MANAGEMENT OF CONTINUOUS SYSTEM MODELS IN DEVS-SCHEME: TIME WINDOWS FOR EVENT BASED CONTROL

by

Qingsu Wang

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfilment of the Requirements

For the Degree of

MASTER OF SCIENCE

WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 8 9

STATEMENT BY AUTHOR

SIGNED: [illegible]

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

[illegible] November 28, 1989

F. E. Cellier
Associate Professor of
Electrical and Computer Engineering

Date

ACKNOWLEDGMENTS

This work has been completed under the supervision of Dr. Francois E. Cellier, my thesis advisor. I wish to express my great appreciation to him for his invaluable guidance, support, help, and encouragement. I am also very grateful to Dr. Bernard P. Zeigler and Dr. Jerzy W. Rozenblit for their many helpful suggestions during the research, and their reviewing of this thesis.

I am deeply indebted to my parents and my younger sister, whose love and support inspired me to put forth the effort necessary to reach this degree.

I also wish to acknowledge my sincere gratitude to many other people for their discussion of the research, improvement of the writing of the thesis, and moral support.

This research has been supported by NASA-Ames Cooperative Agreement No. NCC 2-525, "A Simulation Environment for Laboratory Management by Robot Organizations".

CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis describes the design and implementation of an extended knowledge-based modeling and simulation environment, in which the management of continuous-time models in DEVS-SCHEME is realized to meet the requirements of the modeling and simulation of a robot-managed laboratory aboard the forthcoming Space Station Freedom.

The modular hierarchical modeling scheme is preserved in the continuous models by using DYMOLA, a continuous modeling language, as a bridge between the abstracted DEVS models and the continuous simulation language code (in DESIRE). Through operations on the System Entity Structure (SES), a knowledge representation scheme, models at different granularity levels can be generated.

Time-windows can be obtained by manipulating a pruned SES. These time windows can be used to automatically generate a discrete-event model which, at a coarser granularity, exhibits a behavior equivalent to the original continuous-time model.

Therefore, an event-based intelligent control strategy can be realized in this knowledge-based multi-facetted modeling environment. Continuous-time and discrete-event modeling and simulation can be merged with AI techniques.

CHAPTER 1
INTRODUCTION

With the rapid advance of Space technology, a variety of scientific experiments have been carried on in Space to overcome limitations of processes conducted on Earth. The experiments performed during Space Shuttle missions in the past were largely "canned experiments", in that scientists on the ground had little control over the progress of their experiments. The forthcoming Space Station Freedom (SSF) is different from the Space Shuttle in this respect, because the SSF is a multiple purpose user facility with a planned life-time of greater than 20 years. Space-based laboratories or other facilities on the SSF require that many of the functions be performed by intelligent robots, since man-power is limited and very expensive. Astronauts in Space are very valuable resources, who should be used to the best advantage. Automation and robots will enable the astronauts to function more productively in Space by freeing them from tedious, repetitive chores so that they can concentrate on congnitive tasks requiring humans.

Robots in today's market are primarily applied in the "heavier" manufacturing areas, such as automobiles and electric machinery, or in hazardous working environments. The capabilities of these robots are limited by their prestored control strategies. They are not smart enough to act or react flexibly. One kind of robot, called a *teleoperator*, operates in a tightly coupled mode with a human operator. Such robots may have computers in their control loops to accomplish remote control or to enable the human operator to sense the environment of the robot. Theoretically, they are capable of running autonomously, but they are not designed to operate without a human in the control loop. On Earth, these systems work well. However, for tasks aboard the SSF, the long communication time delays (the SSF project managers talk about two seconds round-trip delay time, but those numbers have not yet been confirmed) for the control signal to pass between the human operator on Earth and the robot aboard the SSF will make the overall system unstable.

In order to fulfil tasks in Space, a robot must know its surroundings, its own duty, its limitations, etc., and must be able to communicate with other robots. Thus, intelligent robots are coming to be needed to satisfy customers in Space. A simulation environment capable of supporting the design of robot organizations for managing laboratories aboard the SSF is under development in our Department (Zeigler *et al.*, 1988). The ultimate goals of the research are to employ a simulation environment to develop robot cognitive system strategies for effective multi-robot management of laboratory experiments. The research is based on a knowledge-based simulation environment called DEVS-SCHEME (Zeigler, 1986; Kim 1988), which combines discrete-event modeling and simulation with AI knowledge representation schemes. Robot models and laboratory environments are being constructed on the basis of object-oriented and hierarchical models of robots and laboratory components at multiple levels of abstraction.

