface variables in the interface class and defined the interior variables in the model class.

Library Organization

In a process application the number of classes in the model data base will be large, from hundreds to thousands. It will be hard to find the class that is of interest. The majority of the classes are not unit model classes, like the CSTR unit, that are directly reusable in flowsheet composite models. There are terminal and parameter definitions and incomplete super classes to unit and subunit classes. The class hierarchy must be organized and structured in order to increase the comprehensibility of the database. Super classes can also be used to create a better tree structure. The main purpose of these super classes is to make it possible to group classes together that conceptually belong to the same category.

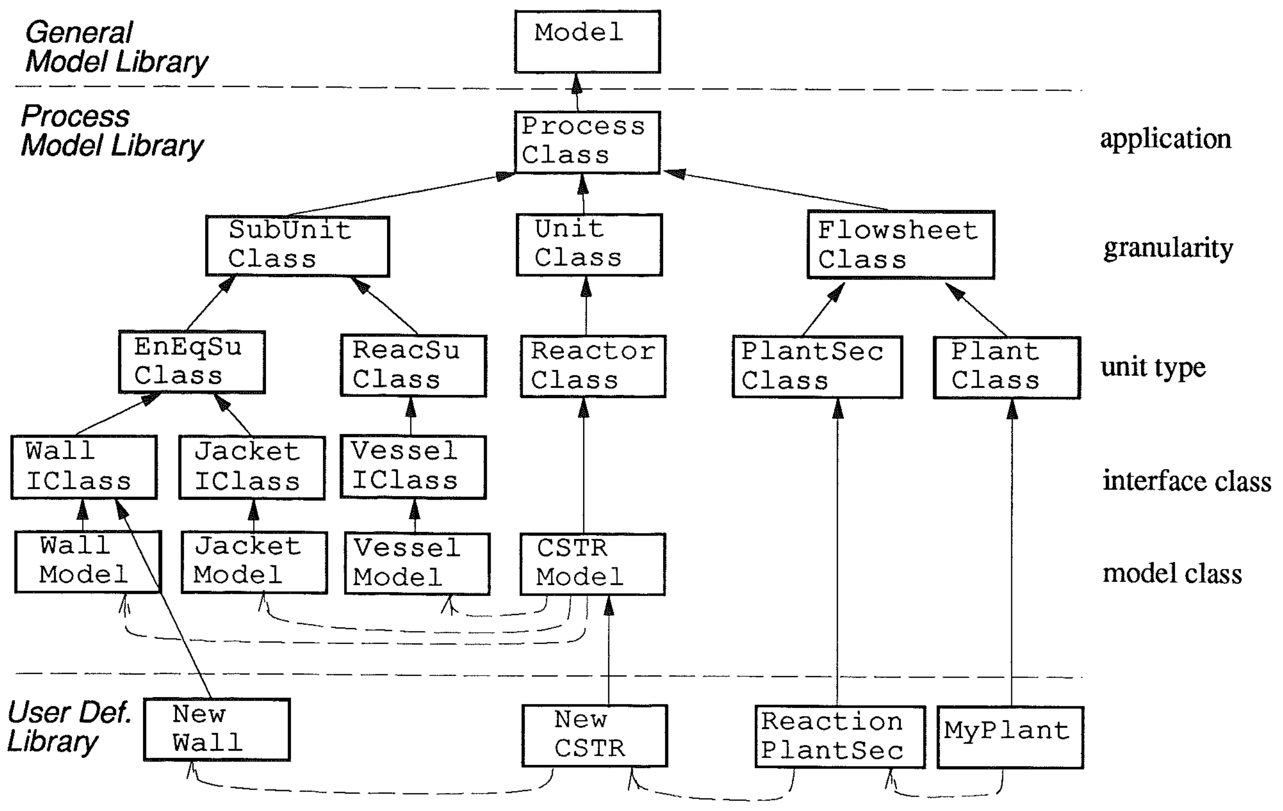


Figure 4.7 The CSTR hierarchy with additional classes structuring the class tree. The reusable CSTR classes are leaves on the CSTR tree, a subtree in the class tree. (arrow - *is a* relation, dashed arrow - *has a* relation.)

Four new super classes are added in Figure 4.7, compared to the class hierarchy in Figure 4.6. The process model class, `ProcessClass`, makes an application type categorization of the tree. Examples of other application type classes are `ControlClass`, `MechanicClass` etc. The postfix `Class` indicate that this class is just an empty classification class. The next added class level is a process model categorization into types of different granularity, e. g., flowsheet, unit and subunit. A flowsheet class is composed of units and a unit is composed of subunits. The next level divides the tree into process unit types. The subunit class tree is divided into energy equipment subunits and reactor subunits. The unit class tree has only a reactor class; other examples are separator class and energy equipment class. The flowsheet class is divided into plant section class and plant class on the unit type level. The actual model classes for reuse and specialization are found below this level. In the example a new CSTR is specialized to have a new wall description and it is also used in a reaction plant section in a plant model called `MyPlant`.

Process Class Hierarchy Guidelines

Guidelines for the structure hierarchy was discussed in Section 4.1. Similar guidelines for the process model class hierarchy are suggested below. The guidelines are illustrated in Figure 4.7.

- The *application type* categorization class is used to make a rough decomposition of the model data base into different application areas. Examples are `ProcessClass`, `ControlClass` etc.
- The *granularity type* classifies process classes into different branches of the tree depending on the grain size of the model. Examples are `FlowsheetClass`, `UnitClass` and `SubUnitClass`.
- The *unit type* categorization class is used to structure the class tree into different unit subtrees. Examples of unit classes are `ReactorClass` and `SeparatorClass`.
- The *unit interface class* is the root class for polymorphic classes concerning a particular unit. The `WallIClass` is an example in the Figure 4.7 which has two subclasses.
- The *unit model* is the leaf in the library class tree and is the actual model used in different problems. It can be reused in other composite models or specialized to new models. An example is `CSTRModel` in the example above.

Once again this class structure is a guideline for object-oriented modeling. The class hierarchy has four purposes, namely direct reuse, specialization, structure reuse through polymorphic models, and for class tree organization.

4.4 Conclusions

The abstraction of large models is done by hierarchical decomposition of the model into well defined submodels with minimal interaction. One of the main conclusions of this chapter is *process structure hierarchy guidelines*. A plant model is suggested to be structure decomposed into the following levels; `plant - plant section - unit - subunit`.

Abstraction is also aided by reuse. The ability to reuse predefined submodels makes it easy to define large structures of submodels. Reuse is categorized into three different methods. *Direct reuse* is done when a model class is reused as a submodel in a composite model. A model class can also be reused through *specialization*, where additional properties are defined in a subclass. One example is the specialization of a model interface class into a model class. A third method is *polymorphism*. Polymorphic models have the same interfaces and they can therefore be exchanged in the same composite model. The suggested guidelines for *class hierarchy decomposition* are `application - granularity - unit type - interface - model`.

5

Decomposition of Model Behavior

This chapter discusses the use of decomposition without a corresponding physical pattern. The decomposition of structures into smaller modules in a hierarchy has being discussed and exemplified in Chapter 4. The decoupling of media and unit descriptions is a common structuring feature in flowsheeting. The model developer can select units from a unit library and make a medium description separately. The medium models and physical parameters are automatically found in a medium database, see [Aspen, 1982]. A small ASPEN PLUS example is found in Appendix A. An object-oriented approach to medium and unit decomposition is discussed in the first part of this chapter. The basic tools for this are polymorphic medium models and unit models.

The second part of the chapter deals with the decomposition of primitive behavior descriptions into behavior components, i. e., equation objects. Each individual unit and medium model is described by a set of equations. Each equation describes a certain physical property. To support development of new units that are missing in the model database, different behavior characteristics can be reused from a library. This library contains equation objects that may be easily put together to create a new model. The discussion concludes that new class description concepts are needed.

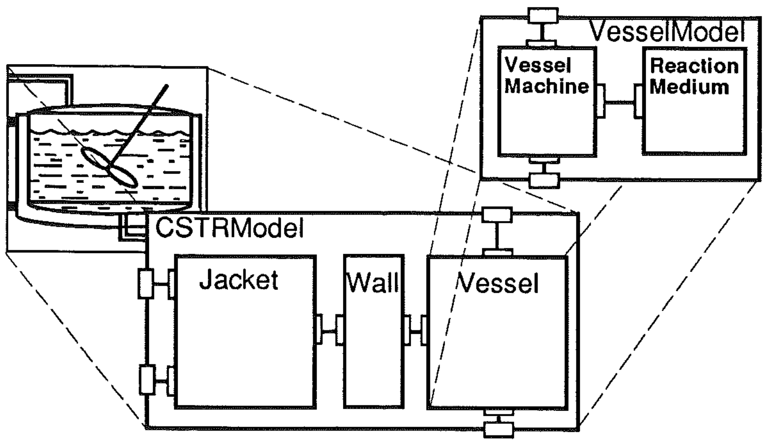


Figure 5.1 A medium and machine (unit) decomposed reactor vessel model is the second level of decomposition of the CSTR.

5.1 Medium and Unit Decomposition

The traditional decomposition of medium and unit in flowsheeting is to view the medium model as a function, notice the FORTRAN association. The unit modules make function calls to medium property functions. The model user can separately describe the medium of interest. The flowsheet package can then make proper parameter selections from a medium database to tune the medium function correctly according to the selected medium. This section and the following section discuss how to do this in an object-oriented modeling language like Omola. The discussion is based on the tank reactor example defined in the previous chapter, Chapter 4. A better name is medium and unit machine decomposition or just medium and machine decomposition.

Medium and machine decomposition can be done in two ways using different modeling concepts, structure decomposition and multiple inheritance. The *medium and machine structure decomposition* can be based on a structural decomposition of an unit model into one unit machine submodel and one medium submodel. The medium and machine models interact through a medium and machine connection with terminals. This is illustrated in the Figure 5.1. Another approach is to use *medium and machine multiple inheritance* wherein the machine and the medium are viewed as two separate super classes to the unit model. The structure decomposition approach is discussed in this section and the multiple inheritance approach is discussed in the following section.

The Tank Reactor Vessel Model

A mathematical description of the model is needed to make a detailed discussion about the decomposition properties of different approaches. The mathematical description of the tank reactor vessel is as follows. The model of the vessel is a set of four nonlinear differential equations:

$$\rho\frac{dV}{dt} = \rho q_{in} - \rho q_{out} \tag{5.1}$$

$$\frac{d(Vc)}{dt} = q_{in}c_{in} - q_{out}c + Vr \tag{5.2}$$

$$\rho C_p\frac{d(VT)}{dt} = \rho C_p q_{in}(T_{in} - T^r) - \rho C_p q_{out}(T - T^r) + Q_p - Q_t \tag{5.3}$$

$$\rho g\frac{V}{A} + p_t = \frac{|q_{out}|q_{out}/a^2}{2} + p_{out}. \tag{5.4}$$

The equations describe the vessel dynamics, i. e., dynamic material and energy balances. The equations assume perfect mixing in the vessel, homogeneous concentrations and temperature. The medium model is the medium parameters, ρ, C_p and T^r, together with the medium functions describing, r and Q_p, reaction velocity and reaction produced heat:

$$r = [r_A \quad r_B]^T = [-1 \quad 1]^T k_0 e^{-\frac{E_a}{RT}} c_A \tag{5.5}$$

$$Q_p = H_{reac} V r_A. \tag{5.6}$$

Static Medium Based Decomposition

One way of medium and machine decomposition is shown in Figure 5.2. The machine model contains the main behavior description, e. g., balance equations, and machine parameters, e. g., tank cross area. The medium model contains the medium behavior, e. g., reaction velocity, and medium parameters, e. g., medium density. This means that the submodels can be changed independently of each other as long as their interfaces remain unchanged. In other words, they have to have the same terminal description in the medium and machine communication. This decomposition can be seen as a parameterization of the reactor vessel and it is also discussed in [Nilsson, 1989a], [Nilsson, 1989b], and [Nilsson, 1990].

The interesting details of the medium and machine decomposition are illustrated in Figure 5.2. The balance equations of volume, chemical composition, thermal energy, and mechanical energy are described in the vessel machine model. The inflow and outflow components in these equations are expressed by terminal variables using the dot notation, e. g., `In.q`. The heat transferred to the surrounding is described in the `Ht` terminal, as discussed in Chapter 4. The reaction velocity and the produced reaction heat are a medium and machine subterminal. The medium and machine terminal, `MMC`, is connected to the medium model in the composite vessel subunit model. Inside the medium model the reaction velocity is described and in this case it is a first order chemical reaction with

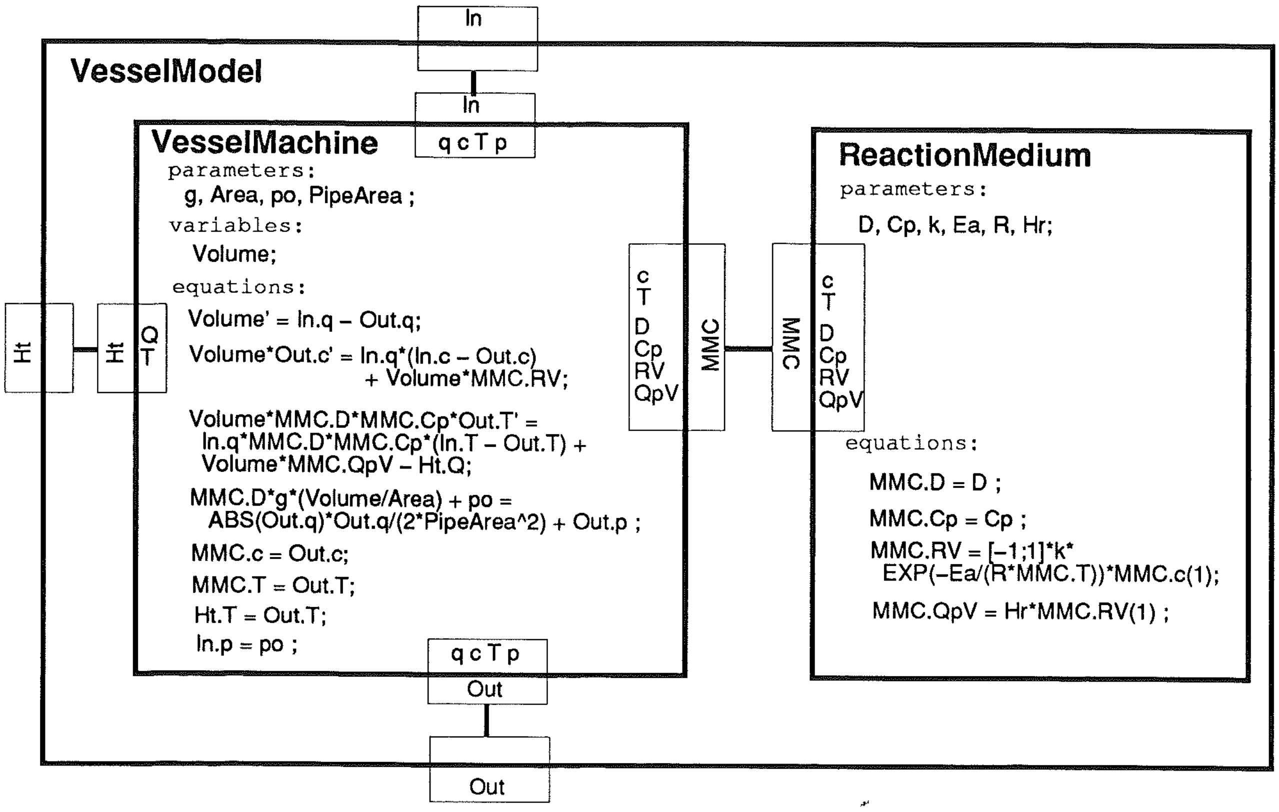


Figure 5.2 The function oriented static medium model based decomposition of the reactor vessel.

an Arrhenius temperature dependency. The medium model describes the reaction velocity as a function of the states which are described in the machine model. In this example these are the concentration, c, and the temperature, T. The pressure should also be defined in MMC but it is not used in this simple case. The information in the medium and machine terminal goes in both directions. The medium state is expressed in the processing machine and this information goes to the medium model which describes a number of medium state dependent properties. The medium model is like a function with no internal state and therefore this is called a *static medium based decomposition.*

Dynamic Medium Based Decomposition

The medium model describes the medium behavior as functions of the medium state in the static medium based decomposition. This way of decomposition is similar to the medium models in traditional flowsheeting, where the medium models are functions calculating physical properties.

Another way of making a medium and machine decomposition is to separate every medium dependent part from the machine dependent part. Both the concentrations and energy content in a reactor are quantities of the medium, e. g., medium state, and are described per mass unit or mole unit. They are intensive variables. Therefore the component and

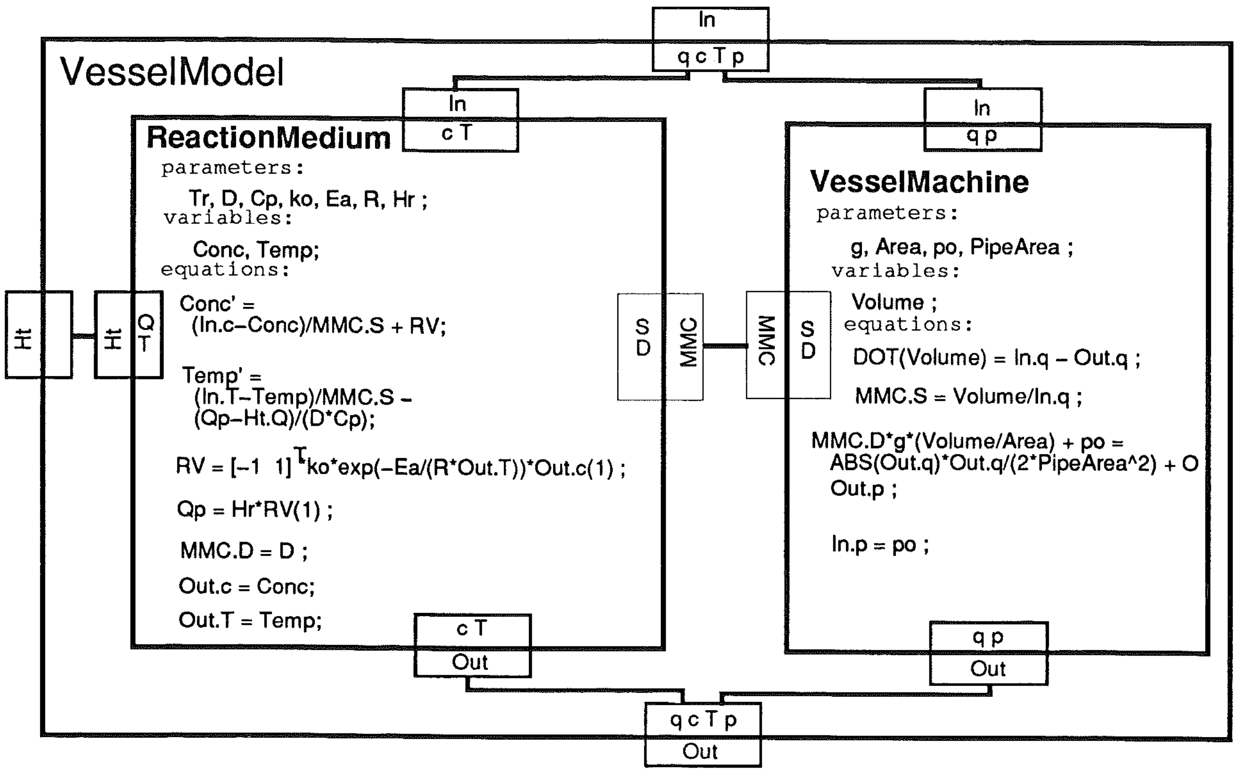


Figure 5.3 The dynamic medium model based decomposition of the reactor vessel.

enthalpy balances should be described in a medium model and not in a machine model. The amount of mass in the reactor is a machine dependent description. It is an extensive variable. The same goes for fluid mechanic descriptions.

This type of decomposition is shown in Figure 5.3. The medium and machine communication is minimized. In this case, under the assumption that the density and heat capacity are constant, the interaction variables are the space time, S, and the density. Space time is the volume divided by the flow. This example is perhaps a conceptually more correct medium and machine decomposition than the function oriented static medium based decomposition described in Figure 5.2. The separation of the in and out flow in one machine dependent part and one medium dependent part is however less natural and harder to interpret.

Polymorphic Medium Models

The decomposition of unit machine models and medium models become really powerful if the medium models can be made polymorphic. This allows changes of the medium model without changes in the machine model, according to the discussion in the previous chapter. Inherited attributes can be modified by overwriting in order to create new models. If the new submodel has the same interface as the original one and has the same degree of freedom, then the new model can be used in the same structure.

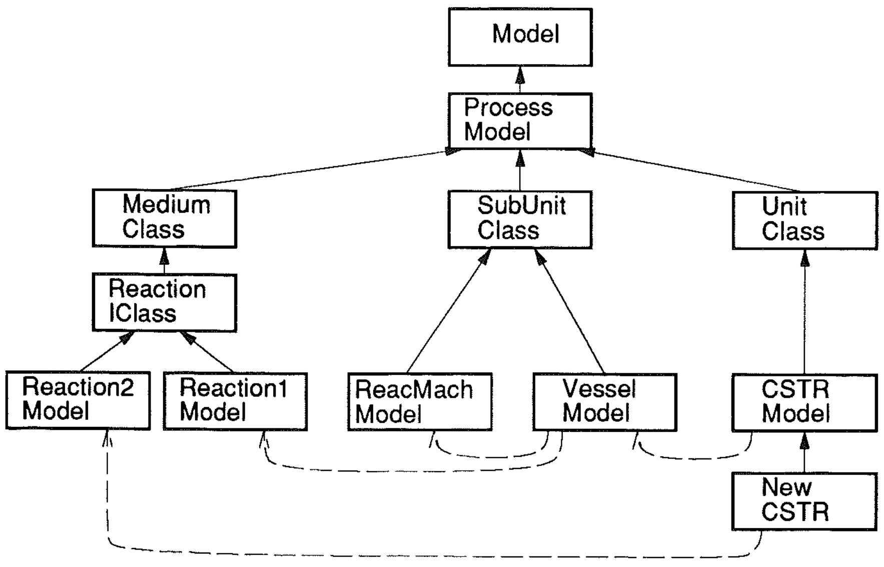


Figure 5.4 A definition of a new tank reactor with a new reaction model. The new reactor is a subclass of the old tank reactor with a new medium model super class.

Polymorphism is discussed in more detail in Chapter 4. This is the idea behind the medium and machine decomposition.

The medium model class can be modified by changing the super class definition to a new medium super class name. This is illustrated in Figure 5.4. It is now easy to reuse different medium models from a medium library. The use of the medium models in other types of machines, like non ideal reactors and tubular reactors, is also possible.

5.2 Medium and Machine Inheritance

The structure decomposition discussed above is artificial and is not a natural decomposition of a physical object. An alternative method is to view the vessel as composed of the properties from a medium description and from a unit machine decomposition. The reactor vessel is a typical example where multiple inheritance may be attractive. Multiple inheritance makes it possible to inherit properties from more then one super class. A suggestion of how multiple inheritance can look in Omola is listed in 5.1 where the super classes are given in a list in the class definition header. All attributes from the super classes are inherited and local specialization is possible using the overwriting rule. This is a natural extension which

```
Vessel ISA VesselMachine AND Medium WITH
  {class body}
END;
```

Listing 5.1 A suggestion of how multiple inheritance in Omola may look like.

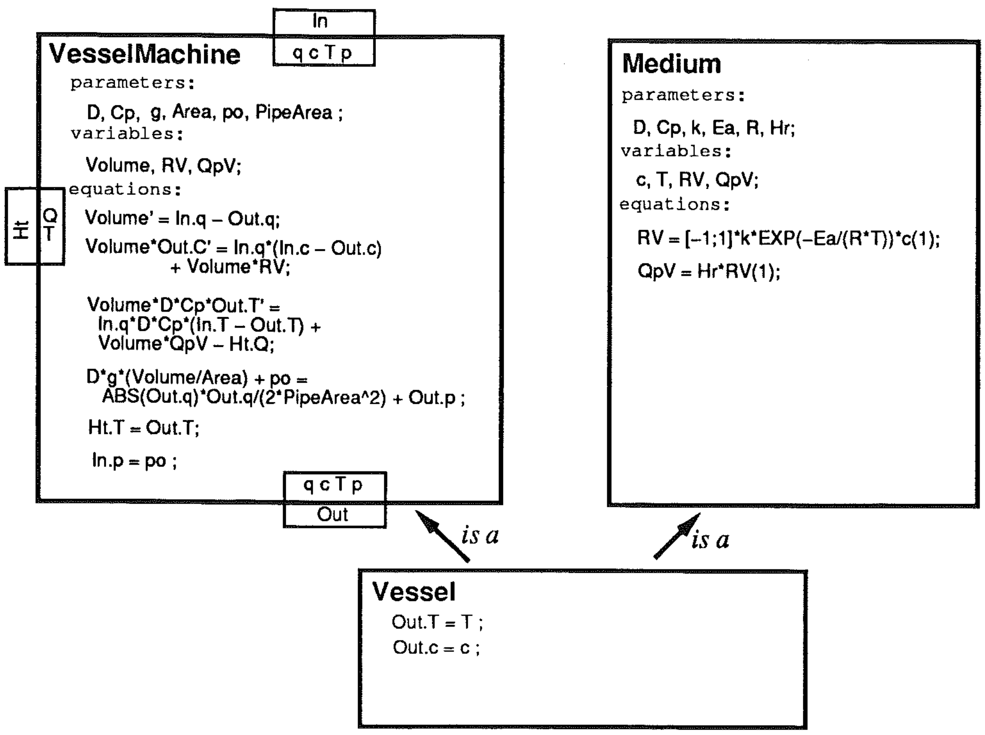


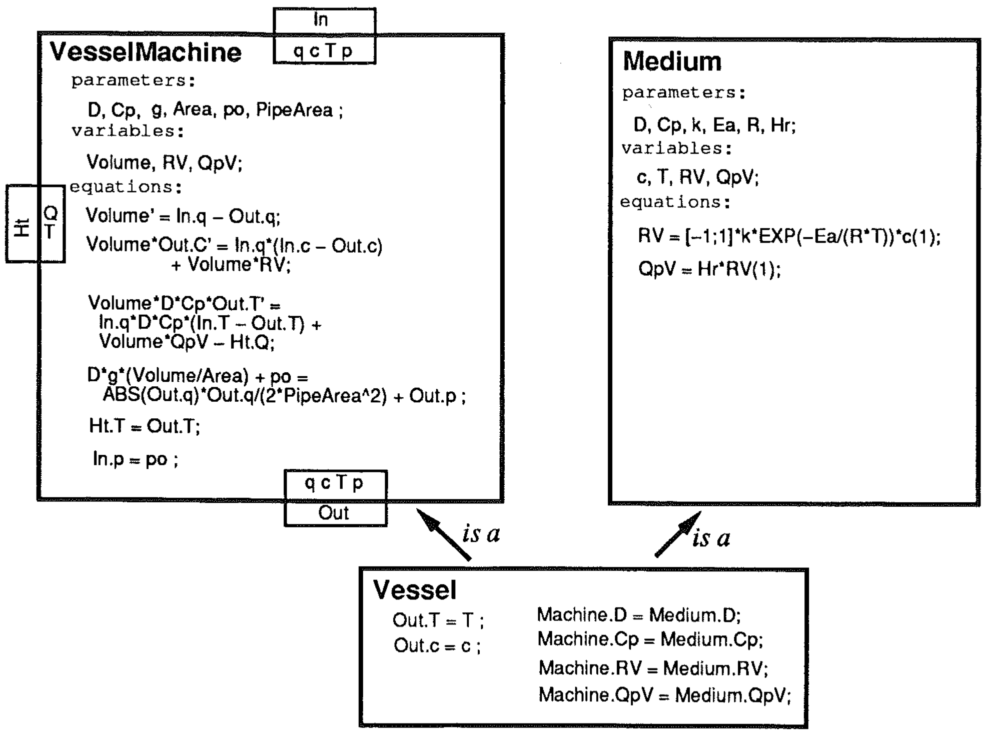
Figure 5.5 Multiple inheritance with implicit interaction using overwriting in the reactor vessel example.

is straightforward as long as the attributes do not overwrite each other, i. e., the inherited attributes are "orthogonal".

Assume static medium model based decomposition, as discussed in the previous section and assume also a machine model class and a medium model class as described in Figure 5.5. Instead of a reactor vessel decomposition and terminal connections a machine class and a medium class are used as super classes in a reactor vessel model class description. In the structure decomposition version of medium and machine models the communication is explicitly described in the MMC terminals. In the multiple inheritance version this interaction must also be described. It can be implicit or explicit.

Multiple Inheritance with Implicit Interaction

Implicit interaction uses the overwriting rule and a common implicit name convention. The inherited attributes might have the same name in multiple inheritance. Therefore a priority rule describing which way inherited attributes overwrite each other would be needed. Assume that the vessel machine class overwrites the medium class. The medium class has a variable called RV describing the reaction velocity. This is overwritten by the corresponding definition with the same name in the vessel machine class. This means that the super classes must know of each other in order to obey the overwriting rule. The concentration definitions do not have the same name in the super classes. This means that an additional equation



```
VesselMachine
parameters:
  D, Cp, g, Area, po, PipeArea ;
variables:
  Volume, RV, QpV;
equations:
  Volume' = In.q - Out.q;
  Volume*Out.C' = In.q*(In.c - Out.c)
            + Volume*RV;
  Volume*D*Cp*Out.T' =
   In.q*D*Cp*(In.T - Out.T) +
   Volume*QpV - Ht.Q;
  D*g*(Volume/Area) + po =
    ABS(Out.q)*Out.q/(2*PipeArea^2) + Out.p ;
  Ht.T = Out.T;
  In.p = po ;

Medium
parameters:
  D, Cp, k, Ea, R, Hr;
variables:
  c, T, RV, QpV;
equations:
  RV = [-1;1]*k*EXP(-Ea/(R*T))*c(1);
  QpV = Hr*RV(1);

Vessel
  Out.T = T ;     Machine.D = Medium.D;
  Out.c = c ;     Machine.Cp = Medium.Cp;
                  Machine.RV = Medium.RV;
                  Machine.QpV = Medium.QpV;
```

Figure 5.6 Multiple inheritance without overwriting used in the reactor vessel example.

would have to be added to make the variable association, as shown in the vessel class in Figure 5.5. Developing models using multiple inheritance with overwriting would mean a careful use of name conventions.

Multiple Inheritance with Explicit Interaction

The way to solve the last problem with the naming conflict is to abandon the overwriting rule between inherited attributes. The result in the reactor vessel example is that there will be two RV variables, see Figure 5.6. The communication, describing that they are equal, must therefore be added as a local attribute in the new class as shown in the Figure 5.6.

This solution is very similar to the decomposition approach. It creates a lot of description overhead compared to an implicit interaction approach. An advantage, however, is the possibility to describe the interaction in the subclass instead of distributing it in two super classes through an informal name convention.

Summary

The medium and machine decomposition can be made in a number of ways. The structure decomposition, discussed in the previous section, results in a composite model with encapsulated submodels using explicit connections for interaction description. The well defined interfaces facilitate the control of polymorphism of medium models. Neither of the approaches using multiple inheritance have this property. Structure decomposition is used in the rest of the thesis because of this and also because it can be directly implemented in the current version of Omola. The static medium model based decomposition is also preferred. This choice is based on traditional reasons and not on experience.

5.3 Primitive Behavior Decomposition

The behavior of a primitive model is described by a number of equations. These equations represents particular aspects of the model behavior, e. g., conservation of mass. The equations have a given form, e. g., a dynamic mass balance. An interesting idea is to have equation or behavior objects in the model database. These equation objects can be reused in a new model in order to create a new behavior. It could also increase the readability of the model due to the abstraction of the behavior into a number of objects instead of a chunk of equations. These ideas are discussed in [Stephanopoulos *et al.*, 1987], [Sørlie, 1990], [Stephanopoulos *et al.*, 1990a] and [Lund, 1992].

A process unit model behavior can be decomposed into equation objects describing orthogonal quantities, like conservation of mass and energy. These equation objects can be represented in two different ways, as equation submodels or as inherited equation super classes. It can be described by structure decomposition or by multiple inheritance. This question is closely related to the previous medium and machine decomposition problem. A discussion based on three different ways of equation reuse follow below:

- The encapsulated equation decomposition approach describing the equation objects as submodels, interacting through connections,
- the open equation decomposition using common variables instead of explicit connections for interaction descriptions, and
- multiple inheritance of equation super classes.

Some common problems and their solutions are also discussed.

Encapsulated Equation Decomposition

One way of representing equation objects is as submodels in a composite model. The key problem here is whether the equation objects are encapsulated modules or not. If the objects are represented as in a composite model then they are encapsulated. This means that relations between equations must be explicit and must be described by connections. Every variable that is used more than locally, in one equation object, must be defined in the interface.

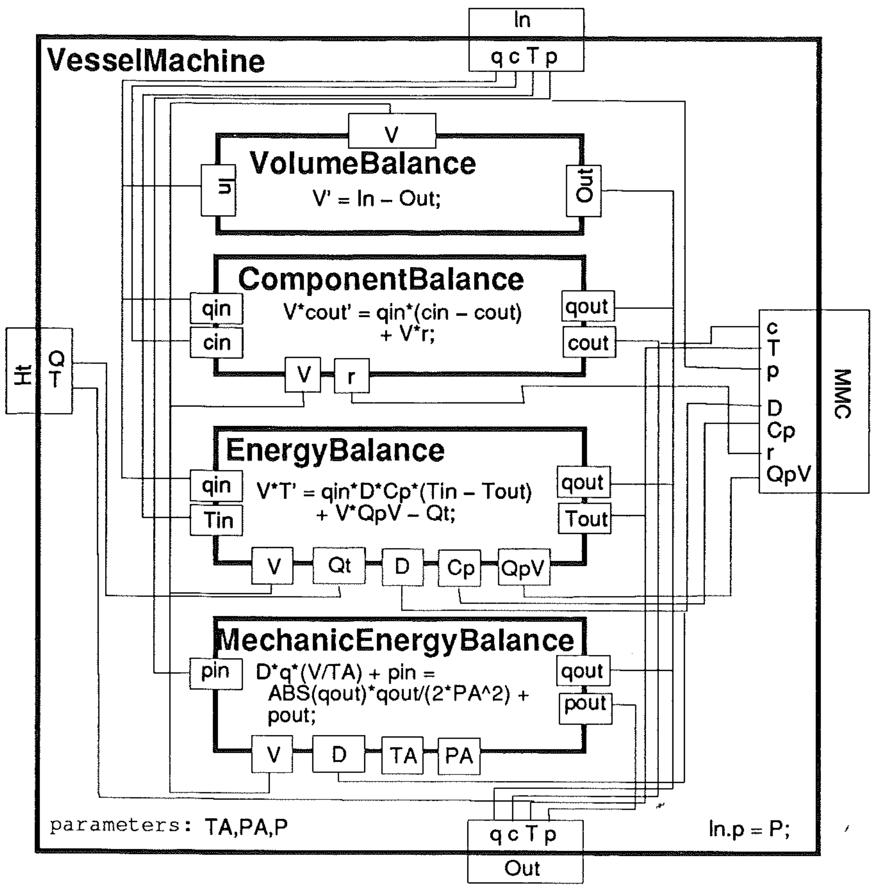


Figure 5.7 The primitive behavior of the tank reactor is turned into a composite model with encapsulated equation submodels.

The tank reactor behavior can be decomposed into a number of equations, as discussed in Section 5.1. Let us use the structure decomposition mechanisms to make the vessel machine into a composite model with equation submodels. The equation submodels are subclasses of global equation model classes. In this case the interactions between equations must be explicitly defined by connections. It also means that common variables are duplicated as simple terminals in every submodel. This is illustrated in Figure 5.7. The interaction structure is complex and this is not a good modularization. Perhaps a variable transformation can decrease the object interaction, but the problem with almost no abstraction of the interaction is still not solved.

Open Equation Decomposition

Another natural way is to allow open equation objects. This means that variables do not have to be defined inside the equation objects. However this also implies that the objects are incomplete, creating another problem. Omola cannot parse incomplete classes into the model database. Another problem is the name convention that must be used in all equation objects. The common variables referred to in the different equation objects must have the same names. This name convention is informal and not explicitly

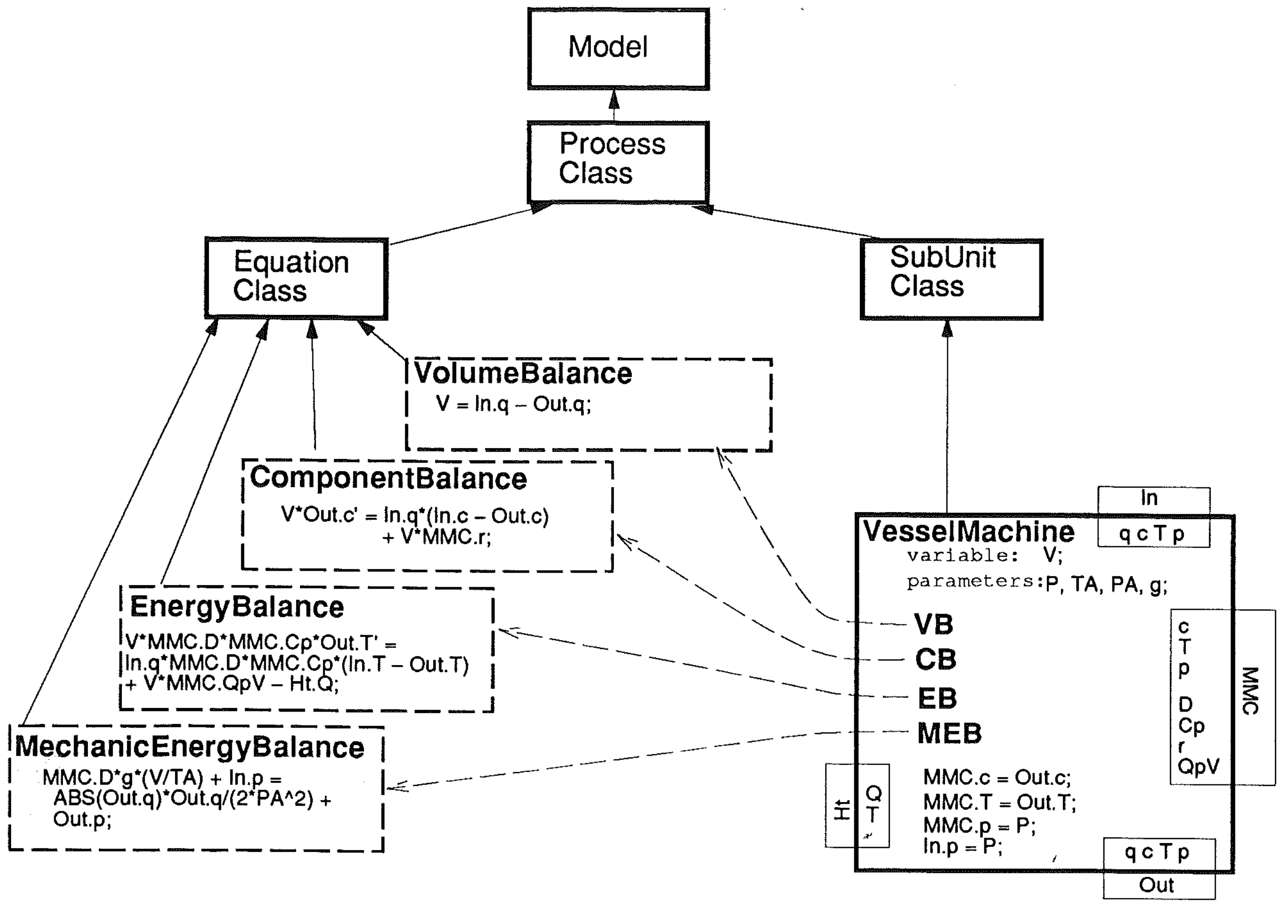


Figure 5.8 The primitive behavior of the tank reactor is turned into a composite model with open equation objects.

described until they all appear in a complete model with a given set of common variables. Instead of encapsulated equation submodels with thick walls, we have open objects where the variables are allowed to be defined outside the equation object.

Open equation classes are illustrated in Figure 5.8. Compared to Figure 5.7 the connections disappear because all equations refer to the same variables. The walls of the equation objects are transparent. The equations refer to variables according to the scope rules. The price that is paid for this is that the global equation super classes in the database are not consistent. They must be used in a given context with a specific name convention. To make this use of globally defined open equation classes meaningful one has to use an informal name convention which all equations obey. Otherwise this use of equation objects can be dangerous. This way of abstracting the primitive model behavior decreases the complexity. The possibility to give an equation a name and an equation type makes it easy to interpret the model description. Today Omola represents equations as nameless attributes which therefore cannot be overwritten. It is possible to encapsulate equations into equation objects but this can only be done locally inside a given context and it is therefore unhelpful for reuse.

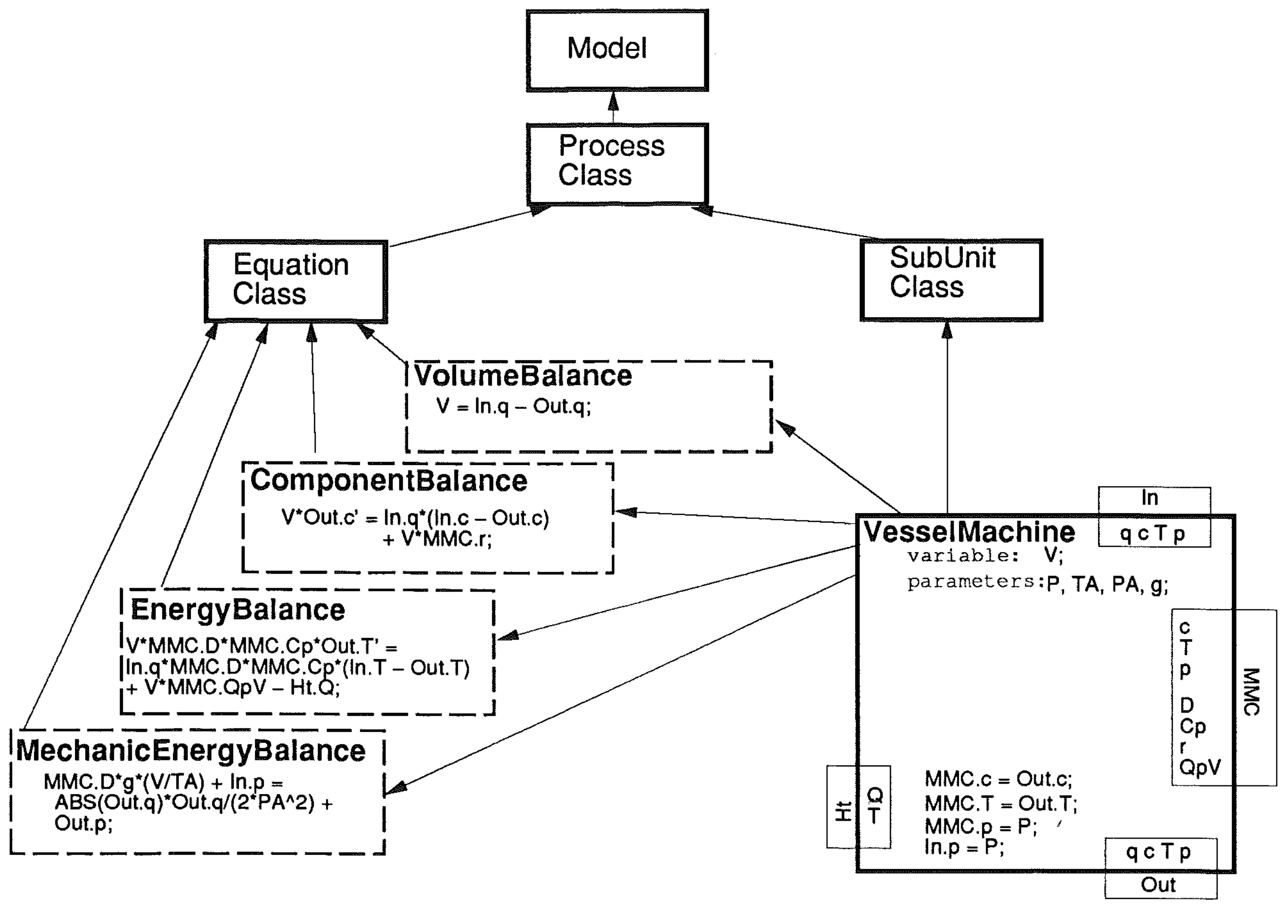


Figure 5.9 The use of multiple inheritance in the behavior definition in the tank reactor example. Notice that the global equation objects have undefined variables in their equations.

Multiple Inheritance of Equation Objects

Behavior can also be seen as something that is inherited from more general behavior classes. A behavior is composed of a number of different behavior components which means that single inheritance is not enough. Single inheritance means that all combinations of behavior components must be predefined and a large tree of all possible super classes created beforehand. This is a not a good solution. Multiple inheritance makes it possible to inherit a number of behavior components at one level in the class tree.

Yet another version of the reactor vessel machine class is seen in Figure 5.9. Assume that there are globally defined equation objects in the database. The vessel machine inherits through multiple inheritance a number of equations which become nameless attributes of the machine. We still have the problem with encapsulated or open global equation classes. The problem with multiple inheritance overwriting, discussed in the medium and machine decomposition in Section 5.1, must also be taken care of in this case. Since the use of open equation classes requires an informal name convention in any case, we have to solve the problem. The inheritance of a number of encapsulated equations into one class is exactly the same problem as Figure 5.5 and 5.6. If overwriting is possible then an informal name convention is important. Without overwriting, every inherited equation class has its own variable names. The model user has to make explicit declarations of the variable interactions.

Abstract Classes

An open equation object cannot be parsed into the Omola model database. Omola will complain about an inconsistent object with undefined variables. One way to handle inconsistent classes is to use a new type of attribute definition. It should be possible to declare an attribute as abstract which means that it should not be instantiated. An abstract attribute can be inherited and used in consistency analysis. An abstract attribute definition defines a name and a super class. A class containing abstract definitions must be used in a context where all names are defined as ordinary attributes. When the model is instantiated the local abstract attributes are not instantiated and the Omola scoping rules must find another variable with the same name in the model hierarchy. The variables with the same name are checked to be of the same type in order to define a consistent model.

A name convention is explicitly described by making these abstract

```
EquationClass ISA ProcessClass WITH
% Not correct Omola
variables:
  CLASS mass ISA Variable;
  CLASS In   ISA InFlowTerminal;
  CLASS Out  ISA OutFlowTerminal;
END;

MassBalance ISA EquationClass WITH
equation:
  DOT(mass) = In.q - Out.q;
END;

Tank ISA Model WITH
variables:
  mass ISA Variable;
  In   ISA InFlowTerminal;
  Out  ISA OutFlowTerminal;
equation_objects:
  MB ISA MassBalance;
END;
```

Listing 5.2 One suggestion of how to create open equation classes. It is based on the definition of dynamic scoping of variables using abstract class definitions.

variable class definitions in the common super class for all equation objects. This is done in EquationClass in Figure 5.8. The idea is illustrated in Listing 5.2. An abstract class is defined with the prefix CLASS. A set of abstract classes is defined in the EquationClass. The subclass MassBalance inherits these definitions which are referred to in the local equation. The mass balance object is the super class of the MB attribute in the tank class. The equation inside the mass balance object MB will refer to the attributes in the tank during the model compilation. This solves both the informal name convention problem and the inconsistency problem.

Parameterized Classes

An alternative to abstract classes is the concept of parameterized classes. The idea is to move the local abstract class definition up to the head of the class definition. Attribute definitions in the class head are not instantiated. It is similar to the macro concept in ordinary simulation languages, like ACSL [Mitchell and Gauthier, 1986]. The macro concept is discussed thoroughly in [Cellier, 1991].

Listing 5.3 illustrates the concept of parameterized classes. The class, MassBalance, has three class definitions in the class header. The class definitions are referred to in the equation in the normal way. Submodel MB of Tank class is defined with an argument list which refers to the variables in the Tank which have other names.

Parameterized classes create the possibility of having local names in classes. However the class header becomes more complicated. Notice that

```
MassBalance (mass ISA Variable,
             In ISA InFlowTerminal,
             Out ISA OutFlowTerminal)
  ISA EquationClass WITH
% Not correct Omola
equation:
  DOT(mass) = In.q - Out.q;
END;

Tank ISA Model WITH
variables:
  M        ISA Variable;
  InFlow   ISA InFlowTerminal;
  OutFlow  ISA OutFlowTerminal;
equation:
  MB(M,InFlow,OutFlow) ISA MassBalance;
END;
```

Listing 5.3 A second suggestion of how to create open equation classes. It is based on parameterized classes.

the equation object becomes encapsulated with a well defined interface via the argument list in the class header.

Discussion

Two related problems has been discussed. The first problem is encapsulated versus open equation classes and the second is multiple inheritance of equations versus equation component attributes. The problems have the common difficulty with informal name convention.

Encapsulated equations are hard to use and require many connections due to the explicit relations between variables. The only practical way of using encapsulated equations is together with multiple inheritance with overwriting. Then one has to solve the name convention problem and one solution is suggested using parameterized classes.

The open equation class concept also has problems with inconsistent classes with undefined variables. It also requires an informal name convention which is common for all equation classes. Compared to the interaction description problem in the encapsulated equation example the open equation class concept is superior.

The difference between decomposition and inheritance is more a matter of taste. A modeling language should support both. In the problem with equation reuse the decomposition is superior to the use of multiple inheritance. The benefit is the abstraction of the behavior into local objects with super classes. This is contrary to some of the result found in the literature [Stephanopoulos *et al.*, 1987], [Stephanopoulos *et al.*, 1990a], and [Lund, 1992].

An alternative to the discussion in this chapter is to use graphics in order to make primitive behavior descriptions. Two examples are the analog computer description and the bond graph approach. Analog computer descriptions of equations is common in control engineering. Here electrical components are used to construct a circuit with the same properties as the equation in question. The model developer describes the equation using predefined objects like integrators, adders, multipliers etc. SIMULINK [MathWorks, 1991] is based on this approach. This is a nice way of describing relations between variables but for a process engineer it is not a practical way of describing equations.

Bond graph descriptions are another example of graphic based behavior descriptions, see [Cellier, 1991] or [Ljung and Glad, 1991]. A bond graph describes the energy flow in the system and it is based on a few modeling objects. Bond graphs Expressing behavior with graphics is interesting and Bond graphs can probably be implemented in Omola without any problems.

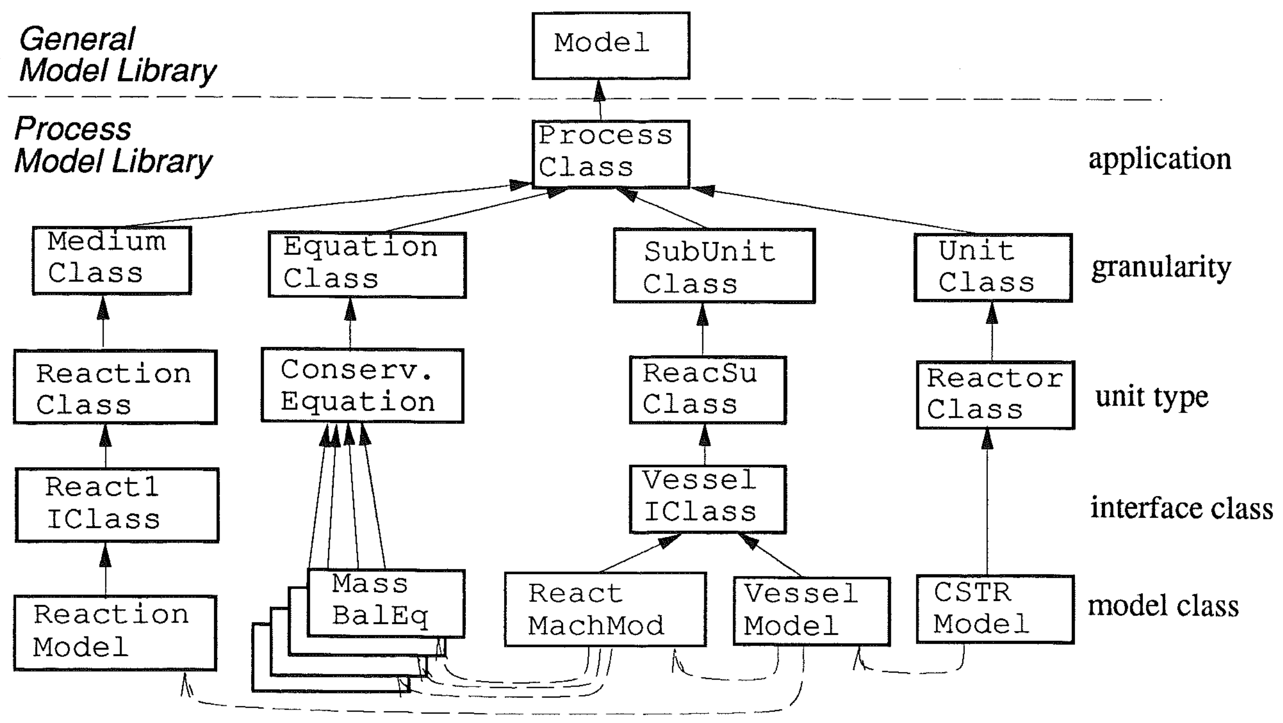


Figure 5.10 The class inheritance guideline is complemented by two new granularity class, `MediumClass` and `EquationClass`.

5.4 Conclusions

The decomposition of processing unit behaviors is discussed in this chapter. The traditional medium and unit decomposition can be achieved by coordinated decomposition and single inheritance or by multiple inheritance alone. The structure decomposition approach is found to be superior with its explicit described interaction. One of the major benefits is the ability to create polymorphic medium models. The static medium based decomposition was chosen because of its simplicity. This results in a medium and machine decomposition that can be implemented in Omola.

In the discussion about behavior decomposition into equation objects the encapsulated model concept was found to be too rigid. An open class concept is suggested where models are allowed to have noninstantiated attributes. Open classes can therefore not be instantiated and can only be used for inheritance in a given context. Also in this problem the idea of equation object attributes is superior to the use of multiple inheritance of equation super classes.

The structure hierarchy guidelines are extended by a new level below the subunit. *Equation objects* can be used to build up new subunits. Also the class hierarchy guideline can be commented. The super class of all equation objects is `EquationClass` which is a new class on the granularity level. The super class to all medium models is also a new class on this level. Figure 5.10 illustrate the two new granularity classes that are discussed in this chapter.

6

Parameterization

Parameterization is an important method for abstraction of internal complexity. Different parameterization methods are discussed in this chapter. The methods are exemplified in distillation column examples. Distillation is probably the most common unit operation in the chemical process industry. The behavior description of the distillation example is based on first principle physics. The structure of the unit and the demand on reuse create interesting modeling problems. Decomposition, inheritance, and parameterization are used to solve these problems. The aim is to create basic submodels that are reusable and abstracted to a user oriented view. In some situations the Omola language cannot handle the problems and in these cases suggestions of Omola extensions are presented.

6.1 Demands on Parameterization

A parameter is something that can be changed by the user in order to adapt the model behavior to a new application. In Omola a parameter is a time invariant variable that can be changed between simulations and treated as a constant by the the simulator. These two statements about a parameter are very different. The Omola interpretation of a parameter is a hard restriction of the first statement. The first definition of a parameter could be almost anything, as will be discussed in this chapter.

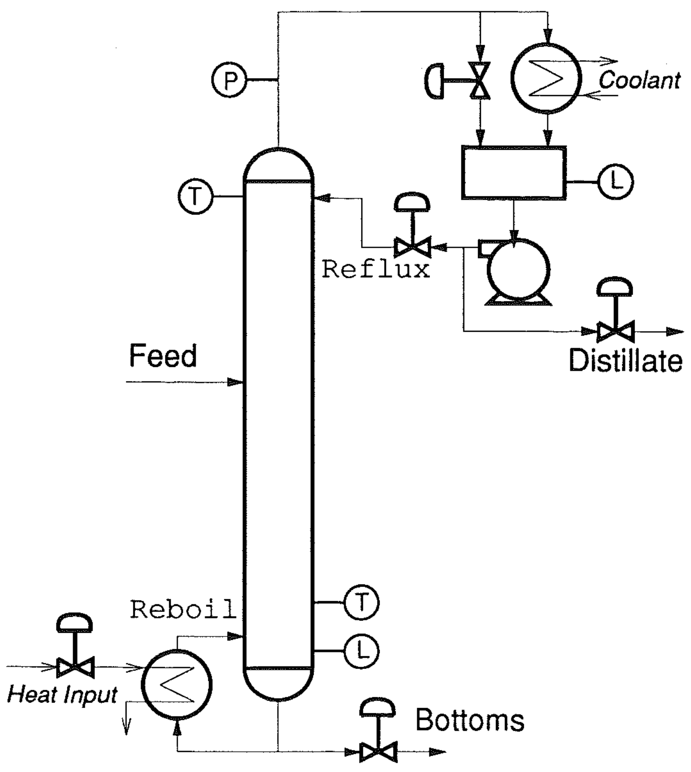


Figure 6.1 The basic structure of a distillation unit.

The Distillation Unit

Distillation is an unit operation in the process industry that uses energy to split a stream into two streams with different chemical compositions. This separation is done in a distillation tower where heated vapor and cooled liquid are forced to interact. The liquid is boiled in the bottom of the column to create a rising vapor flow and the vapor at the top is condensed to create a liquid stream falling downwards. The feed enters the column somewhere in the middle and the product streams are often taken at the top and in the bottom. More complicated columns can have multiple feeds and multiple product streams, so called side streams.

The basic unit is a structure of physical objects, like reboiler, column, condenser, reflux drum, pump, valves and sensors and it is illustrated in Figure 6.1. Some of the physical objects have an internal structure of objects similar to the decomposition in the previous chapter. The tray based column has an internal structure of trays. The number of trays can vary from a few to hundreds. The trays are connected in a regular structure. Vapor from the tray below and liquid from the tray above enter the tray and give the tray a certain content of liquid and vapor. Vapor leaves the tray to the tray above and the liquid to the tray below, see Figure 6.2. Distillation columns are often used in series or distillation trains. Each unit configuration often differs slightly. For instance the column can be tray based or packed based, the condenser can be a total condenser or a partial condenser, the reboiler can have different configurations etc. This creates a family of distillation component objects that can be put together to create a certain distillation unit.

The user of a distillation unit model has to specify a number of parameters characterizing the unit. The unit and medium should be defined separately. The number of trays and the feed tray number should be parameters. It should also be easy to change between different reboiler and condenser configurations. See the ASPEN PLUS example in Appendix A.

6.2 Medium and Tray Parameterization

Liquid and vapor are forced to interact in a tray. This phenomenon is quite complex and can be modeled at different levels of detail.

A Tray Model

A common way to model trays is to assume that the dynamics in the vapor phase is fast compared to the liquid phase. This means that pressure, vapor flow, and vapor dynamics in the tray are neglected. The model discussed in more detail in this section is based on component mass and energy conservation in the liquid only. In Figure 6.2 the principle of the vapor and liquid contact in tray is illustrated. The medium and machine decomposition discussed in Chapter 5 becomes even more important and powerful in this application. This is because of the column structure which is discussed further in Section 6.4. this chapter. The tray model terminals are two liquid flow and two vapor flow terminals describing the flows through the tray.

An Omola code description is found in Listing 6.1. It is a composite model with one tray machine model and one distillation medium model. They are connected to each other and to the tray model terminals. The terminals in the tray model are subclasses of record terminals discussed in Chapter 4, see the Listing 4.1 and 4.2. Note that the chemical dimension of the tray submodels and terminals is assigned a value that comes from the medium model.