To achieve greater autonomy, a robot must have knowledge about the world around it. Using models is the main way to convey knowledge both in simulation and in artificial intelligence techniques. A robot recognizes his environment also by means of these models. Models are simplifications of reality. An important aspect of modeling involves levels of detail, that is, granularity. The goal of modeling drives the modeler to extract from the real system only those essential facts needed to have the model be a valid basis for the decision making in question. This can easily be reflected by the traffic control field problem. One may study the problem at three levels of detail:

"a) macroscopic models which are entirely continuous and in which the single vehicle is not considered but only the traffic flow with accompanying densities

b) microscopic models which are entirely discrete with vehicles entering the considered system, traveling through it in discrete steps, e.g., from one corner to the next with queuing situations in front of traffic lights, etc.

c) submicroscopic models (not often used in this context) which are continuous with specific discontinuities describing the dynamic behavior of every single vehicle (Cellier, 1976)"

It is shown from this example that different modeling and simulation techniques can be selected for the same physical process. Even though several simulation languages today, such as GASP-V, SYSMOD, ect., have the feature of combining discrete-event models and continuous models, it is useful to search for a way that helps in switching automatically from a model of one granularity to another model of another granularity in order to meet the given objectives.

Research work done in this thesis focuses on the enhancement of DEVS-SCHEME to the manipulation of continuous system models, by which the automatic transformation of continuous models into equivalent discrete-event models can be realized. An existing software CESS (Controlled Environment Simulation System) is capable of managing continuous models in DEVS-SCHEME (Zhang, 1988; Kim *et al.,* 1988). However, CESS deals only with a special continuous simulation system, TRNSYS, which is well suited for the analysis of energy systems. Now, a continuous system modeling language DYMOLA (Elmqvist, 1979, 1981) has been selected as an interface between discrete event simulation models and continuous simulation languages. This approach is chosen mainly for two reasons: DEVS-SCHEME should be able to handle models of general purpose continuous simulation languages; on the other hand, real modularity of continuous models is required for the hierarchical modeling concept.

In DEVS-SCHEME, the continuous system models have now become subclasses of discrete event models. These models provide the knowledge level descriptions of the continuous models and serve as pointers to corresponding continuous models that are stored in a continuous system model base. The realization of modular and hierarchical modeling of continuous systems is captured by DYMOLA. The real continuous simulation run takes place in either DESIRE or SIMNON, two directly executing simulation languages. The DYMOLA preprocessor acts as a program generator that can alternatively generate either DESIRE or SIMNON programs. A knowledge representation scheme, the System Entity Structure (SES), is employed for the organization of all the models at different levels of granularity. The management of these models can be carried out by operations on SES. Simulation trajectories will be produced to form the time information which can then be automatically mapped to DEVS discrete-event models for future reference.

The following tasks have been done to achieve these objectives:

1. Migration of DYMOLA (originally coded in VAX PASCAL) from VAX/VMS to PC (coded in TURBO PASCAL);
2. Enhancement of DYMOLA to generate DESIRE models; previous versions of DYMOLA were able to generate SIMNON models only;
3. Enhancement of DYMOLA to generate DESIRE executable files;
4. Some minor enhancements of DYMOLA itself;
5. Creation of continuous classes in DEVS;
6. Defining methods, functions, and macros for the manipulation of continuous models in DEVS; and