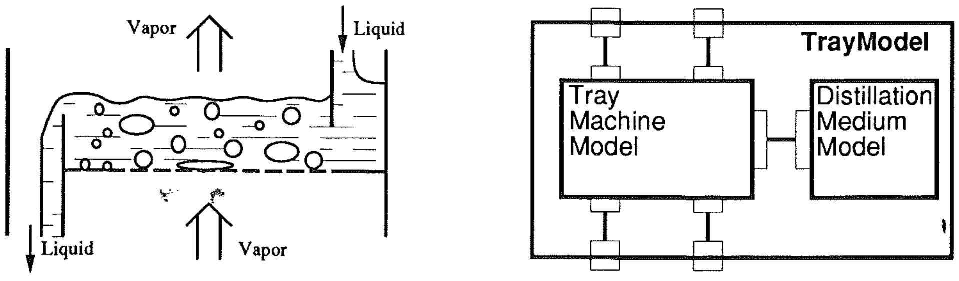


Figure 6.2 A tray in a distillation column and the conceptual decomposition into medium and machine models.

```
TrayModel ISA Model WITH
structure_parameter:
  ChemDim TYPE Integer := Medium.NumberOfComponents;
  Machine.ChemDim      := Medium.NumberOfComponents;
terminals:
  LIn  ISA LiquidInTerminal  WITH NoComp:=ChemDim; END;
  VIn  ISA VaporInTerminal   WITH NoComp:=ChemDim; END;
  LOut ISA LiquidOutTerminal WITH NoComp:=ChemDim; END;
  VOut ISA VaporOutTerminal  WITH NoComp:=ChemDim; END;
submodels:
  Machine ISA TrayMachineModel;
  Medium  ISA DistillationMediumModel;
connections:
  LIn AT Machine.LIn;
  VIn AT Machine.VIn;
  Machine.LOut AT LOut;
  Machine.VOut AT VOut;
  Machine.MMC AT Medium.MMC;
END;
```

Listing 6.1 The tray model is composed of one tray machine model and one distillation medium model.

Tray Machine Parameterization

The tray machine model is composed of the differential equations describing the mass and energy dynamics, as seen in Listing 6.2. It has terminals, parameters, variables, and equations. There are 14 equations in all. The first one is a dynamic component mole balance in vector form. Its dimension is equal to the `ChemDim` structure parameter. Notice that all composition subterminals in the flow terminals are also vectors with the length specified by this `ChemDim` attribute. The chemical dimension of the machine model is parameterized by the `ChemDim` parameter.

The second and third equations describe the total amount of a chemical in mole units and the height of the liquid level of the tray. The fourth equation is a dynamic energy balance. Since it has been assumed that vapor dynamics can be neglected, energy is modeled as the liquid enthalpy. The following three equations describe the liquid outflow of the tray in terms of mole flow, composition, and energy. After the outflow equations the pressure on the tray is calculated as a simple subtraction of the pressure drop from the pressure of the tray below. The last five equations are the medium and machine interaction. The liquid composition, liquid enthalpy, and pressure are all described in the machine model. The relation between these variables and the composition and enthalpy in the vapor and the liquid density and mole weight are medium specific and thus described in the medium model.

```
TrayMachineModel ISA Model WITH
structure_parameter:
  ChemDim TYPE Integer;
terminals:
  MMC  ISA MediumMachineCommunication WITH NoComp := ChemDim; END;
  LIn  ISA LiquidInTerminal           WITH NoComp := ChemDim; END;
  LOut ISA LiquidOutTerminal          WITH NoComp := ChemDim; END;
  VIn  ISA VaporInTerminal            WITH NoComp := ChemDim; END;
  VOut ISA VaporOutTerminal           WITH NoComp := ChemDim; END;
parameters:
  TrayArea, WeirLenght, WeirHeight, G, PressureDrop ISA Parameter;
variables:
  Xmole ISA ColumnVariable WITH n := ChemDim; END;
  mole, level, energy ISA Variable;
equations:
  %% component mole balances
  Xmole' =
    LIn.Flow*LIn.Composition   + VIn.Flow*VIn.Composition -
    LOut.Flow*LOut.Composition - VOut.Flow*VOut.Composition;
  mole = SUM(Xmole);
  level = mole*MMC.MoleWeight / (MMC.Density*TrayArea);
  %% energy balance
  energy' =
    LIn.Flow*LIn.Enthalpy   + VIn.Flow*VIn.Enthalpy -
    LOut.Flow*LOut.Enthalpy - VOut.Flow*VOut.Enthalpy;
  %% out flow models
  LOut.Flow = IF height<WeirHeight THEN 0 ELSE
    WeirLenght/1.5*SQRT(2*G*(height-WeirHeight)^3);
  LOut.Composition = Xmole*(1/mole);
  LOut.Enthalpy = energy/(mole*MMC.MoleWeight);
  %% pressure drop
  VOut.Pressure = VIn.Pressure - PressureDrop;
  LOut.Pressure = VOut.Pressure;
  %% medium model communications
  MMC.LiquidComposition = LOut.Composition;
  MMC.VaporComposition  = VOut.Composition;
  MMC.LiquidEnthalpy    = LOut.Enthalpy;
  MMC.VaporEnthalpy     = VOut.Enthalpy;
  MMC.Pressure          = VOut.Pressure;
END;
```

Listing 6.2 The machine part in the tray model.

Distillation Medium Parameterization

The distillation medium model is seen in Listing 6.3. The structure parameter NumberOfComponents, used in TrayModel in Listing 6.1, is assigned its value here. The actual medium behavior is one phase equilibrium model, two enthalpy descriptions, and density and mole weight

```
DistillationMediumModel ISA MediumClass WITH
structure_parameter:
  NumberOfComponents TYPE Integer := 2;
terminal:
  MMC ISA MediumMachineCommunication WITH
    NoComp:=NumberOfComponents;
  END;
parameters:
  A, B, LiquidCp1, VaporCp0, VaporCp1, CompDensity,
  CompMoleWeight ISA RowParameter WITH
    n := NumberOfComponents;
  END;
variables:
  LOGPartPressure ISA ColumnVariable WITH
    n := NumberOfComponents;
  END;
  temperature ISA Variable;
equations:
  %%phase equalibrium
  LOGPartPressure = B - A*(1/temperature);
  MMC.VaporComposition = EXP(LOGPartPressure) *
    (1/MMC.Pressure) ./ MMC.LiquidComposition;
  SUMABS(MMC.VaporComposition) = 1;
 %%enthalpy
  MMC.LiquidEnthalpy = (LiquidCp1*temperature) *
    MMC.LiquidComposition;
  MMC.VaporEnthalpy = (VaporCp0 +
    VaporCp1*temperature)*MMC.VaporComposition;
  %%density
  MMC.Density     = CompDensity * MMC.LiquidComposition;
  MMC.MoleWeight = CompMoleWeight * MMC.LiquidComposition;
END;
```

Listing 6.3 The distillation medium part in the tray model.

descriptions. The phase equilibrium model describes the relation between the composition in the vapor and in the liquid, the temperature and the pressure. The enthalpy descriptions, one for the liquid and one for the vapor, relate the enthalpy to the composition and temperature. The last two equations describe the density and mole weight relations with the composition respectively.

Medium and Machine Communication

The medium and machine communication class is found in Listing 6.4. It contains information that goes in both directions. Liquid composition, liquid enthalpy, and tray pressure are all described in the machine model. The vapor composition and enthalpy are calculated inside the medium model. The liquid density is just a function of the liquid composition.

```
MediumMachineCommunication ISA RecordTerminal WITH
  NoComp TYPE Integer;
  LiquidComposition ISA SimpleTerminal WITH
    value   TYPE column[NoComp];
  END;
  VaporComposition ISA LiquidComposition;
  LiquidEnthalpy ISA SimpleTerminal;
  VaporEnthalpy  ISA LiquidEnthalpy;
  Pressure   ISA SimpleTerminal;
  Density    ISA SimpleTerminal;
  MoleWeight ISA SimpleTerminal;
END;
```

Listing 6.4 The medium and machine communication terminal in the tray model.

Reuse of the tray model

A typical way to reuse the tray model is to use it for a new application with a different medium. Assume that the same model assumptions made before are valid in the new application. An example of a new medium model that inherits the medium model described in Listing 6.3 would look like the one in Listing 6.5. A new tray model definition is seen with the new distillation medium model as the super class of the medium model. The only thing that has to be changed is the definition of the medium super model. This is done by overwriting the original definition with a new one. Notice that, in the new medium model, the number of components is set to three. The chemical dimension in the tray machine, compositions in terminals, and in composition state vector is set by the ChemDim parameter. The chemical dimension in the machine is set equal to the number of components in the medium model in the composite tray model. This means that the chemical dimension of the whole tray model is set by one parameter in the medium model. This is not an ordinary parameter because if the parameter is changed the model has to be recompiled. It is called a *structure parameter* because it changes the structure of the model and not the behavior as ordinary parameters do. The parameters are distributed in the model structure. The distributed parameters are assigned values through parameter equations.

Comments on the Mathematical Model

The tray model is described in Listings 6.1 to 6.4. The machine model consists of mass balances and one energy balance. The medium model describes the vapor composition in the form of an equilibrium model defining the vapor composition as a function of liquid composition and tempera-

```
DistMediumModel2 ISA DistillationMediumModel WITH
structure_parameter:
  NumberOfComponents TYPE Integer := 3;
parameters:
  A               := [0.00637,0.0073,0.0157];
  B               := [32.4,31.1,46.0];
  CpLiquid        := [2.51,1.67,1.67];
  Cp0Vapour       := [559,523,600];
  Cp1Vapour       := [1.67,1.26,1.26];
  CompDensity     := [640,1120,1440];
  CompMoleWeight  := [58,45,18];
END;


NewTrayModel ISA TrayModel WITH
  Medium ISA DistMediumModel2;
END;
```

Listing 6.5 A subclass of the medium model seen in Listing 6.3. It is specialized with a chemical dimension of three and associated vectors for the physical parameters. After this a new tray model is defined which uses this medium model.

ture, under the constraint that the vapor composition elements must sum to one. The vapor composition constraint results in an algebraic equation which means that the vapor composition and the temperature must be solved simultaneously. This is a differential-algebraic problem of index one. The temperature is then used in the enthalpy calculations. The energy balance has therefore only one unknown which is not the derivative; the unknown is instead the vapor outflow. This model is found in Luyben [Luyben, 1973]. It is common to model a tray at equilibrium like this, and these models are common in process engineering. However, it makes the dynamic models unnecessarily difficult, causing index problems, [Ponton and Gawthrop, 1991]. An alternative is to assume constant vapor flow through the tray and remove the energy balance. It is hidden in the equilibrium description. Rate equation based descriptions are a third way where the evaporation is described as a function of a driving force. These are, on the other hand, difficult to validate.

6.3 Column Parameterization

The column in the distillation unit contains a structure of trays on top of each other. To conveniently describe this structure requires mechanisms for structure generation and for parameterization of this structure.

Regular Structures

A regular structure is defined as a structure of identical components. In a regular structure it is more convenient to refer to the individual components by their place in the structure rather than by unique component names. This problem is similar to the need for matrix descriptions in algebra.

A first approach to create a regular structure mechanism in Omola is to introduce indexed named models. Then it is possible to operate on each individual component using its index. There is also a need for a connection concept for regular structures. In a regular structure there is a lot of identical connections. The definition of these connections should be done once. Below follows some example of how these mechanisms can look in Omola.

EXAMPLE 6.1—Explicit iterator based notation

```
% Not correct Omola
FOR i=1 TO 3 CREATE
  Tray[i] ISA TrayModel;
  Tray[i].Pressure = InPressure - PressureDrop*i;
END;
FOR i=1 TO 2 CREATE
  Tray[i].VOut AT Tray[i+1].VIn;
  Tray[i].LIn  AT Tray[i+1].LOut;
END;
```

The need for a regular structure is discussed in [Nilsson, 1987]. This example of a regular structure concept with FOR-loops is suggested in [Nilsson, 1989a]. It is closely related to the suggestion in [Elmqvist, 1978] and is also implemented in ASCEND [Piela, 1989]. □

EXAMPLE 6.2—Implicit iterator based notation

A second approach is influenced by the Maple and Matlab notation for matrices. Instead of using loop structures one can use local loops in one statement. The notation : (colon) has the meaning of a local FOR-loop.

```
% Not correct Omola
Tray[1:3] ISA TrayModel;
Tray[1:2].VOut AT Tray[2:3].VIn;
Tray[1:2].LIn  AT Tray[2:3].LOut;

Tray[i=1:3].Pressure = InPressure - PressureDrop*i;
```

Notice that it is sometimes necessary to use the index as a parameter in the equation, which means that the name of the index variable must be explicitly declared. □

Omola is a declarative language which means that a FOR-loop construction is alien. On the other hand, the FOR-loop construction discussed here can be viewed as a preprocessor for automatic generation of a structure rather than a sequential execution procedure. The second notation is the most compact and well suited for this purpose. In the future in this chapter the second notation is assumed to be implemented. In the current version of OmSim there is no regular structure mechanism.

Column Parameterization

An example of a structure parameter is the number of trays in a distillation column. Assume that the regular structure mechanisms discussed above has been implemented, see Example 6.2. In the example in Listing 6.6 the definition of submodels, connections, and parameter assignments are all parameterized by `NoTrays`.

The only unusual aspect of `NoTrays` is that it cannot be changed during simulation studies as can other parameters. It may be viewed as a constant and therefore the value of the parameter is bound to a number and cannot be changed. Changes must be done in the column class before model compilation.

The relation between different parameters is described explicitly in Listing 6.6 and this relation is declared on a level in the model hierarchy where the parameters can be reached. For example the area parameter in all trays are assigned the same value as the `TrayArea` parameter of the column.

6.4 Distillation Unit Parameterization

The structures of different distillation units are often similar but they always have some unique component. The basic unit is composed of a column, a reboiler configuration, and a condenser configuration. The reboiler configuration is a set of physical objects for creating a rising vapor flow and a bottom product flow. The condenser configuration is similar but with objects for the generation of reflux flow and top product flow, i. e., the distillate. This is seen in Figure 6.1.

The distillation column unit is a complex configuration of objects. The model structure and class hierarchies become deep with five to ten different levels. The structuring problem is discussed in the previous sections but it also creates a parameterization and abstraction problem. The aim is to create reusable models on each level. Down in the model structure hierarchy it is often easy to find out how models look and how they can be reused. Further up in the model hierarchy one must know about the internal structure of the submodels to make proper modifications.

```
ColumnModel ISA Model WITH
structure_parameters:
  NoTrays    TYPE Integer;
  FeedTrayNo TYPE Integer;
parameters:
  TopPressure, BottomPressure, TrayArea, WeirLength, WeirHeight
    ISA Parameter;
terminals:
  LiquidIn  ISA LiquidInTerminal;
  LiquidOut ISA LiquidOutTerminal;
  VapourIn  ISA VapourInTerminal;
  VapourOut ISA VapourOutTerminal;
submodels:
  Tray[1:FeedTrayNo-1]       ISA TrayModel;
  Tray[FeedTrayNo]           ISA FeedTrayModel;
  Tray[FeedTrayNo+1:NoTrays] ISA TrayModel;
connections:
  Tray[2:NoTrays].LiquidOut   AT Tray[1:NoTrays-1].LiquidIn;
  Tray[1:NoTrays-1].VapourOut AT Tray[2:NoTrays].VapourIn;
  LiquidIn  AT Tray[NoTrays].LiquidIn;
  VapourOut AT Tray[NoTrays].VapourOut;
  LiquidOut AT Tray[1].LiquidOut;
  VapourIn  AT Tray[1].VapourIn;
parameter_equations:
  Tray[1:NoTrays].PressureDrop:=
    (BottomPressure-TopPressure)/NoTrays;
  Tray[1:NoTrays].TrayArea    := TrayArea;
  Tray[1:NoTrays].WeirLenght  := WeirLength;
  Tray[1:NoTrays].WeirHeight  := WeirHeight;
END;

Column25Trays ISA Column WITH
structure_parameters:
  NoTrays    := 25;
  FeedTrayNo := 16;
parameters:
  TopPressure.default    := 10000;
  BottomPressure.default := 11250;
END;
```

Listing 6.6 The column model assuming a regular structure mechanism becomes compact both for the structure description and the parameter assignments.

Assume that an unit has the right configuration of components. Then a typical distillation column unit design include following: medium specifications, number of trays, feed entering tray, tray and column dimensions and heating and cooling data. Assignments of these parameters are now discussed.

Super Model Parameter Assignment

Assigning values to parameters can be done in the class definition, as a constant or as a default value, or it can be done interactively during simulation studies. In the distillation column example, parameters deep down in the model structure hierarchy must be given values. In the attempt to create reusable models, abstraction is important. Abstraction facilitates reuse of models and makes it possible for the user to forget about the internal details. Basic and commonly changed parameters of a composite model should be found on the top level.

A straightforward problem is to assign values to heating and cooling parameters in the reboiler and condenser. It is possible to bound parameters to other parameters in a model in Omola. This is done by *parameter assignment*, which describes a relation between parameters. Parameters are local and defined in submodels. They are reached by the dot notation. If the top level, the distillation unit, has a parameter called `HeatArea` then it can be used to assign the value of the submodel parameter in the structure hierarchy through a parameter assignment. `Area` is a parameter in the heat transfer model which is a submodel of the reboiler model. It is interesting to lift the assignment of this particular parameter up in the hierarchy to the unit operation model interface. This is done by the use of the dot notation in a parameter assignment.

```
Reboiler.HeatTransfer.Area := HeatArea;
```

This parameter assignment is declared inside the distillation unit model. A more complicated problem is to assign values to the trays in the regular structure. A new column design may have hundreds of trays and the user does not want to set all these parameters by hand. Omola has no mechanism for repeated assignments, as discussed above. The problem is similar to the regular structure connection problem.

```
% Not correct Omola
Column.Tray[1:NoTrays].Machine.TrayArea := ColumnDiameter^2*PHI/4;
```

The area parameter in all trays is given a value through this parameter assignment which is calculated through an expression using a super model parameter, the column diameter. This is nice because of the similarity to the regular structure mechanism that was discussed in the previous section.

Important parameters in distillation unit design are the number of trays and the feed tray number. As discussed in the column parameterization section, these parameters are used to generate the structure. They are structure parameters. These parameters are also lifted up to the top level by the use of parameter equations.

Submodel Parameterization

Each tray in the column is composed of one machine model and one medium model, see the tray model class in Figure 6.3. By overwriting the inherited medium class attribute with a new medium class a totally different column is created. As mentioned in Section 6.2 the only restriction is a well defined medium and machine communication. The trays in the column form a regular structure model which in turn is a submodel in the distillation unit. If the user now wants to change the medium model super class the user must know about the internal structure of the unit and change the medium class definition on the right level in the model hierarchy. This means not only in the column but also in the reboiler and condenser. This is not user friendly. A well defined distillation unit model has a medium model parameter in its interface, i. e., a medium parameter on the top level in the structure hierarchy. It can be solved by two different methods, also mentioned in the previous chapter, abstract classes and parameterized classes.

Abstract Classes

An *abstract class* is a class definition which can be used for inheritance but cannot be instantiated, as discussed in end of Chapter 5. The abstract class is defined by a prefix CLASS. An abstract class definition at the top level of a structure hierarchy can be a super class for internal classes. By the redefinition of the top level abstract class all the internal subclass are redefined. This idea can be used to simplify the medium parameterization problem in a large structure as a distillation unit. The idea is illustrated in Figure 6.3. A global tray model class is defined as discussed in the beginning of this chapter. It has a medium model that is a subclass of a global medium model in a medium model library. A number of trays are defined as subclasses of the composite tray class with the difference that their medium model super class is redefined. The new medium model super class for all the trays are instead an abstract class on the column level. The globally defined tray based column model is inherited in a distillation unit which also has an abstract medium model class definition. The inherited column class is specialized by the change of its abstract class definition. The local column in the unit has an abstract class which is a subclass of the unit defined abstract class. This will result in all internal medium model classes in the whole distillation unit being defined as subclasses of only one abstract class. A change of the medium classes of all internal components is done by the redefinition of the super class to the unit abstract class. This idea is also discussed in [Nilsson, 1992].

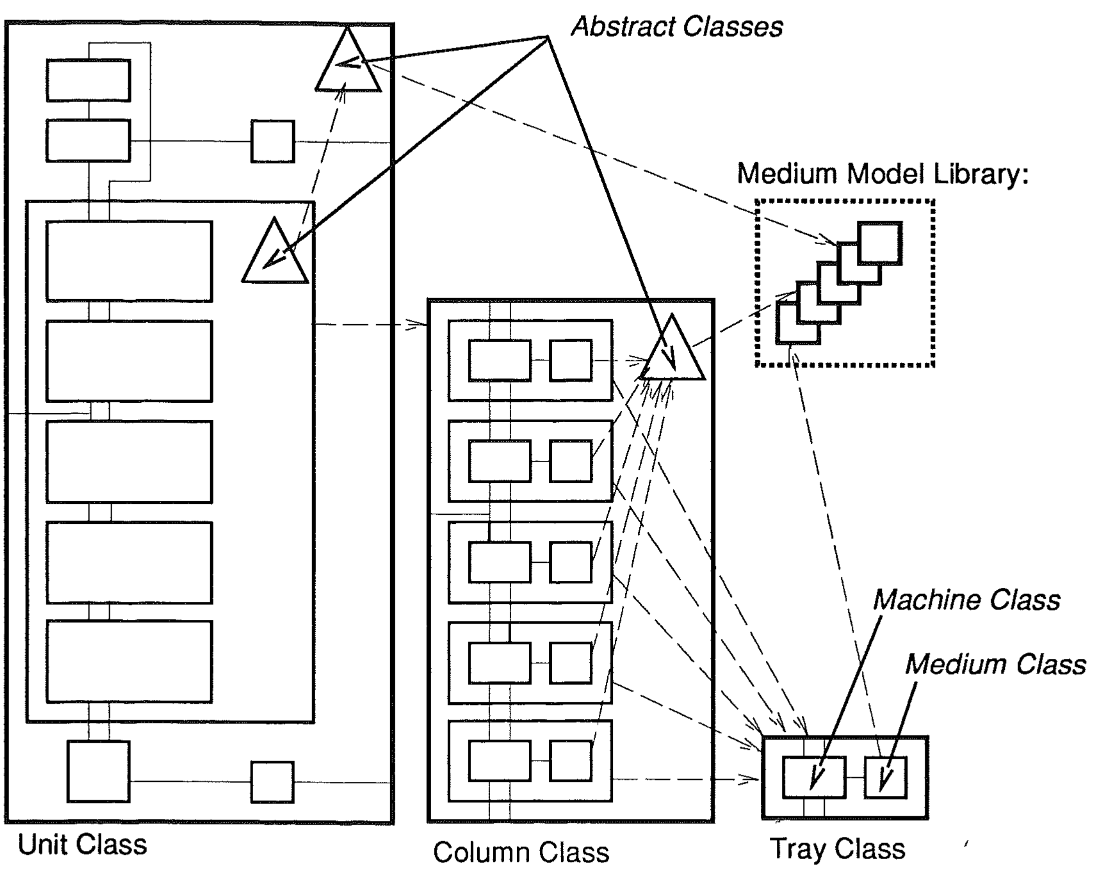


Figure 6.3 The parameterization method using abstract classes. The internal medium model classes are subclasses to the an abstract class on the top level in the model.

Listing 6.7 illustrates how it can look in Omola. A column model is defined similar to the one discussed in the previous section. The major difference is the abstract class definition of a medium model. The tray medium model definition is overwritten by a definition that points to this local abstract medium class. The OUTER prefix indicates that the class is found in the structure hierarchy inside the composite model and not in the global model database. This medium class definition in the column has the prefix CLASS which means that it is abstract and not instantiated and it can only be used for local inheritance.

```
ColumnModel ISA Model WITH
  % Not correct Omola
  CLASS MediumModel ISA DistMedium8;
  :
  Tray[1:NoTrays] ISA TrayModel WITH
    MediumModel ISA OUTER::MediumModel;
  END;
  :
END;
```

Listing 6.7 A suggestion of how abstract classes can be used for inheritance of a local class definition.

Actually this idea of local classes for inheritance inside the structure hierarchy can be used without abstract classes. Assume that the medium model in the distillation unit discussed here is a static description. This means that the local unconnected medium classes for reuse are instantiated, but the model compilation will analyze these equations and find them static. Therefore will they only be executed once.

Parameterized Classes

Another way to solve the same problem is to extend the use of parameterized classes. Parameterized classes were discussed in Section 5.3. It makes it possible to have local names in a class and refer to the same variables by other names by the user. It is done by the definition of a class interface to the owner. An extension of this concept is to allow redefinition of the entities in the class header.

One example of this concept is illustrated in Listing 6.8. A tray model is defined with the medium model class definition in the class argument list. The following column model class has a tray subclass as a component. The tray model, T1, has a new argument list where the super class of the argument is redefined to a class in the column model argument. On the unit model level a local column model, CM, is defined with a new medium model class. This makes it possible to lift up the definition of the medium model to the unit top level.

```
TrayModel(MM ISA MediumModel) ISA Model WITH
  % Not correct Omola
  :
  TM ISA MachineModel;
  :
  MM.MMC AT TM.MMC;
  :
END;

ColumnModel(DM ISA MediumModel) ISA Model WITH
  :
  T1(MM ISA DM) ISA TrayModel;
  :
END;

UnitModel ISA Model WITH
  :
  CM(DM ISA DistMedium8) ISA ColumnModel;
  :
END;
```

Listing 6.8 A suggestion of how parameterized classes can be used for super class redefinition.

```
DistillationUnit ISA Model WITH
  % Not correct Omola
  UnitSelect TYPE (Unit1,Unit2) := 'Unit1;
  CASE UnitSelect OF
  Unit1:
    MediumModel ISA EtOHWaterModel;
    NoTrays := 12;
  Unit2:
    MediumModel ISA ButanToluenModel;
    NoTrays := 62;
    FeedTrayNo := 47;
  END;
END;
```

Listing 6.9 A suggestion of as CASE construction in Omola.

Predefined Parameter Alternatives

Often a selection of parameters can be grouped together and assigned by a meta parameter. In the distillation unit example, this can be a choice of a special kind of unit. This can be achieved in two ways. First it can be done by a straightforward use of inheritance, i. e., by setting all the parameters and creating a class of the unit in question. This may result in a large number of specific designed classes which makes the model database more difficult to organize. An alternative way is to make it possible to define different parameter setups which the user selects from. This can be done by a CASE construction, which ASCEND has [Piela, 1989], and a suggestion is listed in Listing 6.9. The UnitSelect variable can have one of the two discrete values, Unit1 or Unit2. The case statement selects one of the two different definitions depending on the value on UnitSelect. Notice that this case statement assigns values to structure parameters and that it is executed before model compilation. It is, like the FOR loop construction discussed earlier in this chapter, more like a preprocessor for structure generation than a sequential execution statement.

6.5 Conclusions

This chapter focuses on the parameterization problem of a typical chemical unit operation, the distillation column. It is shown that there is a need for a regular structure mechanism. There is also a need for a number of different parameterization methods. The parameters can be of different kinds. Ordinary parameters can be changed between simulations. Structure parameters cannot be changed after model compilation. The structure parameter must be assigned values on the class level before model compilation. A directly reused submodel class can be seen as a special kind of structure parameter which can be changed by the redefinition of its super class:

- *Super model parameter assignment* is used to lift up a parameter deep down in the structure hierarchy and assign it to a value of the top level. It is done by a local parameter of the top level and a parameter assignment describing the relation between the parameters.
- *Parameter propagation* is used to propagate a parameter through a connection from one submodel to another. This is not valid for structure parameters.
- *Submodel parameterization* can be done by removing one submodel and replace it with a new one. Overwriting a well defined submodel with a new super class definition is a compact way to change submodels. A special version is the use of local abstract classes for local inheritance.

7

Control Systems

Dynamic studies of industrial processes include both dynamic models of the actual process and dynamic models of the control systems. Dynamic studies have not been so common in the process industry. They have mainly been used in process control construction and in operation studies and seldom in process design. This is changing and dynamic studies are on their way to becoming an important process design tool.

The traditional design of processes is based on static model descriptions used in static simulation and optimization. Dynamic problems have been taken care of by control engineers in the design of the control system. The dynamic problems that the control system cannot handle have been taken care of by the process operator. Examples are drastic operation changes and failure situations. This means that dynamic properties of the plant are studied after the plant has been build.

The increasing demands on process plants require dynamic considerations already in the design phase of the plant construction. The design must take account for flexible production, failure situations etc. Process design computer environments will therefore include dynamic simulation tools in the future, see [Evans, 1990], [Vogel, 1991] and [Wozny *et al.*, 1992].

The process control systems of tomorrow are going to handle many other operation problems than today, like diagnosis, on-line optimization, planning etc, see [Årzén, 1992]. To develop complex control systems like this requires tools for verification before the control system is connected to the plant. This means that the control systems in the future will also include process modeling facilities and dynamic (real time) simulation tools. It means that developed control system models can be implemented in a real time environment. This real time control system can then be used to control a real time simulation of the process model.

This chapter discusses some of the problems in modeling process control systems. The primitives and the mathematical formulation of continuous and sampled controllers are discussed in Section 7.1 and event based controllers in Section 7.2. The structure and representation of process control systems are discussed in Section 7.3. In Section 7.4 a hierarchical control system concept is suggested followed in Section 7.5 with suggestions of class hierarchy guidelines.

7.1 Continuous and Sampled Controllers

A conventional process control system can be divided into two parts, continuous and sequential control. The continuous control system can be implemented by analog controllers or by digital controllers in a computer. The sequential control system contains sequences and logic and is implemented in Programmable Logical Controllers, PLCs, or computers.

To handle discrete controllers and sequences Omola has primitives for discrete events. Events can be time dependent and scheduled to occur some time units in the future. The `Sample` event is scheduled to occur 1 time unit in the future.

```
Sample ISAN Event;
schedule(Sample,1.0);
```

Events can also be state dependent and occur when a state condition becomes true. An event can cause two kinds of actions. First, an event can generate new events, `CAUSE`, and second they can cause an execution of assignments. Assignments inside `DO` and `END` are executed when the event occurs.

```
y TYPE DISCRETE Real;
ONEVENT x>0.5 CAUSE Sample;
ONEVENT Sample DO
  new(y) := y + 1;
END;
```

Variables that are only assigned values in events, like the y variable above, are defined as `DISCRETE` variables. Their values remain valid until they are changed in a new event. The `new` operator refers to the variable

```
PIDcontrollerIClass ISA Model WITH
terminals:
  y,yr,uc ISA SimpleInput;
  u       ISA SimpleOutput;
parameters:
  K,Ti,Td ISA Parameter;
  b,Tt,N  ISA Parameter;
END;

PIDcontrollerModel ISA PIDcontrollerIClass WITH
variables:
  e,p,i,yf,d ISA Variable;
equations:
  e  = yr - y;
  p  = K*(b*yr - y);
  i' = K*e/Ti + (uc - u)/Tt;
  yf' = N/Td*(y - yf);
  d = -K*N*(y - yf)
  u = p + i + d;
END;
```

Listing 7.1 A simple PID controller divided into one interface class and one model class.

value after the event and the variable without operator refers to the value before the event. A variable that is defined as discrete real is ignored in the manipulation of the continuous differential equation system during model compilation.

Continuous Time Controllers

Continuous time controllers are described by dynamic systems, often low order finite dimensional linear systems. The classical PID controller, proportional, integral and derivative controller, is very common in the process industry. It is used in almost every control loop. It is common to use more than one PID controller in a given structure and a multi-PID controller functions more like a complex controller. Complex controllers are therefore built up by the use of PIDs as building blocks. Implementation of PID controllers is discussed in [Åström, 1987].

An example of a classical PID controller is seen in Listing 7.1. The controller description is decomposed into two classes, one interface class which is the super class for the actual controller model class. The controller interface class has four terminals, three inputs and one output, and six parameters. The internal behavior described in the model class can be described by six equations and four internal variables. The algorithm is a simple PID algorithm with tracking and filtering of the measurement.

```
PIDdiscreteModel ISA PIDcontrollerIClass WITH
parameter:
  h ISA Parameter;
variables:
  e,p,i,yf,d,v ISA Variable WITH
    value TYPE Discrete Real;
  END;
events:
  Init, Sample ISAN Event;
behavior:
  ONEVENT Init, Sample DO
    new(e) := yr - y;
    new(p) := K*(b*yr - y);
    new(i) := i + K*h/Ti*e + h/Tt*(uc-u);
    new(yf):= yf + h*N*Td*(y - yf);
    new(d) := -K*N*(y - new(yf));
    new(v) := new(p) + i + new(d);
    schedule(Sample,h);
  END;
equation:
  u = v;
END;
```

Listing 7.2 A simple discrete PID controller.

A continuous time controller is a continuous dynamic system and can be modeled by the same tools as discussed before in this thesis. Omola can also handle matrices which means that state space controllers are easy to describe.

Discrete Time Controllers

Discrete time controllers are described by difference equations with a sampling interval. They can be designed in two different ways, directly in discrete time or by discretization of analog controllers. The latter design can be done if the controller can sample fast enough.

A discrete PID controller can be a discretization of the continuous controller behavior, see [Åström, 1987]. This discretized behavior is executed in the action part of a sampling event. The discrete PID controller in Listing 7.2 is similar to the continuous one in Listing 7.1. The continuous and discrete PIDs are subclasses of the same interface class. Notice the extra parameter, `h`, the sampling period. The system defined event `Init` occurs at time zero. The assignments are calculated and a new event, `Sample`, is scheduled to occur after one sampling period. The variables in the controller, except for the three input terminals, are discrete and thus constant between the sampling events. Notice the use of the `new` operator.

7.2 Event Driven Controllers

Event driven or asynchronous controllers are other important parts in process control systems. These controllers describe sequences and logic which often are state dependent.

Sequential Controllers

Events can be used to describe sequences. One way to illustrate this is to describe how Grafcet can be implemented in Omola. Grafcet based sequences can be developed by graphics. A text book on Grafcet and Petri nets is [David and Alle, 1992] and Grafcet and Petri net implementations in Omola are discussed in [Nilsson, 1991].

The basic objects in Grafcet are steps and transitions. Steps represents the states of the sequence and they can be activated or deactivated. The activation is done by a transition which governs the change of state from one step to another step. A transition is fired if the step above the transition is active and the transition condition is fulfilled. Then the step above is deactivated and the step below is activated. Actions can be associated with a step and when a step is active the associated action is executed. An example of a simple Grafcet is illustrated in Figure 7.1. The Grafcet sequence is built up by objects which make it possible to use graphics. It is possible to implement Grafcet objects in Omola and one possible implementation is described in this section.

An Omola example of a transition is seen in Listing 7.3 and a step in Listing 7.4. If the condition becomes greater then zero and the step connected to Upper is active then the transition event is fired, causing two new events. These two events are propagated through the terminals to the connected steps. To make this possible there are two system predefined terminals describing event output and event input. If an event output in terminal T1 is connected to an event input in terminal T2 then

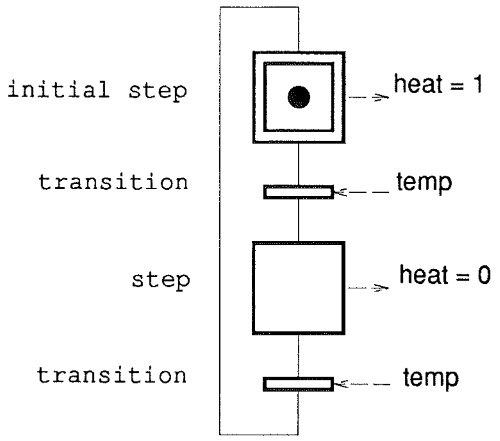


Figure 7.1 The On-Off controller modeled in Grafcet is illustrated by graphics.

```
TransitionModel ISA Model WITH
terminals:
  Upper ISA RecordTerminal WITH
    State ISA SimpleTerminal WITH
      value TYPE DISCRETE Integer;
    END;
    Trigg ISAN EventOutput;
  END;
  Lower ISA RecordTerminal WITH Trigg ISAN EventOutput; END;
  Condition ISA SimpleTerminal;
behavior:
  ONEVENT Condition > 0 AND Upper.State > 0.5 CAUSE
    Upper.Trigg, Lower.Trigg;
END;
```

Listing 7.3 A Grafcet transition in Omola. The condition causes the firing of two events which are propagated out from the transition object through the terminals.

this connection is interpreted as ONEVENT T1.Output CAUSE T2.Input. The transition causes event outputs which are connected to event inputs on the steps. The state is activated in the step if the upper event is fired and the state is deactivated if the lower event is fired. In the small example in Listing 7.5 the Grafcet objects, step and transition, are used to form a sequential controller, an on-off controller. This can be done by a graph-

```
StepModel ISA Model WITH
terminals:
  Upper ISA RecordTerminal WITH Trigg ISAN EventInput; END;
  Lower ISA RecordTerminal WITH
    State ISA SimpleTerminal WITH
      value TYPE DISCRETE Integer;
    END;
    Trigg ISAN EventInput;
  END;
  State ISA SimpleTerminal WITH
    value TYPE DISCRETE Integer;
  END;
event:
  Action ISAN Event;
behavior:
  ONEVENT Upper.Trigg CAUSE Action;
  ONEVENT Upper.Trigg DO new(State):=1; END;
  ONEVENT Lower.Trigg DO new(State):=0; END;
  Lower.State = State;
END;
```

Listing 7.4 A Grafcet step in Omola. Events propagated through the terminals enter the step and change the state.

```
GrafcetExample ISA Model WITH
terminals:
  temp ISA SimpleInput;
  heat ISA SimpleOutput WITH value TYPE DISCRETE Real; END;
parameters:
  Ref ISA Parameter;
submodels:
  S1 ISA InitStepModel WITH
    ONEVENT Action DO new(heat):=1; END;
  END;
  T1 ISA TransitionModel WITH Condition := temp - Ref; END;
  S2 ISA StepModel WITH
    ONEVENT Action DO new(heat):=0; END;
  END;
  T2 ISA TransitionModel WITH Condition := Ref - temp; END;
connections:
  S1.Lower AT T1.Upper;
  T1.Lower AT S2.Upper;
  S2.Lower AT T2.Upper;
  T2.Lower AT S1.Upper;
END;
```

Listing 7.5 A small example of Grafcet in Omola, an On-Off controller. A graphical description is found in Figure 7.1.

ical editor but the specific conditions and actions must be specified in a textual editor. Almost all event handling is encapsulated and abstracted in the transition and in the step super classes. The Grafcet development is abstracted to a graphical level. The Grafcet user must use the predefined `Action` event to cause actions and must also define the `Condition` expression to cause transition firing.

Other Types of Controllers

Describing integrated process control systems requires handling of other types of controllers. Examples of other types are expert systems, neural nets and fuzzy systems. To describe these control methods in Omola today they have to be turned into equations and events.

A rule based expert system is one common way to develop monitoring and diagnostic controllers. With Omola it is possible to use the event concept to describe the forward chaining mechanism in expert systems. One example is a blackboard type of rule based system. An object has terminals and variables and is also composed of a set of rule objects. The rule fires if the input terminals or variable values used in the condition expression changes. The rule action changes the value of the common variables or output terminals. This can cause other rules to fire. This can probably be described by equations and events. Model based expert

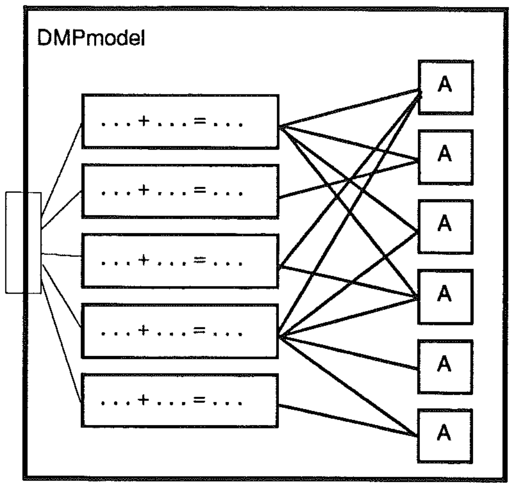


Figure 7.2 A small example of a DMP based diagnosis module.

systems are much more difficult to develop because they reason about objects and the state of objects and Omola does not have any problem solving tool with this capability.

Neural nets are based on quantitative measurements and nonlinear functions, described in a structure. This is well suited for Omola. The only difficult thing is to handle the complexity of the structure. There should be mechanisms for describing regular structures and index numbering objects in such structures.

Another approach, closely related to neural nets, is the diagnostic model processor, DMP diagnosis method, see [Petti and Dhurjati, 1991]. Here measurements enter model equations in residual form. If the equations predict the measurements well then the residuals are small but if not they become larger. The residuals are scaled in nonlinear functions. The equations are based on a number of model assumptions. If the equations are not predicting the measurements well then some of the assumptions are violated. All equations that depend on a particular assumption are used to confirm or reject this assumption. The scaled residuals are weighted to generate a failure likelihood of the assumption. When a failure likelihood is above a certain limit it causes an alarm. A diagnosis module like this is possible to develop in Omola.

A more advanced controller will incorporate tools for design and analysis. These tools can be complicated numerical calculations that are seldom used by the controller. Examples are automatic trajectory generation, automatic model based tuning and model validation from experimental data. They are algorithms with repeated entries and therefore difficult to translate into the event-assignment description in the current Omola. There is a need for procedural language concepts. This procedural language should be used to model computer calculations and should not be used to solve modeling problems. Examples of a procedure concept in a real time computer environment is found in G2, [Gensym, 1992].

7.3 Structuring Control Systems

Dynamic models with control systems for dynamics can be represented in many different ways. Here three approaches are discussed: *P&I diagrams, block diagrams,* and *unit oriented structuring.* The number of abstraction levels and class tree organization are also discussed.

Process and Instrument Diagrams

In the process industry, process and instrument diagrams, P&I diagrams, are the most common way of describing processes with control systems.

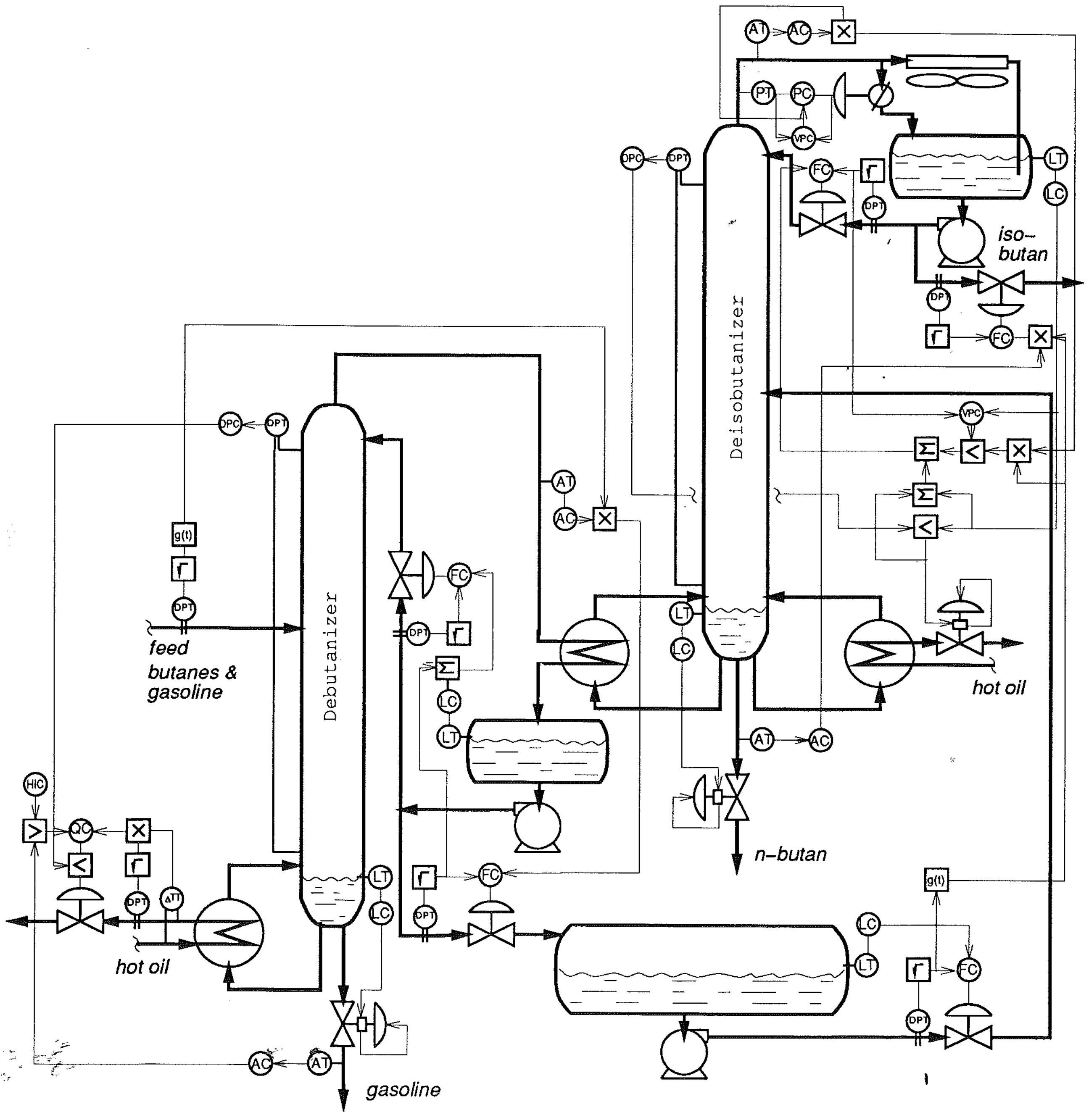


Figure 7.3 Two energy integrated distillation columns with control systems. From [Bristol, 1980].

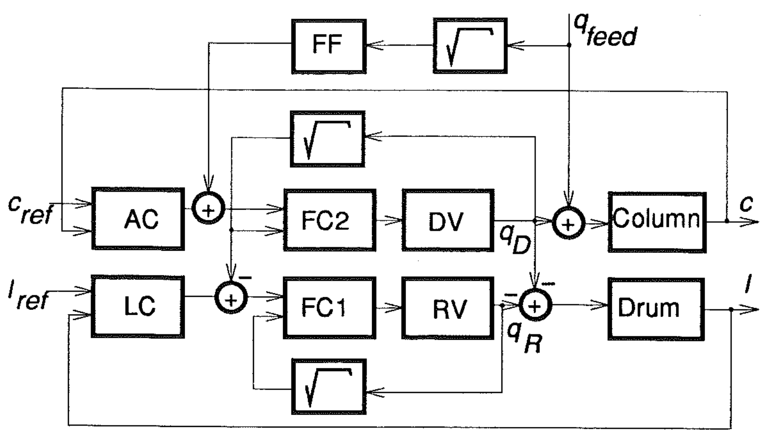


Figure 7.4 A typical block diagram of a part of a distillation column control system. It is the top control of the level in the reflux drum and the composition control in the distillate product. It is also seen in the top of the debutanizer in Figure 7.3.

The control system is described as a set of the continuous controllers and does not describe the logical and sequential controllers. In a process and instrument diagram, P&I diagram, the process objects and the control system objects are all described in the same topology.

The relations between the process units and the controllers become obvious. The control loop with sensor, controller and actuator is physically described. Problems arise when the controllers become more complex. Then it is not obvious how the control loop functions and what it is supposed to do. One good example of a P&I diagram that has this problem is seen in Figure 7.3. The control system is multivariable with ten actuators and twenty sensor measurements. The problem with this description is the lack of abstraction. The control system is distributed in the process description. There is no way to abstract the control loop details.

Control Block Diagram

The block diagram description is the most common way to describe structure in control engineering. A block diagram is similar to a P&I diagram. It is customary to use only blocks as icons and to have a given direction of the information flow through each blocks, i. e., from the left to the right. This makes the process description less intuitive and hard to understand even for a process engineer. The feedback loops are of course the most important details for a control engineer. A number of different process variables are merged and are described by a lumped disturbance variable.

Unit Oriented Control System

In some situations it is better to associate the control system with the unit. The control system becomes a part of the unit description. This is interesting when the internal control system is well known and can be reused from previous applications. Examples are dynamic process design and configuration of large control systems.

Process design that handles dynamic properties must also handle control systems. This means that the control system design and process design meet in the design phase. The process designer must know something about control systems and the control designer must know about the process design. To use the conventional design approach with unit operation decomposition of the process, the control system description must also support this decomposition. This results in an unit operation control system that makes it possible to reuse an unit, with a control system, from a library and simulate it. It should also be possible to parameterize the dynamic unit model with respect to the control system. Different unit designs need different control systems. Common combinations can therefore be predefined with simplified parameterization.

In configurations of large control systems like process control systems it is convenient to reuse well known configuration on the unit level. The process is illustrated by icons describing the process and the units include its control system. The unit control system must be flexible and support interunit control between different units, the coordination control on plant and plant section level.

The unit oriented control system can be represented in two major ways, *distributed* or *centralized*, see Figure 7.5. The distributed representation has a natural interpretation with well defined measurements and control actions. It is a P&I diagram on the unit level. The centralized control system is a composite model containing all control modules for the control of a unit. It interacts with the unit model through composite terminals describing all measurements and all control signals. The centralized version is harder to interpret in a quick look. On the other hand, if the internal structure of control system is not important, like in process design, then it is abstracted into one object. Two different representations of a control system are illustrated in Figure 7.5. The distributed control system with five controllers for controlling the tank inventory, level, and the product quality, temperature. The centralized control system is one process control system object which is composed of two objects, one inventory control and one quality control.

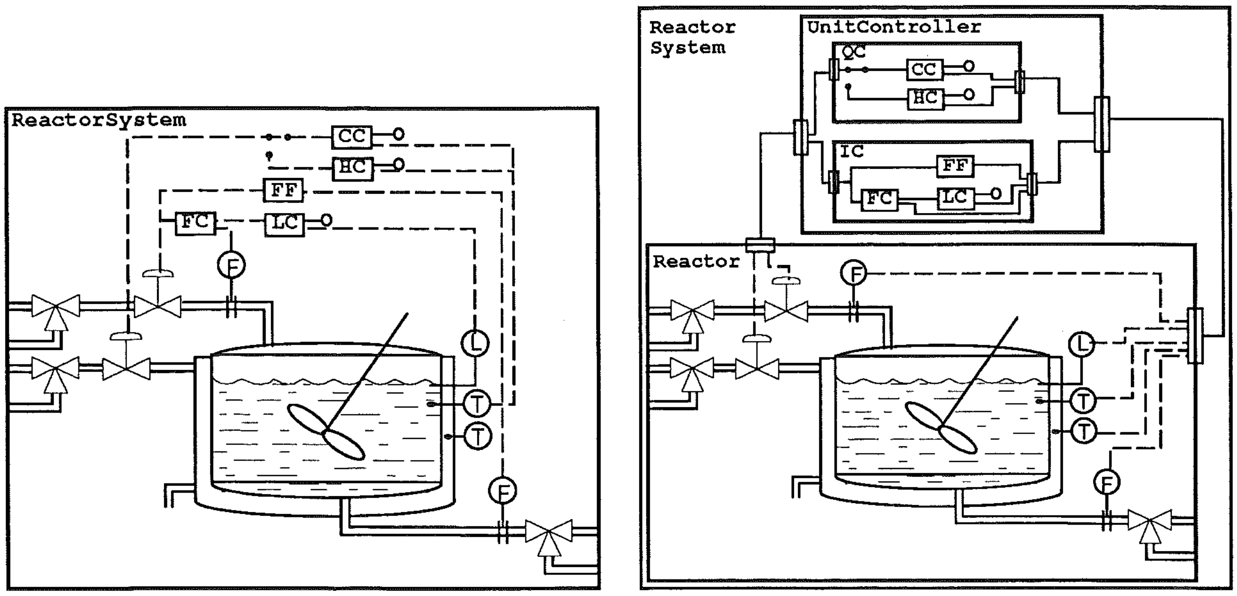


Figure 7.5 Two ways of representing a process control system, distributed to the left and centralized on the right.

7.4 Hierarchical Control Systems

The natural interpretation of a distributed control system is compensated by the abstraction facility that gives a centralized control system an internal structure. The centralized representation of control systems is preferred because of the abstraction facility which is important in process design.

A controlled unit can be decomposed into the physical unit description and the unit control system. The unit control system is a composite description of a number of local control systems. These local control systems are also composite and composed of other controllers. The controllers in the bottom in the control system structure hierarchy are called *primitive controllers* and the unit and local control systems *composite controllers.* This creates an unit operation control system hierarchy. Above the unit, there are control systems on the plant and plant section levels. These control systems are supervisory and coordinative. The unit control system is centralized into one object but from the plant view the control system is distributed in the unit modules.

- The plant control system consists of supervisory and coordinative composite controllers.
- The plant section control system is similar to the plant controller and it consists of supervisory and coordinative composite controller.
- The unit control system is a composite controller for the control of a unit. Notice that it also contains sequential and monitoring control.
- The local control system is a feedback composite controller for local

control of a control objective.

- The elementary controller is a composite controller. Examples are feedback, feedforward, or sequential controllers.
- Primitive control objects are the primitive building blocks of composite controllers.

These six levels of decomposition are now discussed in the reversed order. Structuring of unit controllers is also discussed in [Nilsson, 1993b].

The Building Blocks of Local Controllers

The primitive control objects are the building blocks of hierarchical control systems. Examples of continuous primitive control objects are PID modules, limiters, adders etc. The basic building blocks are few, but more specially designed controllers for multivariable, adaptive, or nonlinear control would increase the number of building blocks. Other examples are the Grafcet primitives, steps and transitions. The elementary controllers are ordinary control objects, like PID controllers, feedforward controllers, etc. They are composed of a number of primitive control objects. They are the parameterized controllers that are easy to reuse.

The local composite control system uses the elementary controllers to make a special control system for a particular control variable. Different types of control variables are production rate, inventory, environment, quality and economics, see [Shinskey, 1987]. The combination of a number of elementary controllers that handle different local control variables are large but in practice there are a limited number of composite controllers. Structures of elementary controllers are sometimes called idioms or idiomatic control, see [Bristol, 1980]. These idiomatic controllers can be described as a number of classes that can be reused in different applications, i. e., in different local composite controllers.

Unit Control Systems

The unit composite controller is composed of a number of local controllers which control a number of control variables, like inventory and product quality. In cases with large number of interactions there is a need for decoupling between local controllers. This decoupling is represented on the unit control level. Many of the idiomatic controller structures found in [Bristol, 1980] and [Shinskey, 1987] use both linear and nonlinear decoupling between different local controllers. An unit control system for the debutanizer distillation column, illustrated in Figure 7.6, is composed of four local controllers. The control actuators are the heat input in the reboiler, heat output in the condenser, outflow in the bottom and in the top, and finally the reflux flow. The column in Figure 7.6 has no heat out-

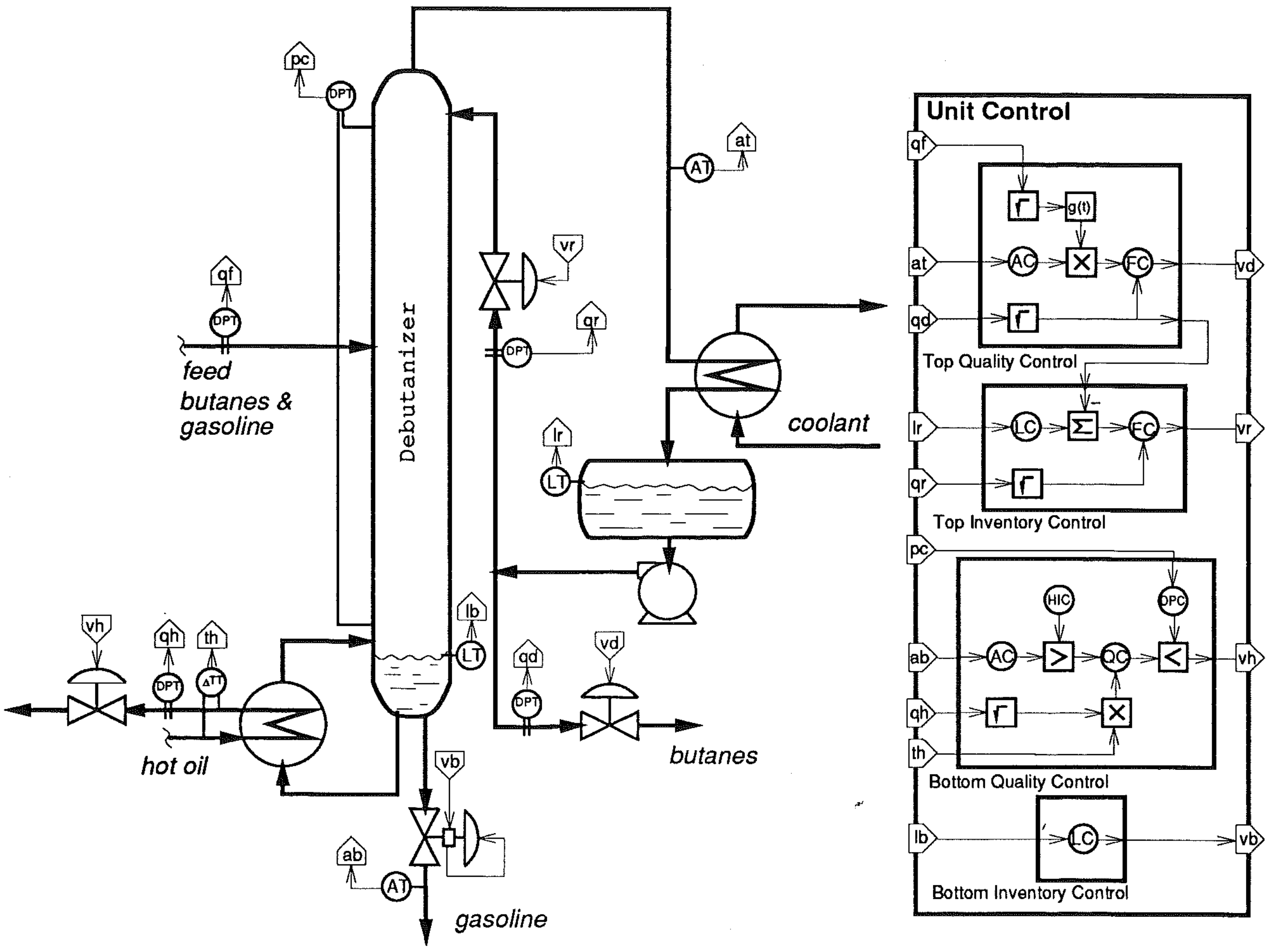


Figure 7.6 The debutanizer distillation column with a centralized unit controller.

put control. The condenser cooling flow is instead set by the surrounding equipment. The centralized unit controller is composed of four local controllers, namely quality and inventory control of the top and bottom of the distillation unit. The only coordinated control on the unit level is a feed forward of the flow measure in the top quality control to the top inventory control. Each local controller is a composite controller except for the bottom inventory controller which is composed of one level controller. The other three local controllers are based on cascade configurations with a primary control loop around the actuator valve. All three of them use different methods for feedforward and nonlinear control. The top inventory controller uses feed forward control of the distillate flow. The top quality controller uses nonlinear scaling with respect to the feed flow. The bottom quality controller is a cascade configuration where the primary loop controls the reboiler effect using a nonlinear transformation of two measurements. It also has two selectors. A minimum of reboiler effect is set by the max selector. The minimum selector switches to pressure control when the pressure or the control signal from the effect controller are too high.