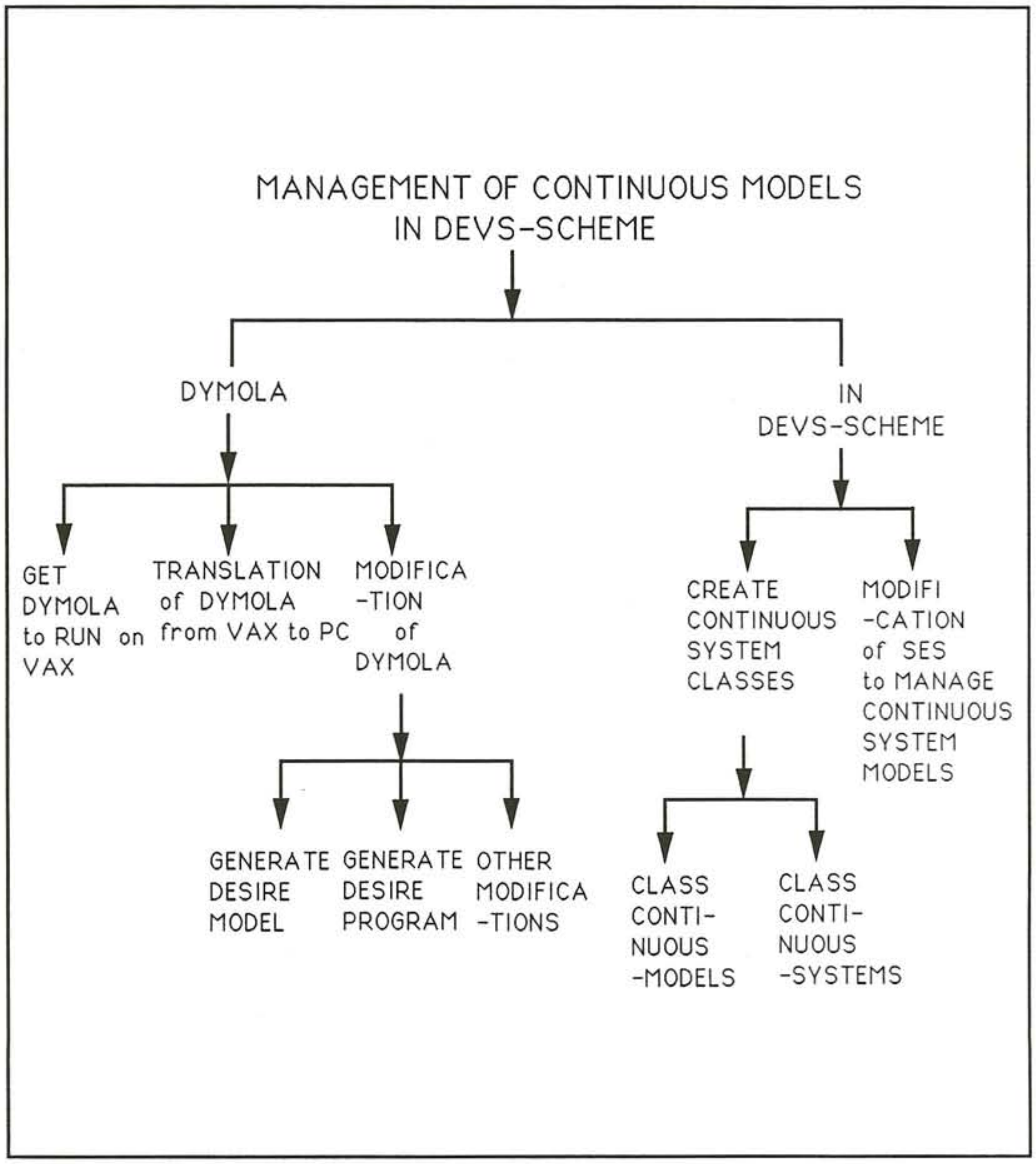

MANAGEMENT OF CONTINUOUS MODELS IN DEVS-SCHEME
DYMOLA
IN DEVS-SCHEME
GET DYMOLA to RUN on VAX
TRANSLATION of DYMOLA from VAX to PC
MODIFICA-TION of DYMOLA
CREATE CONTINUOUS SYSTEM CLASSES
MODIFI-CATION of SES to MANAGE CONTINUOUS SYSTEM MODELS
GENERATE DESIRE MODEL
GENERATE DESIRE PROGRAM
OTHER MODIFICA-TIONS
CLASS CONTI-NUOUS-MODELS
CLASS CONTI-NUOUS-SYSTEMS

Fig. 1.1 Entity Structure of the Project Plan

7. Enhancement of the SES to manage continuous models.

The resulting enhanced implementation environment runs on IBM PC compatibles under MS-DOS. In Fig. 1.1, an SES visualizes the development of the research work.

Hierarchical, modular specification of both discrete-event and continuous models is very important in modeling and simulation. Chapter 2 is intended to provide some theoretical background on these concepts, and the differences between discrete-event models and continuous models will be discussed there. To clarify the software environment, brief introductions of DEVS-SCHEME, DYMOLA, and DESIRE are given. In Chapter 3, the important role of DYMOLA in this research will be explained, and its major properties will be revealed. The two new classes in DEVS-SCHEME, named "continuous-models" and "continuous-systems", are to be presented in Chapter 4. The operations on SES, by which the management of continuous models can be realized at different knowledge levels of granularity, are described in detail in Chapter 5. The resulting knowledge-based modeling and simulation environment, which encompasses the management of the continuous models, will be presented there as well. Chapter 6 will illustrate the modeling and simulation of a fluid-handling laboratory, which demonstrates how, in DEVS-SCHEME, the SES is applied to manage continuous models of both discrete and continuous descriptions, and simulation trajectories are produced to form the time information needed by DEVS-SCHEME for further reference. In this case, the time-windows are produced for event-based intelligent control (Appendix 5). The experimental frame concept is also employed in this example. Chapter 7 concludes this thesis by summarizing the results and suggesting future research topics.

CHAPTER 2
THEORETICAL AND DESIGN CONSIDERATIONS

2.1 Modular Hierarchical Modeling and Model Base Concepts

Modularity and hierarchy are important properties in software system development. The same principles can be applied to model building processes.