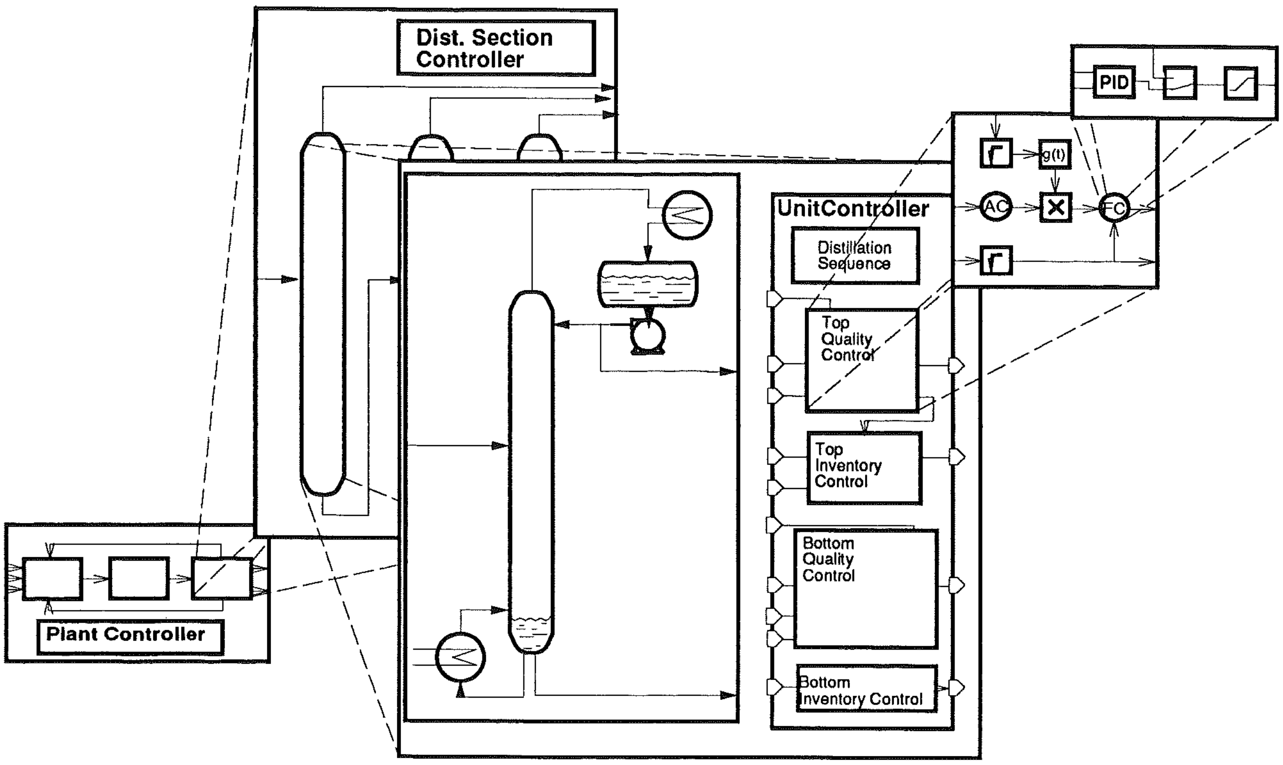


Figure 7.7 The process hierarchy with unit oriented control systems. The unit controller with internal hierarchy of control objects.

Plant Control Systems

The plant and plant section control systems are similar to the unit controller. These controllers start and stop the controlled units. They supervise the unit control systems. Interaction control or coordinative control are also included in these controllers. The control system hierarchy is illustrated in Figure 7.7. The distillation unit controller is composed of four local controllers which in turn are composed of elementary controllers. The distillation section is composed of controlled units and a distillation section controller. The plant is composed of controlled sections and a plant controller. A decomposition of the control system like this relies on the ability to make hierarchically decomposed control systems. There is a need for hierarchical control system design and tuning, like guidelines for structuring and designing of large control system.

7.5 Controller Class Hierarchy

The classification guidelines for control system classes are similar to the process model classification guidelines discussed in Chapter 4. They are therefore unified into common guidelines. The classification guideline is as follows: *application - granularity - unit type - interface class - model class.*

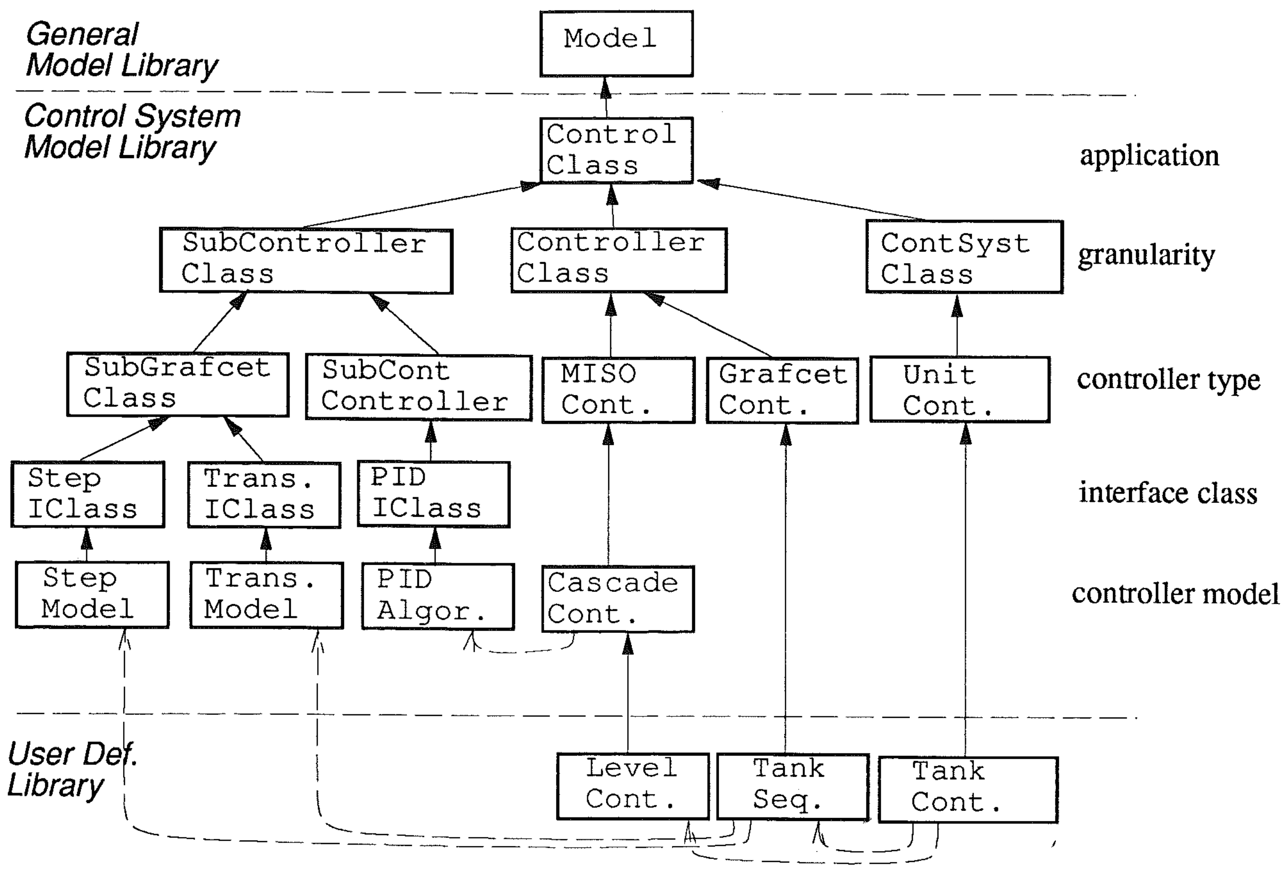


Figure 7.8 The CSTR hierarchy with additional classes structuring the class tree. (arrow - *is-a* relation, dashed arrow - *has-a* relation.)

The guidelines are illustrated in Figure 4.7. The application super class is the `ControlClass` which is the common super class for all control system building blocks. The next level is the granularity level. The control class tree is here divided into three parts, namely `SubControllerClass`, `ControllerClass` and `ControlSystemClass`. The primitive control objects are the building blocks for controllers and are subclasses of the subcontroller class. They are analogous to subunits in the process class tree. Examples are grafcet primitives, like step and transitions, and controller building blocks, like PID algorithms, adders, limiters etc. The elementary and local controllers are the reusable controllers that are well parameterized and can be used directly in graphical control system models. They are subclasses of the controller class and they are composed of primitive controllers. Examples are PID controller, cascade controller, start up sequences etc. The composite controllers are control system models describing integrated unit controllers composed of continuous controllers and sequences and they are subclasses of the control system class.

The unit type level or the controller type level is used to separate different controller types from each other. Examples of controller type classes are Grafcet class, simple controller class, DMP class etc. The composite controllers are often user defined and are analogous to flowsheets in the process model class tree. But for particular units, predefined unit controller may be developed. Examples of control system types are unit controllers, plant section controllers, and plant controllers. These library organized classes are followed by the interface class and the model class levels. They have the same meaning here as in Chapter 4.

7.6 Conclusions

The representation and structure of process control systems are discussed in this chapter. Omola can describe continuous and discrete controllers. It can also describe sequential controllers and other types of control systems that are based on events or quantitative measurements.

Analog controllers can be described directly in Omola using the equation oriented representation. To describe discrete time controllers a sampling mechanism is needed. To describe sequential controllers and logic, Omola has an event concept.

Abstraction of control systems into reusable components is important. A process control system can be decomposed into a structure hierarchy:

- coordinative controllers on plant and plant section level,
- unit controllers on unit class level,
- local composite controllers as components of unit controllers,
- elementary controllers as components of local controllers.

The class hierarchy structure is based on the same levels as for the process class guidelines: *application - granularity - controller type - interface class - controller model.*

8

Multi-Facet Models

Model representations of processes with units and associated control systems have been discussed in the previous chapters. They have been focused on abstraction and reuse which are facilitated through an object-oriented approach. Structure decomposition together with class decomposition create a large number of objects in the model database. The aim is to have a model database with a lot of reusable objects that can create new, more complex objects. To work with a model database like this is a classical database problem. The discussion here will point out some problems and indicate some solutions.

Models, used in multiple problem solving, must fulfill a number of different demands. The universal model that can be used in any problem for any purpose is not practical and is extremely difficult to develop and understand. Instead, a collection of different models for different purposes should describe the system from different perspectives or views. One can talk about multi-facet models. A multi-facet model contains a set of behavior descriptions. In Omola this can be handled by multiple realizations, which means that the realizations have a common interface of terminals. Objects that are modeled with very different models with no common interface must be described in different model classes, multiple class models.

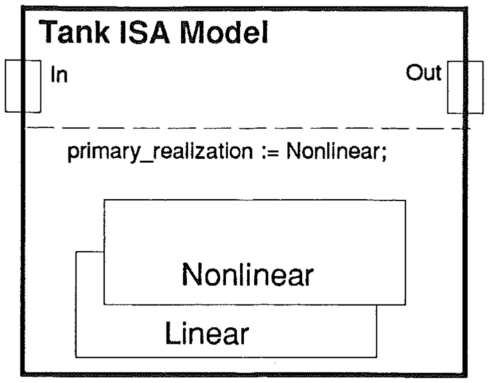


Figure 8.1 A model Tank with two realizations describing a nonlinear and a linear behavior. The Omola code is seen in Listing 8.1.

8.1 Multiple Realizations

An object can be described in a number of different ways. Often it is interesting to have multiple model behaviors. It should be possible to change the interior behavior description if the different behaviors of one model have a common interface of terminals. There is a concept called *multiple realization* in Omola. It is not completely implemented in the current version of OmSim. Realization is a predefined super class of a model component in Omola that can be used to aggregate the interior of a model class, i. e., a set of variables and equations. The realizations have therefore a common interface of terminals, which means that they are polymorphic. The internal behavior can be changed without any interaction problems with the surrounding models. Polymorphic realizations should have common parameter sets, but this is unpractical because a new behavior means that new phenomena are modeled and therefore new parameters are also added.

A typical example of multiple realizations in control engineering is the need for one nonlinear description and one linearized version of the description, which is illustrated in Figure 8.1 and in Listing 8.1. Other examples of descriptions can be state space and transfer function model representations. The selection of the proper realization is made by the user by assigning the primary realization parameter a realization class name. This primary realization parameter is therefore a kind of structure parameter.

EXAMPLE 8.1—Tank model

The tank model in Listing 8.1 has two different realizations, one nonlinear equation description and one linear state space description. The second is a linearization of the first. A model user can select the realization of interest by assigning the parameter `primary_realization` a realization class name. By default the last defined realization is the valid one (pri-

```
Tank ISA Model WITH
terminals:
  In  ISA ZeroSumTerminal WITH direction:='in;  END;
  Out ISA ZeroSumTerminal WITH direction:='out; END;
parameters:
  g ISA Parameter WITH value:=9.81; END;
  PipeArea, CrossArea ISA Parameter;
realizations:
  primary_realization := Nonlinear;
  Nonlinear ISA Primitive WITH
  variables:
    Acc,h ISA Variable;
  equations:
    In = Out + Acc;
    Acc = CrossArea*h';
    Out = IF h>0 THEN PipeArea*sqrt(2*g*h) ELSE 0;
  END;
  Linear ISA Primitive WITH
  parameter:
    ho ISA Parameter;
  variables:
    h,a,b ISA Variable;
  equations:
    h' = a*h + b*In;
    a  = - PipeArea/CrossArea*sqrt(g/(2*ho));
    b  = 1/CrossArea;
  END;
END;
```

Listing 8.1 A tank model with two different realizations, one nonlinear and one linear descriptions.

mary), in this example `Linear`. To choose `Nonlinear` one has to make an explicit parameter declaration of the primary realization. □

8.2 Multiple Class Models

If the differences in the models are large, so large that they do not have a common interface, then they have to be described in different classes. The need to describe some kind of relation between the classes is still relevant.

Semi-Polymorphic Models

Semi-polymorphic models can have the same terminal names but the internal structure of the terminals differs. This means that models like these can have the same interface super class. The subclass inherits the

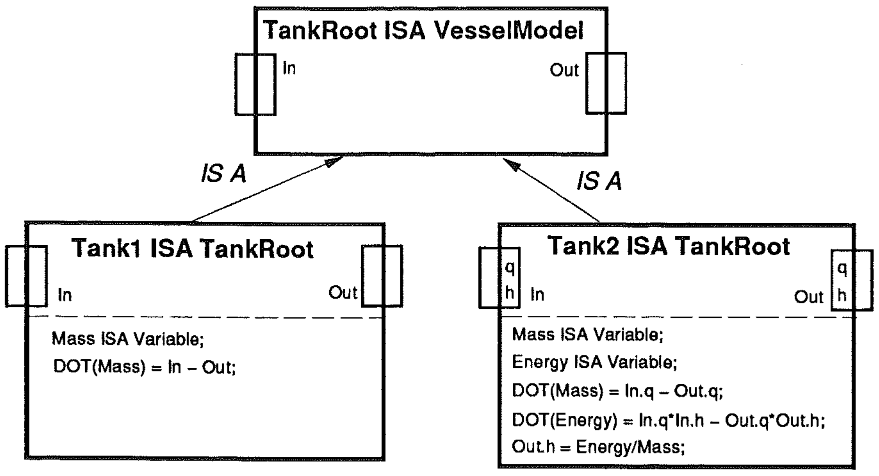


Figure 8.2 A tank model super class with terminal definitions is specialized into two tank classes. Notice that Tank2 must overwrite the terminal definition.

terminals and overwrites the original terminal super classes with new ones with other internal descriptions. One example of semi-polymorphic models in process engineering is seen in Figure 8.2. The class `TankRoot`, which contains two terminal attributes, is specialized into two tank model classes. `Tank1` adds a behavior based on a mass balance. `Tank2` adds a behavior based on both mass and energy balances. This results in new definitions of the inherited terminal attributes. These are overwritten with new ones which are record terminals with two components.

The resulting tank models in Figure 8.2 are not polymorphic. But if the connected models in a composite model also change their internal terminal description the composite super model can be reused. The models are polymorphic seen from the composite model if all submodels are changed in a coordinated way.

Multiple Class Models

Multiple class models are models of the same physical system that are very different. They are so different that they have almost nothing in common, neither interface nor interior. They are truly non-polymorphic. There is still the need to associate them with each other. The problem is how to represent a relation between multiple models of the same system. For maximized reuse the models have different super classes. If they are forced to have a common super class, then it is an empty super class. In other words there is a need for additional relation descriptions that can describe the relations between the models, a multi class model relation. This semantic relation is used to organize the model database and cannot be used in the model compilation.

Relations or semantic links can be implemented in a number of different ways. *Multiple inheritance* can be used where one super class is the

```
LinearTank ISA TankIClass WITH
% Not correct Omola
relation:
  Is_linearization_of ISA Relation WITH
    class := NonlinearTank;
  END;
...
END;

NonlinearTank ISA TankIClass WITH
% Not correct Omola
relation:
  Is_nonlinear_model_of ISA Relation WITH
    class := LinearTank;
  END;
...
END;
```

Listing 8.2 A suggestion of relation attributes in Omola. Two classes with corresponding relation attributes describe the relation between the two classes.

super class for multiple class models. This is not a good solution. A class will have different super classes, one for reuse and one for classification. Using the same concept to describe two different things can be confusing. *Relation attributes* that describe relations between one class and another class, is a second approach. A third approach is to have a *multiple view object*, MVO, which is a meta class for the multi-class models. The models have different relations to the MVO. The MVO represents the *real* system and all model objects are only different views of the real system from certain perspectives. Multiple view objects are discussed in [Årzén, 1992].

Relation attributes are defined locally to describe the links to other classes. One suggestion is illustrated in Listing 8.2. Here the relation is unidirectional, from the owner to the related class. All relations are distributed in the different classes concerning a physical system. A multi view object is a meta object containing relations to the different descriptions of the same physical system. An example is illustrated in Listing 8.3. The MVOTank has two relations, to the nonlinear tank and to the linear tank. The relations are centralized to the MVO which contains the total description of a system in the model database. The relation attribute approach results in a distributed network where each individual class contains the relation descriptions. The whole relation structure will therefore be difficult to describe and need a new display tool. The multiple view object approach results in a centralized relation description with a well defined tree structure.

Multiple class models are not so important in the current OmSim

```
MVOTank ISA MultiViewClass WITH
% Not correct Omola
relations:
  Nonlinear_model ISA Relation WITH
    class := NonlinearTank;
  END;
  Linearized_model ISA Relation WITH
    class := LinearTank;
  END;
END;
```

Listing 8.3 An example of a multi view object in Omola (a suggestion). It contains two relations to two global classes, one nonlinear model and one linearized model.

because the number of very different descriptions of the same object is limited. The number of multiple models will increase in an environment with more than one problem solving tool.

8.3 Batch Process Models

It is often problematic and inconvenient to describe a batch process in one single model. The model must capture all batch phases and handle the discrete changes of operational conditions. These models become too complicated to understand and maintain because of the complexity. One simplification is to divide the model into batch phase models instead. Batch phase models are easier to develop and to interpret.

Automatic Realization Selection

Instead of having realizations that describe different phenomena, one can have realizations with different validity intervals. The choice of realization depends on the state. A natural extension is therefore to have automatic selection of realizations. The decision to change realization can be made by the event mechanism discussed in the previous chapter. The action part in the event is just a parameter assignment. The key problem is that the parameter is a structure parameter which means that the simulator must be able to handle a switch of realizations. This can be done if the model compilation handles all realizations simultaneously before simulation or by making run-time model compilation.

EXAMPLE 8.2—Pressure vessel with a bursting disc
A model of a pressurized gas vessel with a bursting disc is illustrated in Listing 8.4 It is composed of one terminal, a number of parameters and one variable. It also has two realizations which are both composed of

```
GasVessel ISA Model WITH
terminal:
  In ISA PipeInTerminal;
parameters:
  V,M,R,T,K1,K2,P0 ISA Parameter;
variable:
  p ISA Variable;
realizations:
  % Not correct Omola
  Intact ISA SetOfDAE WITH
    p' = R*T*K1/(V*M)*sqrt(In.p - p);
  END;
  Bursting ISA SetOfDAE WITH
    p' = R*T/(V*M)*(K1*sqrt(In.p-p) - K2*sqrt(p-P0));
  END;
events:
  Init, Burst ISAN Event;
selections:
  % Not correct Omola
  ONEVENT Init DO
    primary_realization := Intact;
  END;
  ONEVENT Burst DO
    primary_realization := Bursting;
  END;
  ONEVENT p>10*P0 CAUSE Burst;
END;
```

Listing 8.4 A pressurized gas vessel with a busting disc described in Omola with automatic realization selection.

one equation. The intact realization describes the pressure in the vessel when the in terminal pressure changes and the bursting disc is intact. The bursting equation also describes the changes of the pressure caused by the inflow pressure but in this case with an erupted disc. The model also has two events, `Init` and `Burst`. On the init event, which occurs at time zero, the normal realization is selected as active. On the burst event the second bursting realization is selected. A nameless state event causes the burst event and the condition is that the pressure is above ten times the normal pressure. □

An automatic realization switch must handle continuous state transitions. This can be done in two ways, by common state variables or explicit state assignments. The simplest version is to define the state variable outside the realizations which means that the equations inside the realizations operate on the same variables. This is the case in Example 8.2. The realization switch just changes the active set of equations. The other way is by explicit state assignments. This means that the realizations are

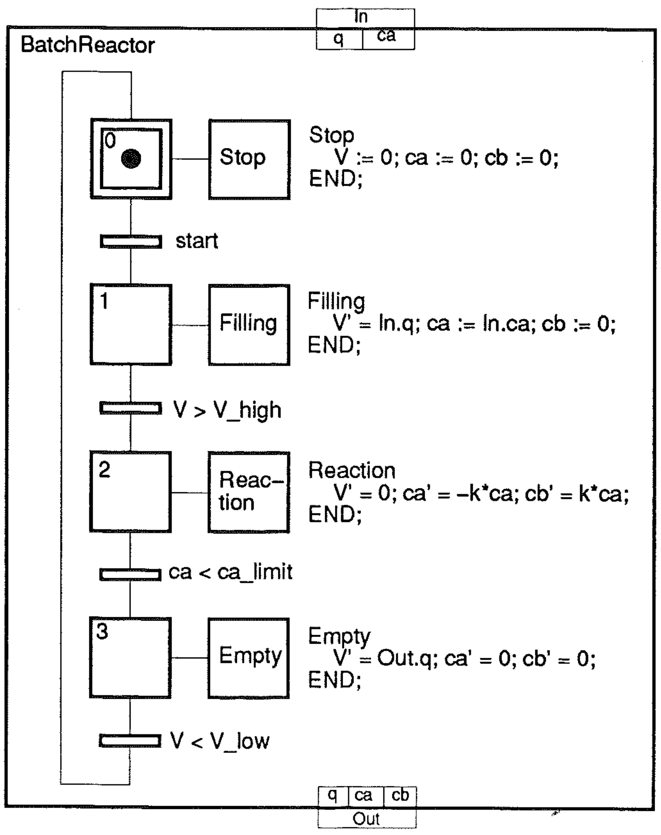


Figure 8.3 A batch reactor model in Omola using a Grafcet for automatic realization selection.

independent of each other and that the realization switch must therefore handle the state transition. The switch makes a transformation of the old state variables to the new states variables.

The behavior of the process can be described distributed in local realizations and the selection of behavior can be described centralized. The descriptions of the behavior phases are abstracted from the description of the phase sequence. Example 8.2 is a modification of an example in [Barton and Pantelides, 1991] using gPROMS. A conditional case statement selects the active equation set in gPROMS. The switch between two cases is described inside the active case.

Batch Process Modeling

The use of multiple realizations becomes powerful in batch process modeling. The behavior in batch processes changes drastically depending on the phase of the batch. To describe the process behavior with common equations for all phases becomes complex and difficult to overview. Distributed descriptions of each phase are much easier to handle. A centralized description of the phase changes make the process description simple. As the batch phases are repeated in a sequence this centralized description can be represented with Grafcet. Modeling of batch processes with Grafcet is described in [Nilsson, 1991].

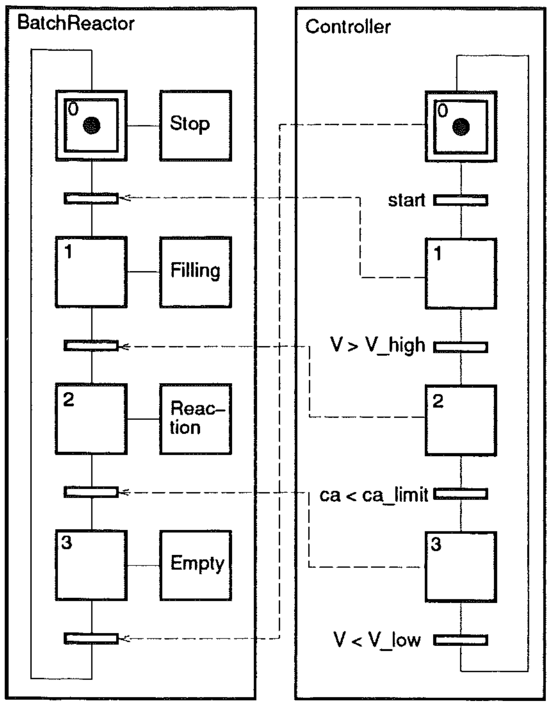


Figure 8.4 The principle of controller and batch process separation.

Grafcet primitives in Omola have been described in detail in Section 7.2. Grafcet can be used to describe a sequence of batch phase models, as illustrated in Figure 8.3. The actions associated with the steps can be assignments of new active realizations. The reactor has different realizations for each step. As mentioned above, the realizations can have internal states. The step actions must include state transformations from one realization to another.

One example is illustrated in Figure 8.3. The batch reactor has one `In` and one `Out` terminal. It has one Grafcet sequence and four realizations. The sequence makes the proper realization selection. It is composed of four steps. The initial step activates the `Stop` realization. This realization initializes the three states to zero. On the event `Start` the sequence goes to the first step. The `Filling` realization has one differential equation describing the mass balance. When the volume reaches a certain value the sequence is switched to step two. Here the realization has three differential equations, one where the volume derivative is zero and two other describing a simple chemical reaction. When the concentration of a reactant is below a predefined concentration the sequence switches to the last step. The `Empty` realization has one volume differential equation.

This type of batch process model is a mixture of the process equipment description and the sequential controller. The model can instead be divided into two sequences, one for the batch reactor and one for the control sequence. The description of batch processes, as illustrated in Figure 8.4, divides the behavior into a number of phase behavior models. These behavior models can be predefined classes in a batch process library. This batch process behavior decomposition makes the development of new models simpler. The open description in Omola makes it possible to change the sequence. Grafcet is developed in a graphical editor. It is also possible to change each individual realization describing the behavior of each batch phase.

Petri nets are used to structure a batch process simulator in [Czulek, 1988]. In spirit this is similar to the Grafcet approach discussed here. The structure of the Petri net based simulator, in [Czulek, 1988], is on the other hand invariant. The user can change the action in each place and change the transition conditions. Another approach to modeling of batch processes and combined discrete and continuous processes is presented in [Barton and Pantelides, 1991], using gPROMS. gPROMS has language constructs for describing procedures. These procedures can be actions of tasks which are similar to events in Omola. Tasks can be hierarchical and parameterized. With the task concept, sequences can be described in gPROMS.

Similar ideas to the batch process model structuring are also discussed in the AI community. A model using different modeling techniques on different abstraction levels is called a *multi-model*. The ideas discussed in this section with finite-state machine descriptions for primitive model abstraction is also presented in [Fishwick, 1992] and [Fishwick, 1993].

8.4 Model Database

The current version of OmSim has an internal database of Omola classes. The classes can be developed inside OmSim or in an external editor. For long time storage OmSim saves the classes in the database on files. The user must organize the storage himself. A permanent object-oriented database management system, OODBMS, is a better solution and will probably be the choice in the future. An OODBMS must handle complex objects, equivalent objects, and object versions, see [Hurson *et al.*, 1993]. Complex objects are the composite models described in Omola. Equivalent objects are related to the multiple class models discussed in Section 8.2. Object versions are not handled at all in current OmSim.

Structure and Modularization

The structure of the class tree is discussed a number of times in this thesis, in Section 4.3, 5.4 and 7.5. The class tree has a number of levels, first a number of structuring levels and then levels for reuse. A typical modeling example includes hundreds of classes. In a situation where predefined class trees for different applications are available a model database will have perhaps over a thousand classes. It will therefore be important to structure the class tree. The classification guidelines discussed in Chapters 7 are as follows:

- The application class categorizes the class tree into different application dependent subtrees.
- The granularity class is a super class classifying the grain size of the class in the structure hierarchy.
- Unit types are super classes which divide the rest of the class tree in particular unit sub trees.
- The interface class is a the super class for polymorphic subclasses.
- The model class is the classification level for the actual reusable model description.

To support multiple developers and users the class tree can be modularized into libraries. The merging of different class trees creates problems with the global name space. Libraries therefore have a local name space which makes it possible for different developers to develop independent libraries. Libraries are stored on files. One library can be stored on many files and one file may contain many libraries. The rule "one library - one file" is recommended.

A class in a library can be a subclass of a class in another library. This means that the second library must be loaded before the first one. This creates a hierarchy of libraries. The library definition looks like the list below. The head is composed of a library name definition and a list of libraries that are used.

```
LIBRARY ProcessEquipmentLib;
USES ProcessTerminalLib;

{list of omola classes}
```

In a library every global class must have a unique name. However, two different libraries can have two classes with the same name which can be referred through the library name subseeded by the class name, `ProcessEquipmentLib::Valve` and `MyHomeMadeModels::Valve`. A model developer only has to worry about unique naming in his or her own library.

There is one type of library in OmSim and it is open for everybody. All classes can be manipulated and this is sometimes not desirable. There is a need for other types of libraries.

- A *work space* is an area in the database where new models currently is developed. It is the same as the OmSim-library today.

```
TankModel ISA Base::Model WITH
% Not correct Omola
relations:
  Graphic     ISA Relation WITH class := TankGraphic;     END;
  Description ISA Relation WITH class := TankDescription; END;
terminals:
  In    ISA TankSystem::PipeInTerminal;
  Out   ISA TankSystem::PipeOutTerminal;
  Level ISA Base::SimpleTerminal;
parameters:
  Density, GravConst, TankArea, PipeArea ISA Base::Parameter;
variable:
  mass ISA Base::Variable;
equations:
  mass' = Density*(In.Flow - Out.Flow);
  mass = Density*TankArea*Level;
  Density*GravConst*Level =
    Out.Pressure + Density*abs(Out.Flow)*Out.Flow/PipeArea^2/2;
  In.Pressure = 0.0;
END;
```

Listing 8.5 The user defined mathematical description of the tank model in the introductory example.

- A *library* is closed and a user is only allowed to read and reuse. The user is not allowed to change or add models in the library unless the user has permission.
- A *reuse-only-library* is a further development of the closed library concept. This kind of library does not allow the user to read the Omola code and the user only has access to the class interfaces. However, the classes can be reused in the normal way.

Presentation and Browsing

A model database will contain many classes. It is important to support the user with tools that facilitate browsing, search, reuse and specialization. To work with a large database is an information problem, especially working with a library developed by another user. The presentation of models and search methods become important issues. The classes in the model database can have a number of different presentations. In the Omola case the following descriptions may be available:

- *Icon* is a graphical description of the class,
- *Unit description and interface* is a mixture of the definition of terminals, parameters and optional comments.
- *Model assumptions* can only be described by comments.

```
TankGraphic ISA Base::Model WITH
% Not correct Omola
relations:
  Model       ISA Relation WITH class := TankModel;       END;
  Description ISA Relation WITH class := TankDescription; END;
icon:
  Graphic ISA super::Graphic WITH
    bitmap TYPE String := "icontank";
  END;
terminals:
  In ISA TankSystem::PipeInTerminal WITH
    Graphic ISA super::Graphic WITH
      x_pos := 200.0;  y_pos := 300.0;  invisible := 1;
    END;
  END;
  Out ISA TankSystem::PipeOutTerminal WITH
    Graphic ISA super::Graphic WITH
      x_pos := 200.0;  y_pos := 0.0;    invisible := 1;
    END;
  END;
  Level ISA Base::SimpleTerminal WITH
    Graphic ISA super::Graphic WITH
      x_pos := 400.0;  y_pos := 225.0;  invisible := 1;
    END;
  END;
END;
```

Listing 8.6 The class with only the graphical description of the tank model in the introductory example in Appendix B.

- *Object relations*, such as structure and class hierarchies, can be described in OmSim today.
- *User oriented description* is a mathematical description or graphical interpretation of the class.
- *Omola code* which can be displayed today in OmSim.

Only Omola code, object relations, and graphical descriptions of composite models can be presented in the current browser, together with the OmSim graphical MED-editor. The object relations are the *is a* and *part of* hierarchies that can be displayed by a graphical tool.

Multiple presentations can be supported in two ways: by tool filtering and by multiple classes. Tool filtering means that the object contains all the different presentations and that the tool selects the one to be presented. This is the way OmSim works today and this makes the Omola code hard to read because of all the graphical information. This is not good. This is clearly seen in the tank system example in Appendix B. Half of the Omola code for the tank model is graphical descriptions which are generated by the graphical editor and not by the user. If the Omola model is divided into one interface class and one model class a separation of the graphical information and the model description can be achieved. The interface with the terminal descriptions contains all graphical descriptions.

A better solution is the multiple class concept. The user-defined model and the automatically defined graphical description are described in separate classes. The classes are related by a semantic link or relation.

This approach can be generalized and allows a number of different model classes or views of an object. These ideas are illustrated in the example in Listing 8.5 to 8.7. The actual model description is found in Listing 8.5. This is the primitive mathematical description. All graphical information is described in the tank graphic class listed in 8.6. The major part is the terminal location descriptions. Textual descriptions of the tank object are defined in the third class, the tank description, seen in Listing 8.7. This distributed tank model representation is called the relation approach in the Multi Class Model section. The multi view object approach can of course also be used here. The example, in Listing 8.5 to 8.7, is used to illustrate the concept of distributed description and not the multi class representation.

It is difficult to find the right model in a large database even if you know what you are looking for. It is almost impossible for an inexperienced user to find models in new libraries. If a modeling environment has advanced model development tools the model developer tends to develop new models of his own instead of searching for predefined models. Efficient search methods and browsing facilities are importance. One simple idea is to introduce keywords that describe the class in question. One example of a keyword list is illustrated in Listing 8.7. The user interact with a search tool by entering keywords and the tool lists all classes that are described by these keywords. By adding and deleting keywords the class list is interactively changed.

As mentioned in the previous subsection it should be possible to lock a library. To lock a library the developer must define class descriptions, model assumptions, and a keyword list. This forces the developer to make proper documentation of the library before being authorized to lock it.

8.5 Conclusions

Multi-facet models in a broad perspective are discussed in this chapter. They are used in multiple model descriptions, batch process models, and in model database structuring. Omola has a multiple realization concept which makes it possible to allow a model to have more than one behavior

```
TankDescription ISA Base::Model WITH
% Not correct Omola
relations:
  Graphic ISA Relation WITH class := TankGraphic; END;
  Model   ISA Relation WITH class := TankModel;   END;
descriptions:
  UnitDescription ISA TextClass WITH
    text :=
    "A class describing a liquid medium container with one
    inflow and one outflow. The flow terminals are composed
    of one flow and one pressure component. The unit has
    also one level measurement terminal";
  END;
  Assumption ISA TextClass WITH
    text :=
    "The model is based on one dynamic mass balance. The
    density and tank cross area are assumed to be constant.
    The outflow is described by a Bernoulli equation.";
  END;
keywords:
  TankKey ISA KeyWordList WITH
    wordlist := (mass conservation, vessel, container);
  END;
END;
```

Listing 8.7 A tank class with textual attributes describing the unit, its interface and model assumption. It also contains a keyword list for quick search.

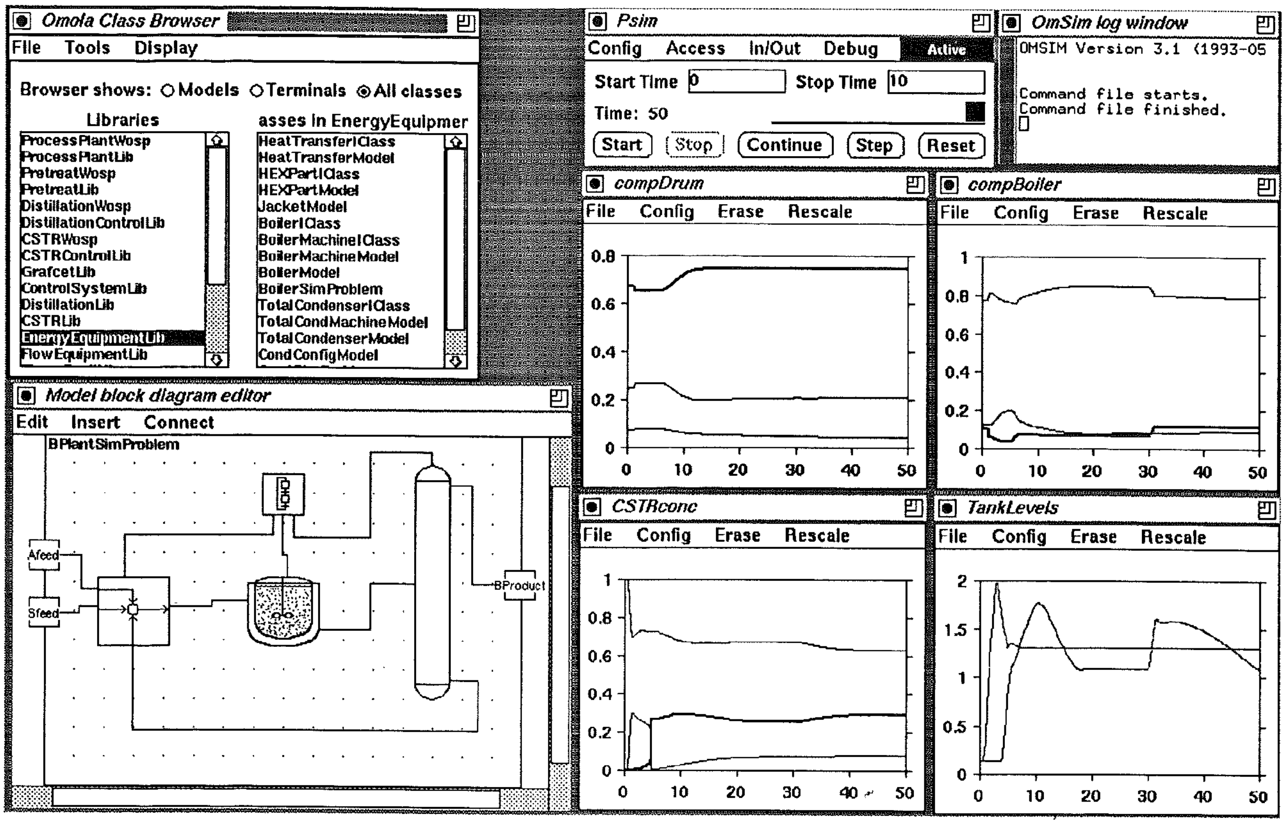
description. This can be used in user defined behavior parameterizations where the user selects the proper behavior. Automatic switching between realizations cannot be done in the current OmSim. It would require a new type of model compilation that can handle multiple realizations simultaneously. In batch process modeling a sequence of realizations can describe the overall behavior. The sequence can be described in an ordinary graphic based sequence language such as Grafcet or Petri nets. The need for a relation concept for database organization is also discussed. It can be implemented in a number of different ways.

9 Process Example in Omola

In this chapter we apply the modeling techniques discussed in previous chapters to develop a model of a simple, but complete chemical process. The example uses the structure and class hierarchy guidelines. It also uses the medium and machine decomposition of units in order to increase the reusability. Each dynamic unit has a unit controller. The control systems are composed of both sequential and continuous controllers. These concepts are exemplified and illustrated in the current version of OmSim.

The example used in this chapter is a chemical plant composed of a pretreatment section, a reaction section and a separation section. The reaction and the separation section are simply composed of one unit each, a continuous stirred tank reactor and a distillation column unit respectively. The simulation illustrates both the sequential and continuous dynamic properties of the process. The Omola code describing the example is listed in [Nilsson, 1993a].

The description of a plant can be done graphically in OmSim by the reuse of library unit models. These models are selected from libraries listed in a browser and placed in a graphical editor. A browser and a MED graphical editor are found on the left in Figure 9.1. The graphical editor is displaying the composite plant model. The connections can then be drawn in the editor.



```
OMSIM Version 3.1 (1993-05

Command file starts.
Command file finished.
```

Figure 9.1 A screen dump of OmSim with a simulation of the process example.

A simulation of the chemical plant example is also shown in Figure 9.1. The composite model shown in the editor can be compiled and transformed into simulation code. The main simulator panel for driving the simulation is found in the top of Figure 9.1. Four plot windows are found below the simulation panel. They show the distillation drum and boiler composition, in the windows *compDrum* and *compBoiler*, and the reactor composition, in *CSTRconc* together with the levels in the reactor and the recycle tank, in *TankLevels*. The start up of the plant, at time 1, is shown together with a reference change of the distillation boiler composition at time 30.

9.1 The Chemical Plant Example

We will consider a small scale chemical plant producing the product B using the raw material A, which is seen in Figure 9.2. The reaction $A \longrightarrow B$ occurs in the solvent S. Solvent and reactant A are mixed to produce the feed to a continuous stirred tank reactor, CSTR. The reactor outflow goes to a distillation column where unreacted material and product are separated. The unreacted material leaves the column in the bottom and is recirculated via a buffer tank. The product leave the column in the top.

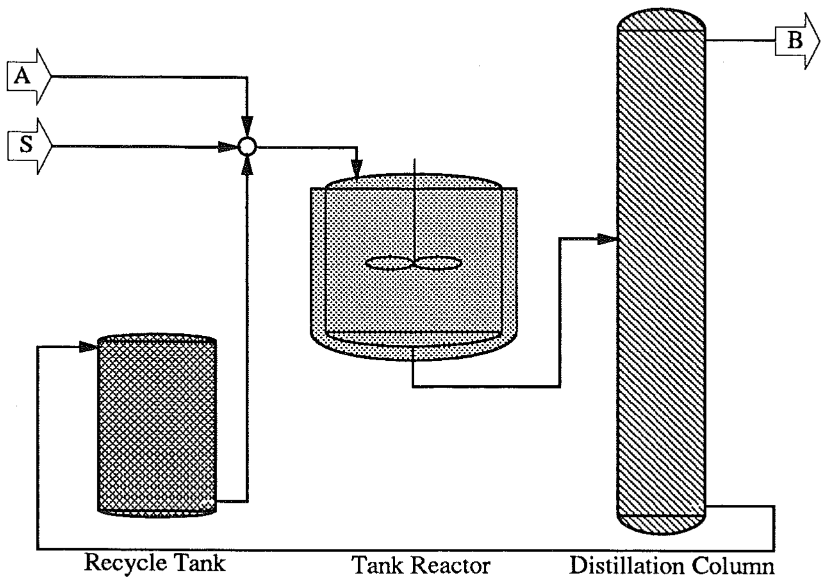


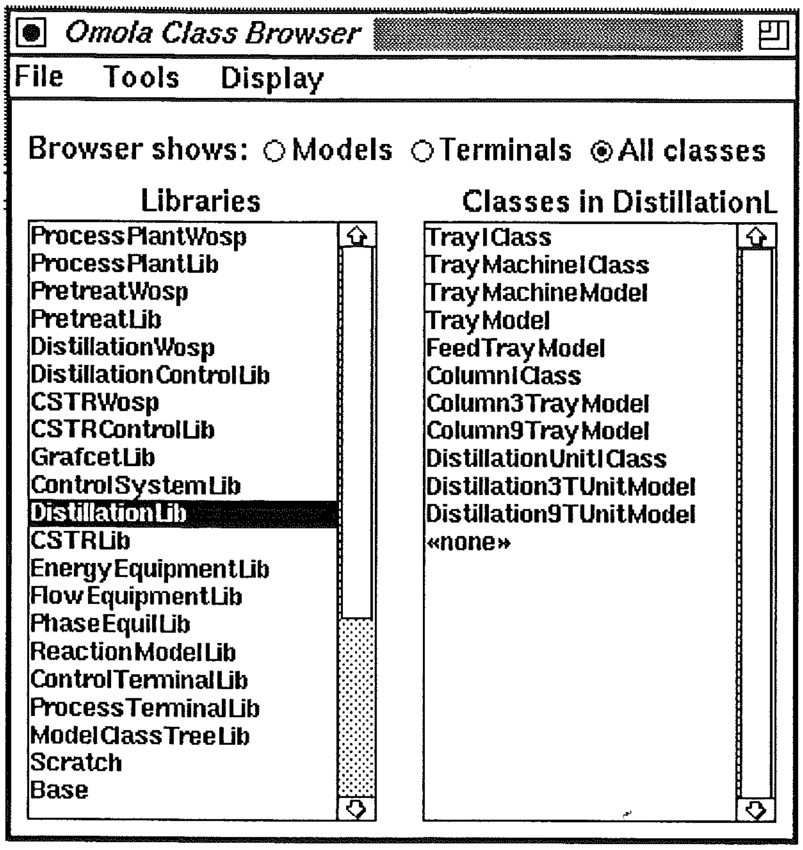
Figure 9.2 The chemical plant example used in this chapter.

Plant Decomposition

According to the decomposition guidelines in Chapter 4, we should have the following levels: plant - plant section - unit - subunit. The plant is decomposed into three plant sections: pretreatment, reaction and separation. The last two plant sections, reaction and separation, are only composed of one submodel each. The plant section level has been excluded. The reactor and distillation units on the other hand have complex internal structures which will be discussed in the following sections. The construction of plant sections and plant configurations are plant dependent while units and subunits often are commonly reused constructions. Unit models are therefore developed in the spirit of being general and reusable. Note that there is a plant controller for supervisory control, see Figure 9.1. It is composed of a sequential controller which coordinateds the start up and shut down of each individual unit.

Chemical Plant Database

There are over 200 globally defined Omola objects in this plant application, where 30 are classification classes, 40 are terminal and parameter classes, and 25 are simulation problem classes. The rest, about 110, are interface classes and model classes. All of these classes are grouped into 19 libraries, which are seen in the Omola class browser in Figure 9.3. Four libraries are called work spaces and they contain simulation problem classes. Two libraries are terminal libraries. The classification classes are defined in the `ModelClassTreeLib`. The last two libraries in Figure 9.3 are predefined in OmSim. All libraries and their classes are listed in [Nilsson, 1993a].



Omola Class Browser

File Tools Display

Browser shows: ○ Models ○ Terminals [✓ All classes]

Libraries

- ProcessPlantWosp
- ProcessPlantLib
- PretreatWosp
- PretreatLib
- DistillationWosp
- DistillationControlLib
- CSTRWosp
- CSTRControlLib
- GrafcetLib
- ControlSystemLib
- DistillationLib
- CSTRLib
- EnergyEquipmentLib
- FlowEquipmentLib
- PhaseEquilLib
- ReactionModelLib
- ControlTerminalLib
- ProcessTerminalLib
- ModelClassTreeLib
- Scratch
- Base

Classes in DistillationL

- TrayIClass
- TrayMachineIClass
- TrayMachineModel
- TrayModel
- FeedTrayModel
- ColumnIClass
- Column3TrayModel
- Column9TrayModel
- DistillationUnitIClass
- Distillation3TUnitModel
- Distillation9TUnitModel
- «none»

Figure 9.3 The Omola class browser loaded with the chemical process libraries.

9.2 Tank Reactor Model

The tank reactor in the plant example is a jacketed continuous stirred tank reactor with a control system with both sequential and continuous controllers. The medium and machine separation concept is exemplified and the internal structure of the control system is discussed in this section.

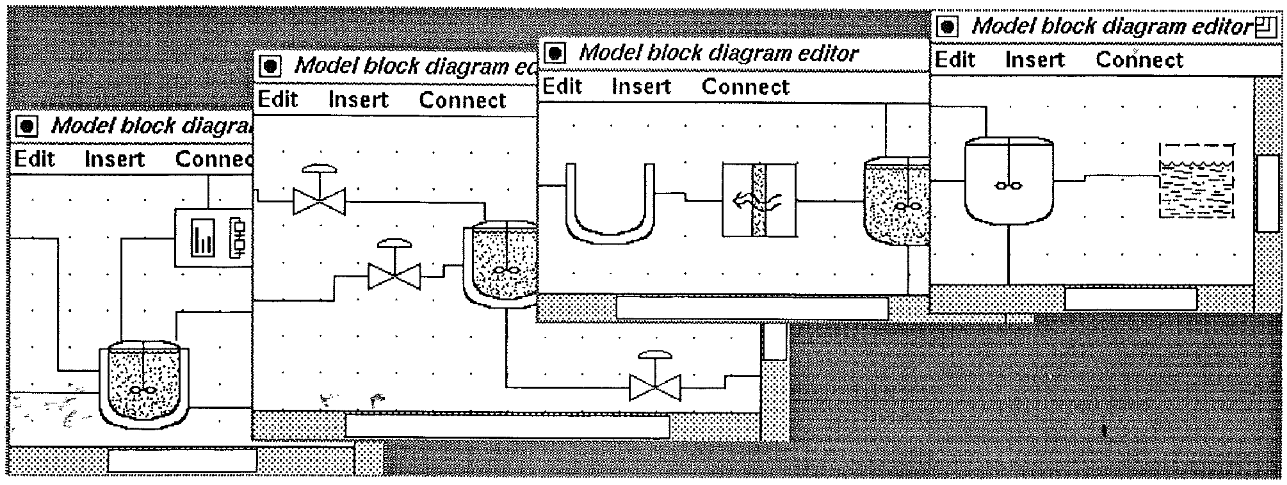
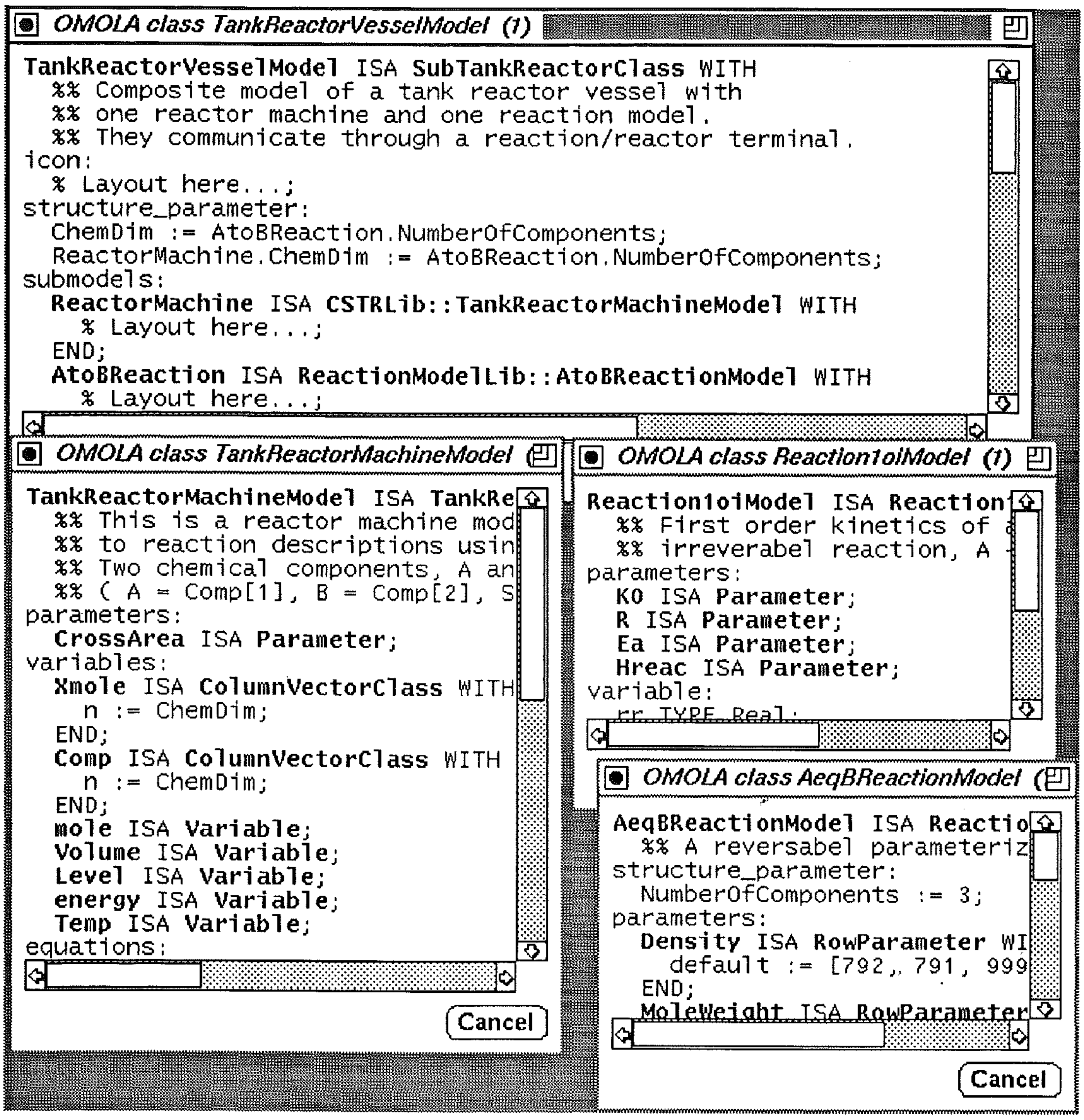


Figure 9.4 The tank reactor decomposition is illustrated in four hierarchical MED editors.


OMOLA class TankReactorVesselModel (1)
OMOLA class TankReactorMachineModel
OMOLA class Reaction1oiModel (1)
OMOLA class AeqBReactionModel (
Cancel
Cancel

```
TankReactorVesselModel ISA SubTankReactorClass WITH
  %% Composite model of a tank reactor vessel with
  %% one reactor machine and one reaction model.
  %% They communicate through a reaction/reactor terminal.
icon:
  % Layout here...;
structure_parameter:
  ChemDim := AtoBReaction.NumberOfComponents;
  ReactorMachine.ChemDim := AtoBReaction.NumberOfComponents;
submodels:
  ReactorMachine ISA CSTRLib::TankReactorMachineModel WITH
    % Layout here...;
  END;
  AtoBReaction ISA ReactionModelLib::AtoBReactionModel WITH
    % Layout here...;
```

```
TankReactorMachineModel ISA TankRe
  %% This is a reactor machine mod
  %% to reaction descriptions usin
  %% Two chemical components, A an
  %% ( A = Comp[1], B = Comp[2], S
parameters:
  CrossArea ISA Parameter;
variables:
  Xmole ISA ColumnVectorClass WITH
    n := ChemDim;
  END;
  Comp ISA ColumnVectorClass WITH
    n := ChemDim;
  END;
  mole ISA Variable;
  Volume ISA Variable;
  Level ISA Variable;
  energy ISA Variable;
  Temp ISA Variable;
equations:
```

```
Reaction1oiModel ISA Reaction
  %% First order kinetics of a
  %% irreverabel reaction, A -
parameters:
  KO ISA Parameter;
  R ISA Parameter;
  Ea ISA Parameter;
  Hreac ISA Parameter;
variable:
  rr TYPE Real;
```

```
AeqBReactionModel ISA Reactio
  %% A reversabel parameteriz
structure_parameter:
  NumberOfComponents := 3;
parameters:
  Density ISA RowParameter WI
    default := [792,. 791, 999
  END;
  MoleWeight ISA RowParameter
```

Figure 9.5 The reaction medium and reactor machine decomposition. One left a part of the machine model and the reactor model are found, and on top right the composite reactor vessel above the specialized reaction description.

Reactor Structure

Figure 9.4 illustrates the decomposition of the tank reactor. The figure is a screeen dump of the actual OmSim environment and it shows four hierarchical graphical editors. The continuous stirred tank reactor with a control system is found in the window on the left. The second window from the left shows the decomposition of the reactor into one reactor and three control valves. The reactor is decomposed futher, in the third window from the left, into one vessel, one jacket and one heat transfering wall. Finally, the vessel is decomposed into one reactor machine model and one reaction model in the window to the right in Figure 9.4.

A part of the Omola description of the reactor vessel is found in the top window of Figure 9.5. The vessel is decomposed into one machine and one medium model and they are found in the windows below the vessel

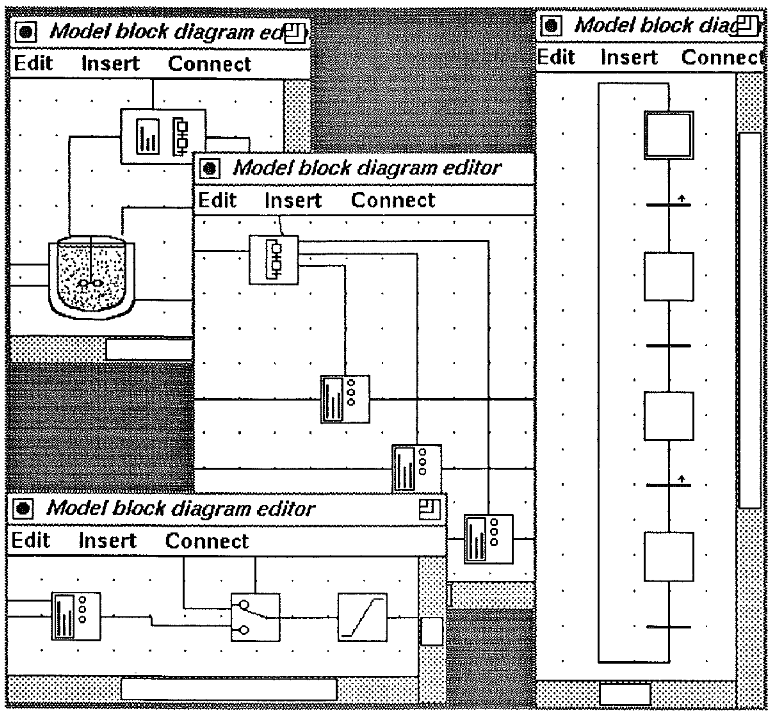


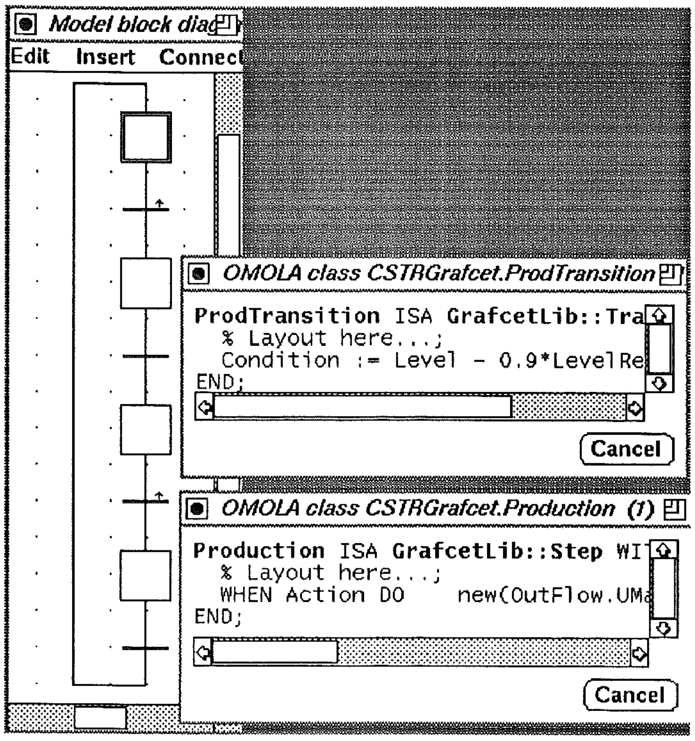
Figure 9.6 Decomposition of the CSTR controller into one sequential controller and three PID controllers, all with internal structure.

window. The fourth window, bottom right, is a specialized medium model description. The chemical dimension parameterization of the reactor vessel can be seen in these windows. The chemical dimension in the vessel and in the vessel machine are set equal to the number of components parameter of the medium model. This is done in `TankReactorVesselModel` under the structure parameter tag. The actual chemical composition variables are defined in the machine model, lower left in Figure 9.5. It interacts with the reaction model found on the right which describes the reaction rate and energy production. A subclass of this reaction model, `AtoBReactionModel`, seen bottom right in Figure 9.5, contains the parameter values together with the definition of the chemical dimension, `NumberOfComponents`. The chemical dimension of the whole reactor is set by the internal reaction model parameter attribute `NumberOfComponents`. This means that another reactor for another application only has to be a specialized subclass of this reactor class, with a locally redefined reaction model super class.

The Composite Reactor Controller

The centralized unit controller of the controlled CSTR is hierarchically decomposed into a number of levels. Three levels are illustrated in Figure 9.6. The unit controller is composed of three PID controllers and one sequential controller.

The sequential controller is described graphically using Grafcet. Note


Model block diag
Edit Insert Connec
OMOLA class CSTRGrafcet.ProdTransition
Cancel
OMOLA class CSTRGrafcet.Production (1)
Cancel

```
ProdTransition ISA GrafcetLib::Tra
  % Layout here...;
  Condition := Level - 0.9*LevelRe
END;
```

```
Production ISA GrafcetLib::Step WI
  % Layout here...;
  WHEN Action DO    new(OutFlow.UM
END;
```

Figure 9.7 The sequential controller in the CSTR control system.

that the user defined Grafcet objects, shown in the Omola display windows in Figure 9.7, are subclasses of predefined library classes. The user only has to define the action and the condition descriptions. The internal behavior of the Grafcet is abstracted in the super classes. The Grafcet controller starts and stops the PID controllers individually through connections.

The PID controller is composed of one PID module, one manual to automatic switch, and one control signal limiter, which is seen in Figure 9.6. Note that there is information going in both directions in the PID controller connections. The information about limiter saturation is propagated backwards to the controller for windup tracking. This also solves the manual mode problem which automatically turns the PID module into tracking.

9.3 Distillation Unit Model

The decomposition of the distillation unit follows the same ideas used in the tank reactor example in the previous section.

Distillation Unit Structure

The controlled unit is composed of one unit controller and one unit model, which is seen on the left in Figure 9.8. The second window from the left is the distillation unit its with control system. The third window displays

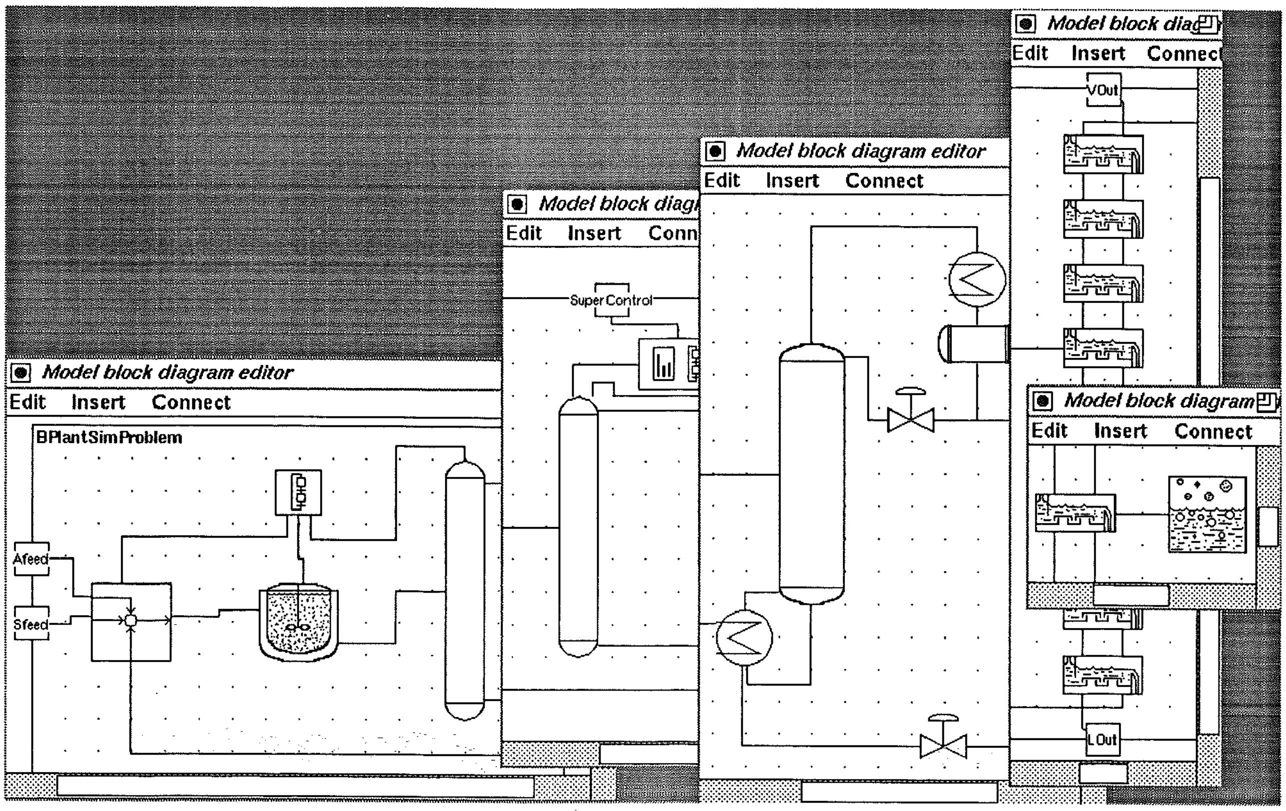


Figure 9.8 Decomposition of the plant model down to the medium and tray machine decomposed tray model in the distillation unit.

the distillation unit model, which is composed of a number of other units: reboiler, reflux drum, tray column, control valves etc. The tray column is composed of a number of trays in a series. Each tray is medium and machine decomposed which is illustrated in the window on the right in Figure 9.8. This means that the distillation unit has the same polymorphic properties as the reactor. It is only possible to change the medium model super class in order to adapt the unit to a new application. Note that a regular structure mechanism that automatically creates the internal structure of the tray column is needed. Also, an abstract super class concept for the medium models is useful in order to parameterize the column efficiently. This is discussed in Chapter 6.

Distillation Model Classes

All distillation model classes used in the example are found in the class tree display in Figure 9.9. The classes shown are all subclasses of the `ProcessClass` which is called application type class in the class hierarchy guidelines in Chapter 4. The next level is the granularity type class, and the `SubUnitClass` and `MediumClass` are seen in Figure 9.9. The unit type classes are the third and final categorization class in the class hierarchy guidelines. `SubSeparatorClass` is the unit type class for the column model classes. It has two subclasses which are interface classes. The tray

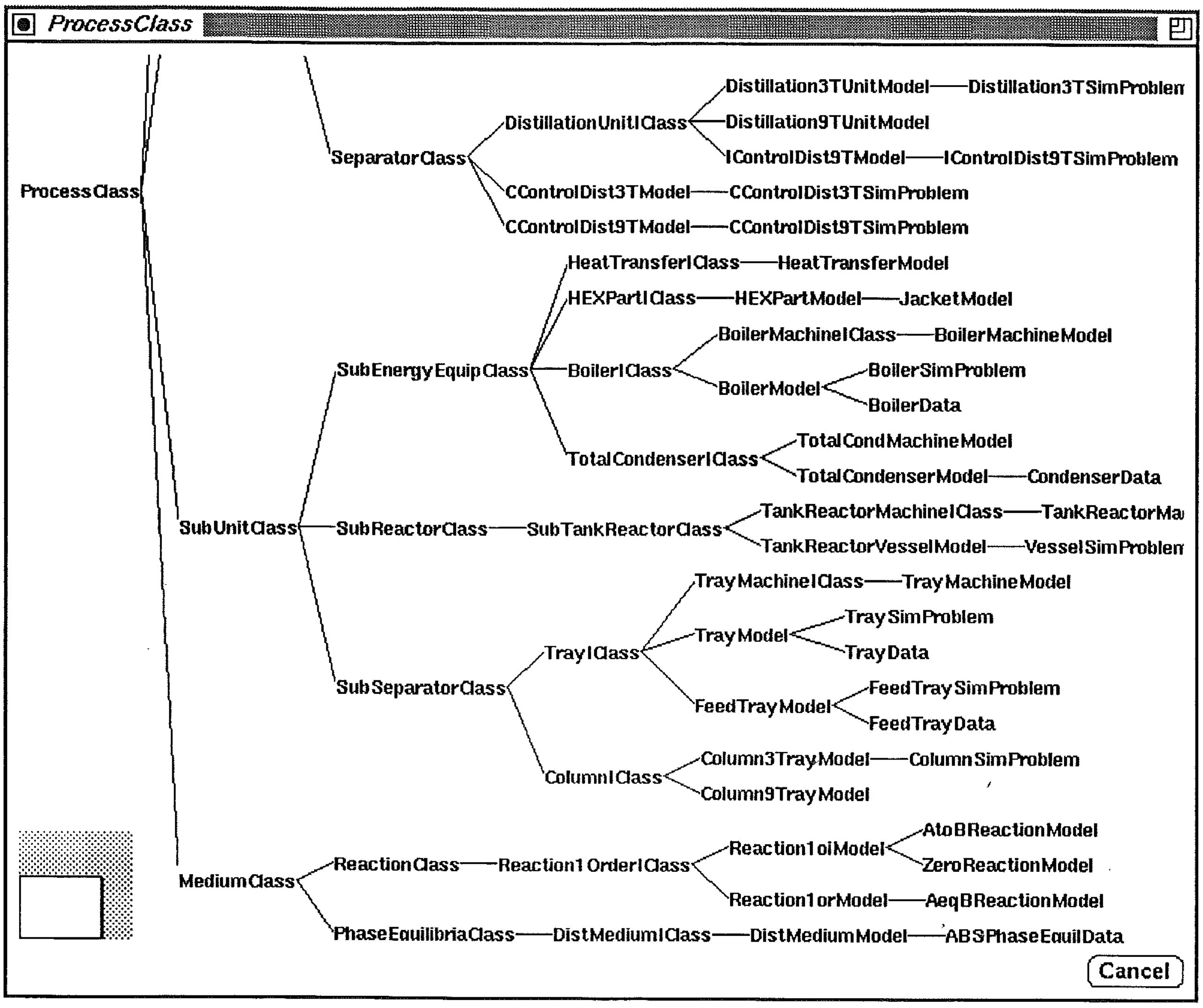


Figure 9.9 A part of the class tree in the chemical plant example.

interface class TrayIClass is the super class for two tray models and one tray machine interface class. This is an exception from the guidelines.

Notice that there are six to eight levels in the class tree. The root class is the predefined Omola class Model followed by three classification levels. The fourth level is the interface class followed by the model class. Direct reuse of a model means that the inherited model class is specialized by parameter assignments and this is done in a simulation problem class. Additional levels in the class hierarchy occur if exceptions from the guidelines are needed, like additional interface class, e. g., TrayMachine-IClass, or specialization of model classes.

Specialization of a primitive model class into a new model class is complicated. Equations and connections are nameless and can therefore not be overwritten. To specialize a well defined primitive model class into a new model class means that the inherited equations are still valid and new ones are added, which together have a new, well defined meaning. One example is found in the introductory example in Chapter 3. It is often not the case. Reuse and specialization are therefore more successful

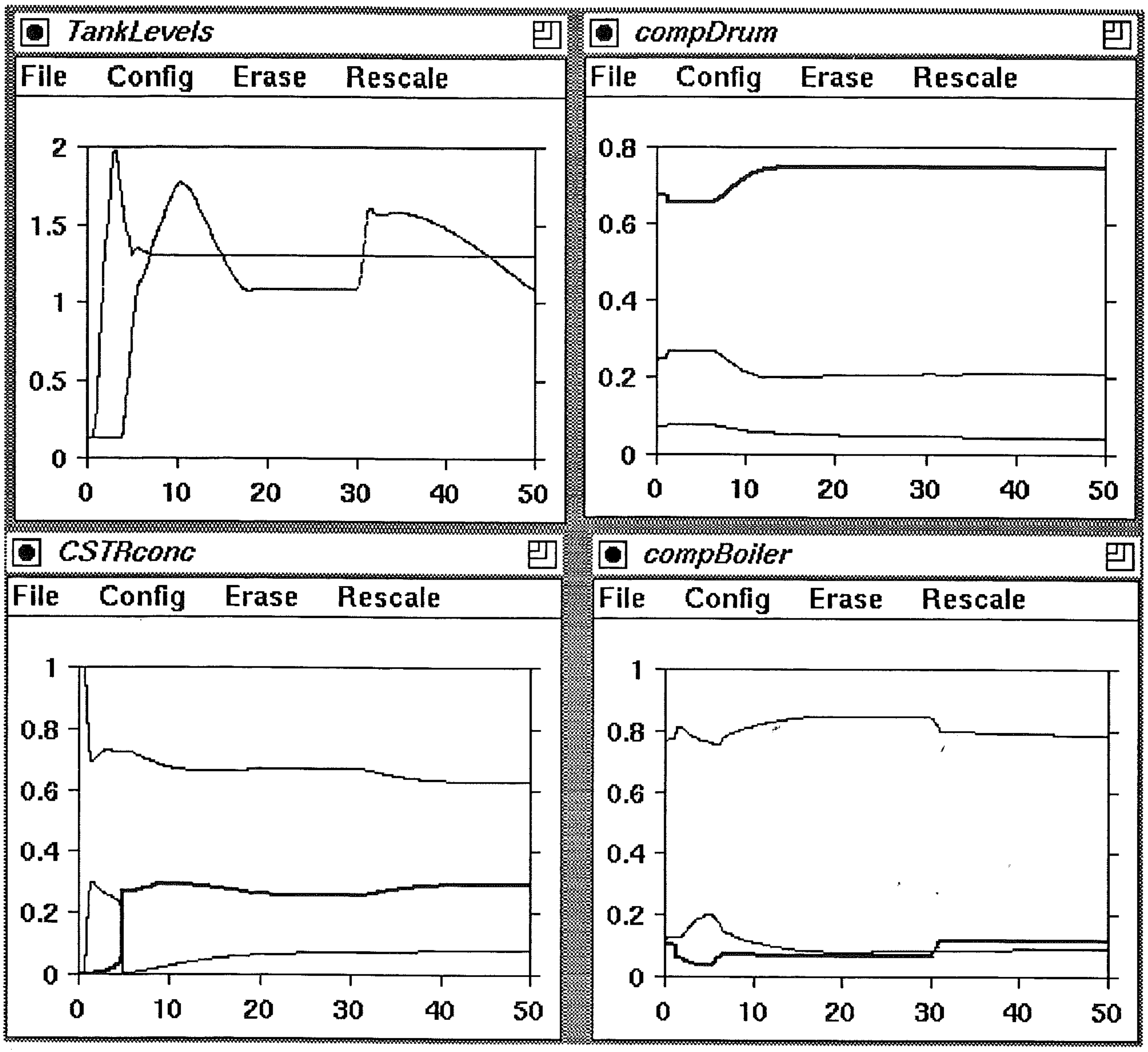


Figure 9.10 A simulation of the process example.

in composite models than in primitive models. It can be good to divide a primitive model description into a number of subseeding classes. This makes it possible to reuse models that are not complete and that lack a number of equations. This is difficult to generalize and therefore is only the interface and interior class separation discussed.

9.4 Process Simulation

The process example is translated into simulation code by the simulator tool. An interactive simulation study can be done by the use of the simulator subtools. A plant simulation is shown in Figure 9.10. It shows some of the process variables, namely top and bottom composition in the distillation column on the right and reactor and buffer tank levels in top left, and finally, the reactor composition in bottom left. After the start up transient, 4 to 8 hours, the process is at steady state after 20 hours. A reference change is done on the bottom composition controller at 30 hours. This change causes large changes in the buffer tank level, which is desired.

The reactor is modeled as discussed in Chapter 4 and also seen in Figure 9.5. The distillation unit model is simplified compared to the discussion in Chapter 6. In the phase equilibrium model the algebraic loop is removed by the use of relative volatility models. The vapor composition is a function of the liquid compositions. This means that the energy dynamic is removed and replaced by a function of the composition. The model assumes also that the liquid is always at equilibrium which means that the start up dynamics are not well described. The assumption that there are no vapor and pressure dynamics has a number of consequences. It simplifies the actual modeling. On the other hand, it makes some descriptions unrealistic and strange. Examples are the control system modeling, particularly controllers that use flow and pressure measurement.

9.5 The Use of OmSim

The development of the process example is done using three different editors, a text editor called Emacs, the MED graphical editor, and a bitmap editor.

The text editor can be used outside OmSim. Developed models are stored in files which can be loaded into the OmSim database. The text editor can also be used from the inside where OmSim and Emacs exchange model descriptions via temporary files. The graphical description developed in a MED editor is translated into Omola code which can be stored together with text editor developed models. Icons are developed in an ordinary bitmap editor in UNIX. The result is stored as a file for each bitmap. The graphical description attribute in every Omola object can have an internal bitmap attribute indicating its bitmap file. The MED editor can then use this information to show the icon in the right position.

It is customary to group Omola models into libraries and which are stored on files for long time storage. A recommendation is to have one library per file. Note that bitmap files are stored separately.

Model Development

The model development process can be divided into three phases: the interface development, the model development and the simulation problem development.

The interface development begins with the definition of the terminal classes in the text editor. The model interface is then defined in the

MED editor. It is good practice to make this a separate interface class. A primitive model can now be developed in the text editor and it is a subclass of the interface class. The same goes for composite models with their internal structures of submodels.

This way of developing models minimizes the interaction with the graphical information which is now found in the interface class. Advanced use of parameterization and submodel changes force the user to interact with the graphical information in composite models. These problems have been discussed in Chapter 8. This can be solved by the construction of specially designed editors but this is not desired. It is better to use standard tools and use advanced representations that make this separation possible.

After the model class is defined a simulation problem formulation should be developed to test the model. Even if the model is a submodel and not intended to be used alone it can be worth while making this test. These simulation problems are stored in separate libraries called work spaces. There are 25 simulation problem classes in the process example mentioned in the first section in this chapter. They are used to check out different parts of the plant model.

Simulation

When a model class with a well defined simulation problem subclass is developed, the subclass can be simulated. The model compilation turns the class into simulation code.

Many different checks are done. Syntax and semantic checks are often carried out already in the loading phase of a library into OmSim. A more detailed analysis is done together with the resolving of all variables to check the mathematical consistency of the problem. After this the equation set is manipulated in a number of different steps. This is discussed in Section 3.3. It can sometimes be hard to realize what is happening and why. Particularly if the problem formulation is consistent but wrong from the application point of view. This forces the equation manipulation into undesired analysis. The error message is hard to understand and to interpret for an inexperienced user. This manipulation step should perhaps be interactive where the user supports the manipulation with problem characteristics and the manipulation gives the user insight to his own problem formulation.

The simulation interface is defined after the model compilation is done. First a model access tool is defined by a menu choice. It contains all parameters, terminals and variables in the simulation problem. It is possible to interactively change parameters and initial values of state variables. After this plot windows can be defined. The variables that are going to be plotted are selected in the model access tool and connected to the right plot window. Before starting the simulation it is possible to change the simulator options, like the numerical method, error tolerances, etc.

The lack of a steady state solver for consistent initial conditions makes it difficult to work with DAEs in the current version of OmSim. The algebraic equations must be initialized with correct initial conditions in order to start the DAE solver, DASRT. Low order DAE systems can be treated as in the introductory water tank example in Chapter 3. Here the DAEs are calculated at time zero by the action of an init event. This can be found in the very end of Appendix B. For large DAE systems, like the distillation columns discussed in Chapter 6, this is not a practical solution.

Interface

Everything in OmSim is window based and each tool and subtool have separate windows. The majority of the user actions are menu or push button driven. An option menu choice results in a special kind of window which cannot be moved. This is sometimes impractical and there should perhaps be ordinary windows. A general comment is that working with OmSim results in many windows and this is valid for many graphical based technical programs.

9.6 Conclusions

The Omola Simulation Environment is successfully used to model and simulate a chemical plant. The chemical plant is composed of a pretreatment section, a tank reactor and a distillation column unit. It is controlled by four sequential controllers and nine continuous PID controllers. Omola has a number of modeling concepts that facilitate development of large models, like hierarchical submodel decomposition for abstraction and inheritance hierarchy for reuse. The medium and unit machine decomposition and the unit model and control system separation are important special cases in dynamic process applications. The process structure is decomposed into about 5 levels. The plant model is composed of about 200 global Omola classes in a class hierarchy with about 7 classification levels.

10

Conclusions

The thesis has presented an approach to object-oriented modeling of chemical processes and process control systems. The thesis presents a modeling methodology and methods which are applied to a process example. The methodology can be used in any process modeling activity, but the presentation is focused on dynamic models. A new modeling language, Omola, has been used as a tool. Although this tool is used to implement the ideas the modeling methodology can be implemented in other languages supporting object-orientation and use of real equation to describe behavior.

Methodology

The modeling methodology is based on the idea of decomposition into a library of reusable submodels. These submodels can easily be specialized by the use of inheritance in a class tree of models.

- Guidelines for hierarchical structure decomposition are presented. A plant is decomposed hierarchically into smaller and smaller pieces: plant sections, units and subunits. A dynamic unit is divided into one control system and one process unit. A unit is divided into one medium model and one unit machine model.
- Guidelines for class tree organization are also presented. All submodels derived in the hierarchical structure can be found in the model class tree. The major part of the submodels are reused from predefined libraries. New submodels are often specialized versions of the old submodels using inheritance.

General Process Modeling Concepts

General modeling concepts can also be extracted from the methodology. A number of the ideas presented are general.

- Guidelines for hierarchical structure decomposition can be used to structure process models. A chemical plant structure is decomposed hierarchically as follows: *plant - plant section - unit - subunit.* This is discussed in Chapter 4 and 7 and illustrated in the example in Chapter 9.
- Guidelines for hierarchical class decomposition are used to organize the class tree and to increase the reuse of model classes. The class tree is decomposed into following levels: *application - granularity - unit type - interface class - model class.* This is also discussed in Chapter 4 and 7.
- The notation of media and machine decomposition have been found to be very useful in models of chemical processes. This is shown, e. g., in the example in Chapter 9.
- Decomposition of behavior into a number of equation objects make it possible to reuse equations. This results in a solution that is based on structure decomposition, single inheritance and a new concept of abstract classes which are discussed in Chapter 5.
- Abstract classes can be used to parameterize a composite model with respect to a submodel. An alternative concept is called parameterized classes. These concepts are discussed in Chapter 6.
- Batch process model decomposition into individual descriptions of each batch phase is suggested in Chapter 8. The selection of the active phase is done by an ordinary sequential controller description.

Omola Extensions

Some of the modeling concepts discussed in the thesis cannot be handled conveniently in the current version of Omola. Some suggestions for extensions of Omola have been made. Some missing features are serious and some are less important. The major suggestions are:

- The lack of regular structure description is probably the most important drawback of the current Omola in process applications.
- It would be useful to have abstract classes of the type discussed in Chapter 5 and 6 in Omola. It is important in some parameterization methods of large models and for local inheritance in structure hierarchies.

- Parameterized classes is a powerful concept as shown in Chapter 5 and 6. It would be useful to have direct support of these parameterized classes in the modeling language.
- Automatic switching between different internal descriptions facilitates modeling of complex behavior. This creates a difficult model compilation and requires a major revision of the simulator as discussed in Chapter 8.
- Multiple inheritance is an interesting concept that is easy to include in Omola. However, there are no arguments in the thesis that defend its importance in process model applications. Some users may find it useful and therefore Omola should allow multiple inheritance.

Future Work

A direct continuation of the work in this thesis is to develop libraries for different applications in the spirit of the thesis methodology. This can be done in the current version of the Omola simulation environment.

Some of the concepts discussed in the thesis are not implemented in Omola today and therefore not explored in detail. Implementation and exploration of these concepts are interesting. Interesting concepts to explore further are automatic switching between realizations, equation decomposition of behavior, parameterized classes, and abstract model classes.

A complement to the OmSim environment could be a modeling assistant, an intelligent system for the generation of Omola code. Together with an equation class library and an user oriented front end the assistant should generate code from an application oriented description. The ideas in `DESIGN-KIT`, [Stephanopoulos *et al.*, 1987], or `HPT`, [Woods, 1993], can be implemented into an open modeling language like Omola using a modeling assistant.

Omola can describe combined continuous and discrete systems. They can be differential and algebraic equation systems with discrete events. To allow other primitive model descriptions is interesting to also include. Transfer function descriptions should be possible to transform into ordinary differential equations, ODEs, and use the current simulation tool. In process applications, particularly partial differential equations, PDEs, are of great interest to describe. A language extension like this is simple. The problem is to develop problem solving tools. One simple solution is the use of the method of lines and static gridding for transformation of PDEs into ODEs. This simple solution is used in some simulation tools and can only be used to a particular class of PDE. A more general approach to PDE simulation is a complicated problem and needs advanced solvers.

11

References

ANDERSSON, M. (1989): "An object-oriented modeling environment." In IAZEOLLA *et al.*, Eds., *Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation*, 1989 European Simulation Multiconference, Rome, pp. 77–82. The Society for Computer Simulation International.

ANDERSSON, M. (1990): *Omola—An Object-Oriented Language for Model Representation.* Lic Tech thesis TFRT-3208, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ANDERSSON, M. (1992): "Discrete event modelling and simulation in Omola." In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design, CADCS '92*, Napa, California.

ANDERSSON, M. (1993a): "Modelling of Combined Discrete Event and Continuous Time Dynamical Systems." In *Proceedings of the 12th World Congress of Automatic Control.*

ANDERSSON, M. (1993b): "OmSim and Omola Tutorial and User's Manual." Technical Report TFRT-7504, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅRZÉN, K.-E. (1992): "A model-based control system concept." Report ISRN LUTFD2/TFRT--3213--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ASPEN (1982): *ASPEN PLUS Introductory Manual.* Aspen Technology, Inc., Cambridge, MA.

ÅSTRÖM, K. J. (1987): "Implementation of PID regulators." Report TFRT-7344, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

AUGUSTIN, C. D. C., M. S. FINEBERG, B. B. JOHNSON, R. N. LINEBARGER, F. J. SANSON, and J. C. STRAUSS (1967): "The SCi Continuous System Simulation Language (CSSL)." *Simulation*, **9**, pp. 281–303.

BARKER, H., M. CHEN, P. GRANT, C. JOBLING, and P. TOWNSEND (1993): "Open Architecture for Computer-Aided Control Engineering." *Control Systems*, **13:2**, pp. 17–27.

BARTON, P. and C. PANTELIDES (1991): "The Modelling and Simulation of Combined Discrete/Continuous Processes." In *Proceedings from Process System Engineering '91*, Montebello, Canada.

BIEGLER, L. (1989): "Chemical Process Simulation." *Chemical Engineering Progress*, October, pp. 50–61.

BRISTOL, E. H. (1980): "After DDC: Idiomatic Control." *AIChE Philadelphia*, May.

CELLIER, F. and H. ELMQVIST (1993): "Automated Formula Manipulation Supports Object-Oriented Continuous-System Modeling." *IEEE Control Systems Magazine*, **13:2**, pp. 28–38.

CELLIER, F. E. (1991): *Continuous System Modelling*. Springer-Verlag.

CZULEK, A. (1988): "An Experimental Simulator for Batch Chemical Processes." *Computers & Chemical Engineering*, **12:2/3**, pp. 253–259.

DAVID, R. and H. ALLE (1992): *Petri Nets and Grafcet*. Prentice Hall.

DOUGLAS, J. (1988): *Conceptual Design of Chemical Processes*. McGraw-Hill.

ELMQVIST, H. (1975): "SIMNON – An interactive simulation program for nonlinear systems." Technical Report TFRT-3091, Dept of Automatic Control, Lund Institute of Technology, Sweden.

ELMQVIST, H. (1978): *A Stuctured Model Language for Large Continuous Systems*. Phd thesis, Lund Institute of Technology.

ELMQVIST, H. (1993): "Object-Oriented Modeling and Automatic Formula Manipulation in Dymola." In *Proceedings of Scandinavian Simulation Society*, Kongsberg, Norway.

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

EVANS, L. B. (1990): "Process Modelling: What Lies Ahead." *Chemical Engineering Progress*, pp. 42–44.

FAGLEY, J. C. and B. CARNAHAN (1990): "The Sequential-clustered method for Dynamic Chemical Plant Simulation." *Computers & Chemical Engineering*, **14:2**, pp. 161–177.

FISHWICK, P. (1992): "An Integrated Approach to System Modelling using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies." *ACM Transactions on Modelling and Computer Simulation*, **2:4**, pp. 307–330.

FISHWICK, P. (1993): "A Simulation Environment for Multimodeling." *Discrete Event Dynamic System: Theory an Applications.*

GENSYM (1992): *G2 Reference Manual, version 3.0.* Gensym Corp., Cambrigde, MA.

HILLESTAD, M. and T. HERTZBERG (1988): "Convergence and Stability of the Sequential Modular Approach to Dynamic Process Simulation." *Computers & Chemical Engineering*, **12:5**, pp. 407–414.

HURSON, A., S. PAKZAD, and J. CHENG (1993): "Object-Oriented Database Management Systems: Evaluation and Performance issues." *Computer*, **26:2**, pp. 48–60.

INTELLICORP (1987): *KEE Software Development System, User's Manual.* IntelliCorp.

KHEIR, N. (1988): *Systems Modeling and Computer Simulation.* Marcel Dekker, Inc.

KREUTZER, W. (1986): *System Simulation – Programming styles and languages.* Addison-Wesley.

KRÖNER, A., P. HOLL, W. MARQUARDT, and E. D. GILLES (1990): "DIVA - an Open Architecture for Dynamic Simulation." *Computers & Chemical Engineering*, **14:11**, pp. 1289–1295.

LJUNG, L. and T. GLAD (1991): *Modellbygge och Simulering (Modelling and Simulation).* Studentlitteratur.

LUND, P. C. (1992): *An Object-Oriented Environment for Process Modelling and Simulation.* PhD thesis 50, the Norwegian Institute of Technology.

LUYBEN, W. L. (1973): *Process Modeling, Simulation, and Control for Chemical Engineers.* McGraw-Hill.

MARQUARDT, W. (1991): "Dynamic Process Simulation—Recent Progress and Future Challanges." In RAY and ARKUN, Eds., *Fourth International Conference on Chemical Process Control*, South Padre Island, Texas.

MARQUARDT, W. (1993): "An Object-oriented Representation of Structured Process Models." In *European Symposium on Computer Aided Process Engineering – 1.*

MARQUARDT, W., P. HOLL, and E. GILLES (1987): "Dynamic Process Flowsheet Simulation—an Important Tool in Process Control." In *Proceedings of the 10th World Congress on Automatic Control*, volume 2, pp. 374–379, Munich, Federal Republic of Germany. International Federation of Automatic Control.

MATHWORKS (1991): *SIMULINK a program for simulating dynamic systems, User's Guide*. The Math Works Inc., Cochituate Place, 24 Prime Park Way, Natick, MA 01760.

MATTSSON, S. E. (1988): "On model structuring concepts." In *Preprints 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, pp. 269–274, P. R. China.

MATTSSON, S. E. (1989): "Modeling of interactions between submodels." In IAZEOLLA *et al.*, Eds., *Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation*, 1989 European Simulation Multiconference, Rome, pp. 63–68. The Society for Computer Simulation International.

MATTSSON, S. E., Ed. (1989): "New tools for model development and simulation. Proceedings of a full-day seminar, Stockholm, 24 October 1989." Report TFRT-7438, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S. E. and M. ANDERSSON (1992): "The Ideas Behind Omola." In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design*, pp. 23–29.

MATTSSON, S. E., M. ANDERSSON, and K. J. ÅSTRÖM (1993): "Object-Oriented Modelling and Simulation." In LINKENS, Ed., *CAD for Control Systems*, pp. 31–69. Marcel Dekker, Inc.

MINSKY, M. (1965): "Models, Minds, machines." In *Proceedings IFIP Congress*, pp. 45–49.

MITCHELL, E. E. L. and J. S. GAUTHIER (1986): *ACSL: Advanced Continuous Simulation Language—User's Guide and Reference Manual.* Mitchell & Gauthier Assoc., Concord, Mass.

NILSSON, B. (1987): "Experiences of describing a distillation column in some modelling languages." Report TFRT-7362, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1989a): *Structured Modelling of Chemical Processes—An Object-Oriented Approach.* Lic Tech thesis TFRT-3203, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1989b): "Structured Modelling of Chemical Processes with Control Systems." In *AIChE annual meeting 1989, 5-10 november, San Fransisco, CA.*

NILSSON, B. (1990): "Object-oriented modelling of a controlled chemical process." In *Preprints 11th IFAC World Congress*, volume 10, pp. 22–27, Tallinn, Estonia.

NILSSON, B. (1991): "En on-linesimulator för operatörsstöd," (An on-line simulation for operator support). Report TFRT-3209, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1992): "Object-oriented chemical process modelling in Omola." In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design, CADCS '92*, Napa, California.

NILSSON, B. (1993a): "A Chemical Plant Model in Omola." Technical Report ISRN LUTFD2/TFRT--7507--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1993b): "Modeling Process Control System in Omola." In *Proceedings 12th World Congress International Federation of Automatic Control*, volume VI, pp. 197–200, Sydney, Australia.

NILSSON, H. (1991): "Implementation of Petri-net and Grafcet primitives in Omola and modelling of Markov-processes." Master thesis TFRT-5452, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

PANTELIDES, C. (1988): "SPEEDUP - Recent Advances in Process Simulation." *Computers & Chemical Engineering*, **12:7**, pp. 745–755.

PANTELIDES, C. and P. BARTON (1992): "Equation-oriented Dynamic Simulation current status and future perspectives." In *Proceedings of European Symposium on Computer Aided Process Engineering-2.*

PERKINS, J. D. (1986): "Survey of existing Systems for the Dynamic Simulation of Industrial Processes." *Modeling, Identification and Control*, **7:2**, pp. 71–81.

PERKINS, J. D. and R. SARGENT (1982): "SPEEDUP: A Computer Program for Steady-State and Dynamic Simulation and Design of Chemical Processes." *AIChE Symposium Series*, **78:214**, pp. 1–11.

PETTI, T. F. and P. S. DHURJATI (1991): "Object-Based Automated Fault Diagnosis." *Chemical Engineering Communications*, **102**, pp. 107–126.

PETZOLD, L. (1982): "A description of DASSL: a differential-algebraic equation solver." In *Proceedings of IMACS World Congress*, Montreal, Canada.

PIELA, P. (1989): *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.

PIELA, P., T. EPPERLY, K. WESTERBERG, and A. WESTERBERG (1991): "ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis: the Modeling Language." *Computers & Chemical Engineering*, **15:1**, pp. 53–72.

PONTON, J. W. and P. J. GAWTHROP (1991): "Systematic Construction of Dynamic Models for Phase Equilibrium Processes." *Computers & Chemical Engineering*, **15:12**, pp. 803–808.

SARGENT, R. W. H. and A. W. WESTERBERG (1964): "SPEED-UP in Chemical Engineering Design." *Transaction Institute in Chemical Engineering*, **42**, pp. 190–197.

SHINSKEY, F. G. (1987): *Controlling Multivariable Processes.* Instrument Society of America, 3 edition.

SØRLIE, C. F. (1990): *A Computer Environment for Process Modelling.* PhD thesis 12, the Norwegian Institute of Technology.

STEFIK, M. and D. G. BOBROW (1984): "Object-Oriented Programming: Themes and Variations." *The AI Magazine*, **6:4**, pp. 40–62.

STEPHANOPOULOS, G., G. HENNING, and H. LEONE (1990a): "MODEL.LA. A Modeling Language for Process Engineering—I. The Formal Framework." *Computers & Chemical Engineering*, **14:8**, pp. 813–846.

STEPHANOPOULOS, G., G. HENNING, and H. LEONE (1990b): "MODEL.LA. A Modeling Language for Process Engineering—II. Multifaceted Modeling of Processing Systems." *Computers & Chemical Engineering*, **14:8**, pp. 847–869.

STEPHANOPOULOS, G., J. JOHNSTON, T. KRITICOS, and R. LAKSHMANAN (1987): "DESIGN-KIT: An Object-oriented Environment for Process Engineering." *Computers & Chemical Engineering*, **11:6**, pp. 655–674.

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

TELNES, K. (1992): *Computer Aided Modeling of Dynamic Processes based on Elementary Physics.* PhD thesis 47, the Norwegian Institute of Technology.

VOGEL, E. F. (1991): "An Industrial Perspective on Dynamic Flowsheet Simulation." In RAY and ARKUN, Eds., *Fourth International Conference on Chemical Process Control*, South Padre Island, Texas.

WESTERBERG, A. W. and D. R. BENJAMIN (1985): "Thoughts on a Future Equation-oriented Flowsheeting System." *Computers & Chemical Engineering*, **9:5**, pp. 517–526.

WINSTON, P. H. and B. K. P. HORN (1984): *Lisp.* Addison-Wesley.

WOODS, E. (1993): *The Hybrid Phenomena Theory.* PhD thesis 60, the

Norwegian Institute of Technology.

WOZNY, G., W. GUNTERMUTH, and W. KOTHE (1992): "CAPE in der Verfahrenstechnik aus industriller Sicht – Status, Bedarf, Prognose oder Vision?" *Chemie-Ingenieur-Technik*, **64:8**, pp. 693–699.

A

Aspen Plus

This appendix breifly introduce a flowsheet simulation packages called ASPEN PLUS. Below follows an example found in [Aspen, 1982]. It is a minor process where two feed streams are mixed together with two recycle streams to produce a reactor feed, see Figure A.1. In the reactor benzene and hydrogen react to produce cyclohexane. The reactor out flow enters a flash where vapor and liquid are separated. Some of the vapor is recyled and some is purged. The liquid is also split into one recyle stream and one column feed. The column separates light components and the desired product of cyclohexane. The input file to ASPEN PLUS is found in Listing A.1. It begins with a title and a textual description. These are just comments and used to make output comments. The problem specifications begin with the definition of engineering units and continue with

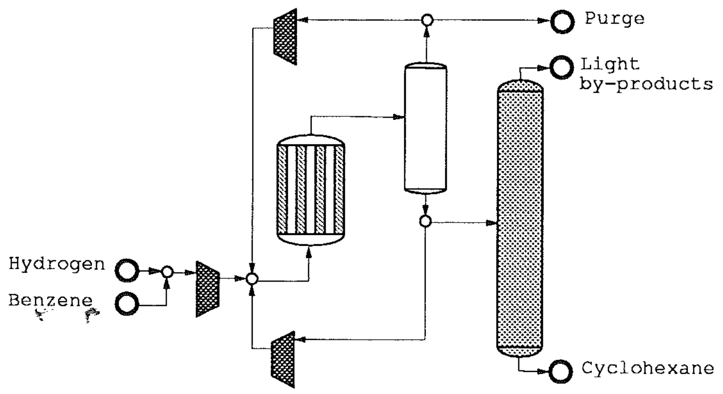


Figure A.1 Flowsheet of the process for hydrogenation of benzene to cyclohexane.

```
TITLE 'HYDROGENATION OF BENZENE TO CYCLOHEXANE'
DESCRIPTION "THIS IS A MODEL OF A PROCESS FOR PRODUCING
             CYCLOHEXANE BY HYDROGENATION OF BENZENE."
IN-UNITS ENG
OUT-UNITS ENG
COMPONENTS H2 HYDROGEN/ N2 NITROGEN/ C1 METANE/ BZ BENZENE/
           CH CYCLOHEXANE
PROPERTIES SYSOP3
FLOWSHEET
  BLOCK FEED-MIX IN=H2IN BZIN H2RCY CHRCY OUT=RXIN
  BLOCK REACT    IN=RXIN                  OUT=RXOUT
  BLOCK HP-SEP   IN=RXOUT                 OUT=VAP     LIQ
  BLOCK V-FLOW   IN=VAP                   OUT=PURGE   H2RCY
  BLOCK L-FLOW   IN=LIQ                   OUT=COLFD   CHRCY
  BLOCK COLUMN   IN=COLFD                 OUT=LTENDS PRODUCT
STREAM H2IN TEMP=120 PRES=335 MOLE-FLOW=300
            MOLE-FRAC H2 0.975/ N2 0.005/ C1 0.02
STREAM BZIN TEMP=100 PRES=15  MOLE-FLOW=100
            MOLE-FRAC BZ 1
BLOCK FEED-MIX HEATER
  PARAM TEMP=300 PRES=330
BLOCK REACT RSTOIC
  PARAM TEMP=400 PRES=-15
  STOIC 1 MIXED BZ -1/ H2 -3/ CH 1
  CONV 1 MIXED BZ 0.998
BLOCK HP-SEP FLASH2
  PARAM TEMP=120 PRES=-5
BLOCK V-FLOW FSPLIT
  FRAC PURGE 0.08
BLOCK L-FLOW FSPLIT
  FRAC CHRCY 0.3
BLOCK COLUMN RADFRAC
  PARAM NSTAGE=15
  FEEDS COLFD 8
  PRODUCTS LTENDS 1 V/PRODUCT 15 L
  P-SPEC 1 200
  COL-SPECS RDV=1 RR=1.2 B=99
  VARY 1 B 97 101
  SPEC 1 MOLE-RECOV .9999 STREAMS=PRODUCT COMPS=CH
```

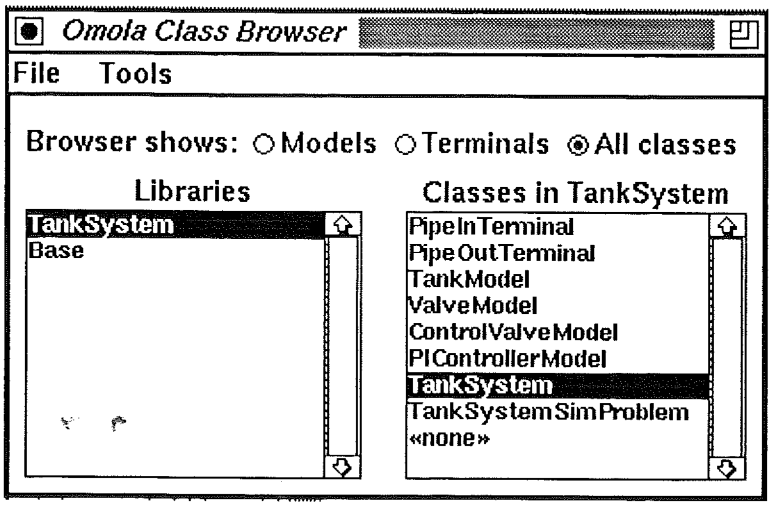
Listing A.1 Listing of the ASPEN PLUS input file of the example.

a medium description. The user only has to specify the chemical components and what kind of termodynamic property description to be used. It is all encapzulated into one property assignment. After this the flowsheet topology is specified. Before the definition of the processing blocks in the flowsheet the feed streams of hydrogen and benzene are specified. The block are predefined units found in the library. They have predefined parameters that can be set, as seen above.

B

Introductory Omola Example

The introductory tank system example in Chapter 3 is described in more detail in this appendix. The library containing all classes in the tank system example is listed in the following section. It is also listed in the model database browser in Figure 9.3. The tank system library contains two terminal classes, four primitive models and two composite models. The primitive models describe a tank, static valve, control valve and PI



Browser shows: ○ Models ○ Terminals [✓ All classes]

Libraries
TankSystem
Base

Classes in TankSystem
PipeInTerminal
PipeOutTerminal
TankModel
ValveModel
ControlValveModel
PIControllerModel
TankSystem
TankSystemSimProblem
«none»

Figure B.1 The tank system library listed in the OmSim model data base browser.

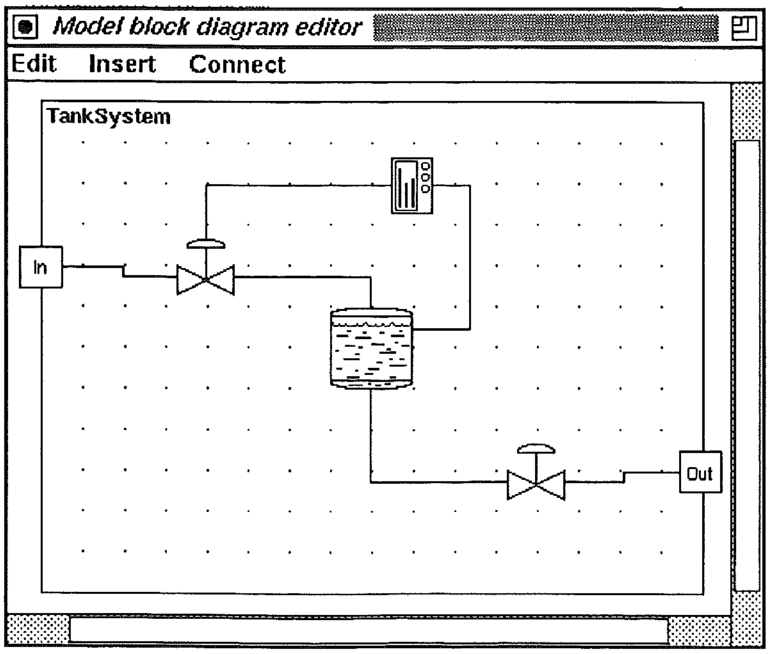


Figure B.2 The tank system model in the graphical model editor, MED.

controller. The composite models is the tank system model and a subclass defining the simulation problem. Compared to the Omola classes in Section 3.1 these library classes also contain graphical descriptions. The graphical description is described in the `Graphic` attribute. It contains information about the icon position (xpos, ypos) and the icon bitmap. The terminal icons can be made invisible. The bitmap description is developed by an ordinary bitmap editor. The bitmap is not a part of the model database and the bitmap attribute is interpret as a file name. The icons are seen in graphical editor in Figure B.2.

```
LIBRARY TankSystem;
%% A library with the introductory tank
%% system example in Chapter 3.

PipeInTerminal ISA Base::RecordTerminal WITH
  Flow ISA Base::ZeroSumTerminal WITH
    direction := 'In;
  END;
  Pressure ISA Base::SimpleTerminal;
END;

PipeOutTerminal ISA TankSystem::PipeInTerminal WITH
  Flow ISA Base::ZeroSumTerminal WITH
    direction := 'Out;
  END;
END;

TankModel ISA Base::Model WITH
icon:
  Graphic ISA super::Graphic WITH
```

```
      bitmap TYPE String := "icontank";
    END;
  terminals:
    In ISA TankSystem::PipeInTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 200.0;
        y_pos := 300.0;
        invisible := 1;
      END;
    END;
    Out ISA TankSystem::PipeOutTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 200.0;
        y_pos := 0.0;
        invisible := 1;
      END;
    END;
    Level ISA Base::SimpleTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 400.0;
        y_pos := 225.0;
        invisible := 1;
      END;
    END;
  parameters:
    Density ISA Base::Parameter;
    GravConst ISA Base::Parameter;
    TankArea ISA Base::Parameter;
    PipeArea ISA Base::Parameter;
  variable:
    mass ISA Base::Variable;
  equations:
    mass' = Density*(In.Flow - Out.Flow);
    mass = Density*TankArea*Level;
    Density*GravConst*Level =
      Out.Pressure + Density*abs(Out.Flow)*Out.Flow/PipeArea^2/2;
    In.Pressure = 0.0;
  END;

  ValveModel ISA Base::Model WITH
  icon:
    Graphic ISA super::Graphic WITH
      bitmap TYPE String := "iconvalve";
    END;
  terminals:
    In ISA TankSystem::PipeInTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 0.0;
        y_pos := 100.0;
        invisible := 1;
      END;
  END;
  Out ISA TankSystem::PipeOutTerminal WITH
    Graphic ISA super::Graphic WITH
      x_pos := 400.0;
      y_pos := 100.0;
      invisible := 1;
    END;
  END;
parameters:
  PressDrop ISA Base::Parameter;
  Density ISA Base::Parameter;
  PipeArea ISA Base::Parameter;
equation:
  In.Flow = Out.Flow;
  PressDrop*(In.Pressure - Out.Pressure) =
    Density*abs(In.Flow)*In.Flow/PipeArea^2/2;
END;

ControlValveModel ISA TankSystem::ValveModel WITH
  Graphic ISA super::Graphic;
terminal:
  Control ISA Base::SimpleTerminal WITH
    Graphic ISA super::Graphic WITH
      x_pos := 200.0;
      y_pos := 300.0;
      invisible := 1;
    END;
  END;
parameter:
  ValvePar ISA Base::Parameter;
variable:
  PressDrop ISA Base::Variable;
  Position ISA Base::Variable;
equations:
  PressDrop = ValvePar*Position;
  Position = if Control < 0 then 0
             else if Control < 1 then Control
             else 1;
END;

PIControllerModel ISA Base::Model WITH
icon:
  Graphic ISA super::Graphic WITH
    bitmap TYPE String := "iconpireg";
  END;
terminal:
  Measure ISA Base::SimpleTerminal WITH
    Graphic ISA super::Graphic WITH
      x_pos := 400.0;
      y_pos := 150.0;
      invisible := 1;
      END;
    END;
    Control ISA Base::SimpleTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 0.0;
        y_pos := 150.0;
        invisible := 1;
      END;
    END;
  parameters:
    K ISA Base::Parameter;
    Ti ISA Base::Parameter;
    Tt ISA Base::Parameter;
    Ref ISA Base::Parameter;
  variables:
    e ISA Base::Variable;
    p ISA Base::Variable;
    i ISA Base::Variable;
    u ISA Base::Variable;
  equations:
    e = Ref - Measure;
    p = K*e;
    i' = K*e/Ti + (u - Control)/Tt;
    u = p + i;
    Control = if u < 0 then 0 else if u < 1 then u else 1;
  END;

  TankSystem ISA Base::Model WITH
  icon:
    Graphic ISA super::Graphic;
  terminals:
    In ISA TankSystem::PipeInTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 0.0;
        y_pos := 200.0;
      END;
    END;
    Out ISA TankSystem::PipeOutTerminal WITH
      Graphic ISA super::Graphic WITH
        x_pos := 400.0;
        y_pos := 75.0;
      END;
    END;
  submodels:
    Tank ISA TankSystem::TankModel WITH
      Graphic ISA super::Graphic WITH
        x_pos := 200.0;
        y_pos := 150.0;
      END;
    END;
    InflowValve ISA TankSystem::ControlValveModel WITH
```

```
      Graphic ISA super::Graphic WITH
        x_pos := 100.0;
        y_pos := 200.0;
      END;
    END;
    OutflowValve ISA TankSystem::ValveModel WITH
      Graphic ISA super::Graphic WITH
        x_pos := 300.0;
        y_pos := 75.0;
      END;
    END;
    PI ISA TankSystem::PIControllerModel WITH
      Graphic ISA super::Graphic WITH
        x_pos := 225.0;
        y_pos := 250.0;
      END;
    END;
  connections:
    C1 ISA Base::Connection WITH
      In AT InflowValve.In;
      bpoints TYPE Matrix [4, 2] :=
        [0, 199; 49, 199; 49, 193; 82, 193];
    END;
    C2 ISA Base::Connection WITH
      InflowValve.Out AT Tank.In;
      bpoints TYPE Matrix [3, 2] := [116, 193; 199, 193; 199, 174];
    END;
    C3 ISA Base::Connection WITH
      Tank.Out AT OutflowValve.In;
      bpoints TYPE Matrix [3, 2] := [199, 125; 199, 68; 282, 68];
    END;
    C4 ISA Base::Connection WITH
      Out AT OutflowValve.Out;
      bpoints TYPE Matrix [4, 2] :=
        [316, 68; 352, 68; 352, 74; 399, 74];
    END;
    C5 ISA Base::Connection WITH
      Tank.Level AT PI.Measure;
      bpoints TYPE Matrix [4, 2] :=
        [224, 161; 259, 161; 259, 249; 236, 249];
    END;
    C6 ISA Base::Connection WITH
      PI.Control AT InflowValve.Control;
      bpoints TYPE Matrix [3, 2] := [212, 249; 99, 249; 99, 216];
    END;
  END;

  TankSystemSimProblem ISA TankSystem::TankSystem WITH
  parameters:
    D ISA Base::Parameter WITH
      default := 1000;
    END;
    PA ISA Base::Parameter WITH
      default := 0.010;
    END;
  parameter_assignments:
    In.Pressure := 10000;
    Out.Pressure := 0;
    InflowValve.Density := D;
    InflowValve.PipeArea := PA;
    InflowValve.ValvePar.default := 0.40;
    Tank ISA super::Tank WITH
      Density := D;
      PipeArea := PA;
      TankArea.default := 1;
      GravConst := 9.810;
      mass.initial := 1000;
    END;
    PI ISA super::PI WITH
      K.default := 1;
      Ti.default := 50;
      Tt.default := 100;
    END;
    OutflowValve.PressDrop.default := 0.20;
    OutflowValve.Density := D;
    OutflowValve.PipeArea := PA;
  event:
    Init, Step1, Step2 ISA Base::Event;
  initial_condition_calculations:
    ONEVENT Init DO
      new(Pi.Ref) := 1;
      new(Tank.Level) := Tank.mass/(D*Tank.TankArea);
      new(Tank.Out.Pressure) := Tank.GravConst*Tank.mass /
        (Tank.TankArea*(1 + OutflowValve.PressDrop));
      new(Tank.Out.Flow) := PA*sqrt(
        abs(2*OutflowValve.PressDrop*new(Tank.Out.Pressure)/D));
      new(InflowValve.In.Flow) := new(Tank.Out.Flow);
      new(InflowValve.PressDrop) :=
        D*abs(new(InFlowValve.In.Flow))*new(InFlowValve.In.Flow) /
        (PA^2*2*(InFlowValve.In.Pressure-InFlowValve.Out.Pressure));
      new(Pi.i) := new(InFlowValve.PressDrop)/InFlowValve.ValvePar;
      schedule(Step1,10);
    END;
  step_change:
    ONEVENT Step1 DO
      new(Pi.Ref) := 1.1;
      schedule(Step2,250);
    END;
    ONEVENT Step2 DO
      new(Pi.Ref) := 1;
    END;
  END;
```

C

Front Cover Example

The figure on the front cover is also found in Figure C.1 with additional textual expanation. The figure illustrate the structure hierarchy guidelines together with the class hierarchy guidelines discussed in Chapter 4.

The plant structure is decomposed into five levels; `plant - plant section - unit - subunit - subsubunit`. The plant structure is described in the bottom in Figure C.1 with the plant on the left to the subsubunits on the right. The plant is decomposed into three plant sections: pretreatment, reaction and separation. The separation plant section is composed of three distillation units in series. The third unit is further decomposed into subunits. One subunit is the reboiler which is decomposed into boiler side, heat transfer wall and steam side. The boiler side part is medium and machine decomposed.

The class hierarchy is described from the top to the bottom. The class hierarchy is decomposed into six levels; `application - granularity - unit type - interface class - model class`. The application class in the process class found in the top, below `Model` class. The next level is the granularity classes, called flowsheet, unit, subunit and medium classes in Figure C.1. The next level is the unit type level, where the class tree is branched into unit specific classes. The fourth level is the interface class level where interfaces to model classes are defined and following level is the actual model class definition level.

The subclasses to the interface classes can be called polymorphic models and they can be used in the same context. Examples of polymorphic

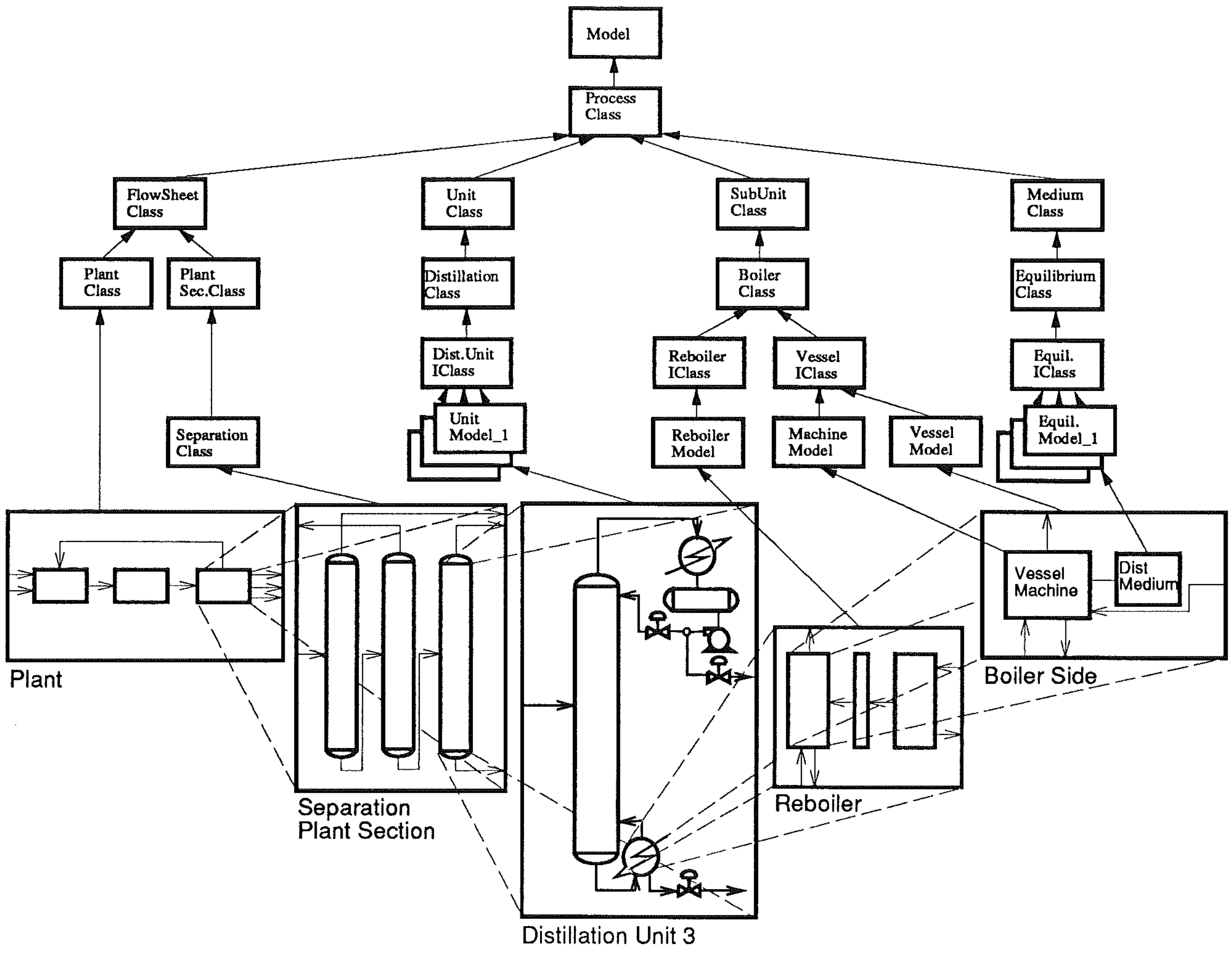


Figure C.1 The figure from the front cover illustration structure and class hierarchies in a chemical plant example.

models in Figure C.1 are the equilibrium models, e.g., `EquilModel_1`. and the distillation unit models, e.g., `UnitModel_1`. This is discussed in Chapter 4.

The boiler side is medium and vessel machine decomposed. This decomposition method is discussed in Chapter 5 and it makes it possible to create independent libraries of unit machines and medium descriptions.