Models may be developed as a set of independent modules. Each module is a single, well-defined component of the system. Such modules are said to be in *proper modular form* if they are invariant to changes in their inputs. Their interaction with the external world is through predefined input/output ports only. If several models are in proper modular form, a new coupled model can be created by specifying coupling relations between these models. The resulting coupled model is always in the proper modular form as well, and it can therefore itself be employed to construct yet larger models in the same manner. This property, called closure under coupling, enables hierarchical construction of models (Zeigler, 1987, 1989-b).

The importance of *modular hierarchical modeling* lies in the re-usability of its models. The re-usable models are stored in a *model base.* The models in the model base represent knowledge of the dynamic behavior of the system under study. New models can be saved in the model base, and saved models can be retrieved from it. Modular hierarchical modeling greatly facilitates the modeling process (Zeigler, 1989-b, 1987; Oren 1984; Standridge 1986; Kim 1988).

2.2 Experimental Frame

The experimental frame concept is a systematic approach to testing models. It is a set of circumstances under which a model or real system is to be observed and experimented with. The experimental frame specifies the input, output, control variables, and constraints required by the experimentation (Zeigler, 1984; Rozenblit, 1985).

The modular construction of models ensures that any model in the model base can be readily and independently tested by coupling an experimental frame to it. The hierarchical construction of models enables such testing at each stage of the hierarchy. This facilitates reliable and efficient verification of large simulation models.

2.3 The System Entity Structure

With the existence of the model bases, effective techniques for organizing and manipulating collections of component models are needed. The System Entity Structure (SES) is a knowledge representation scheme that combines the decomposition, taxonomic, and coupling relations of a system. SES provides the means to organize and generate possible configurations of a system. It also directs the synthesis of models in the model base (Zeigler, 1984, 1989-b).

The SES will be described further in Chapter 5.

2.4 Differences Between Discrete Event Systems and Continuous Systems

Continuous system simulation requires a step-by-step generation of successive model states, while discrete event system simulation computes states only at event times. For this reason, a discrete event model of a system can be simulated more efficiently than a continuous-time model of the same system (described by a set of differential equations). However, the discrete event model is usually a more aggregate representation of the system, and therefore, the trajectory behavior found by simulating such a model is usually a more crude representation of the real system's trajectory behavior than that found by simulating the continuous-time model.

Continuous models exchange information continuously, while discrete event models only exchange information occasionally (at event times). Therefore message passing is inefficient for continuous system simulation.

A system can be viewed differently according to the purpose of modeling. For instance, to design a higher level controller for a system, a discrete event model of this system will be more efficient; while in order to know when precisely the events occur, the complete trajectory is needed, thus a continuous model is preferred.

Depending on the purpose of the simulation study, a continuous model or a discrete model of the system may be better suited. In order to allow an easy transition between the two types of models, it is advantageous if both can be accessed through the same mechanism. Therefore, it was decided to make continuous models available within DEVS-Scheme.

However, simulation languages for continuous-time systems based on numerical integration methods have become well established since the 1960's. Plenty of research went into the design and development of such software systems. It seems therefore to be a waste of time to reimplement a complete continuous system simulation language in DEVS-Scheme unless such a reimplementation offers advantages that cannot be obtained by utilizing tools that already exist.

DEVS-SCHEME supports the modular hierarchical modeling concept. It requires a model with modular specification. Unlike in traditional simulation languages, a model in DEVS-SCHEME can be viewed as processing input and output ports, through which all interaction with the environment is mediated. In the discrete event case, events determine the values appearing on such ports. A systematic method to transform a non-modular discrete event model into a modular one is provided by Zeigler (1984).

For continuous models, the situation is quite different. The differences between the continuous and the discrete event models will be discussed below. It will also be shown that the current Continuous System Simulation Language (CSSL) specification languages (Augustin, 1967) do not provide means for a modular specification of continuous models, as DEVS-SCHEME does for discrete event models.

Discrete event models are directed graphs of hierarchies; continuous models are non-directed graphs of hierarchies. For example, Figure 2.1 and Figure 2.3 show the same RLC network with different inputs. Figure 2.2 and Figure 2.4 show models of this RLC network with these different inputs (Cellier, 1979). It is seen that, because of the different environment (connected to a voltage source or a current source), the input variable of the model must be changed (U_1 or I_1) in order to keep the differential equations in state-space form. As a result of this, the model descriptions are varied. Consequently, by looking at a topological description of a continuous model, connections between subsystems cannot be assigned directions.

In a block diagram representation, the direction of all paths is obviously determined. Block diagrams are directed graphs. It is possible to generate a block diagram representation for an electrical network. However, the required block diagram depends on the embedding of the subsystem, i.e. the same block diagram cannot be used in the two cases shown above. Block diagrams do not preserve the physical (topological) structure of the network. They are, in fact, only graphical representation of the mathematical (computational) structure of the system.

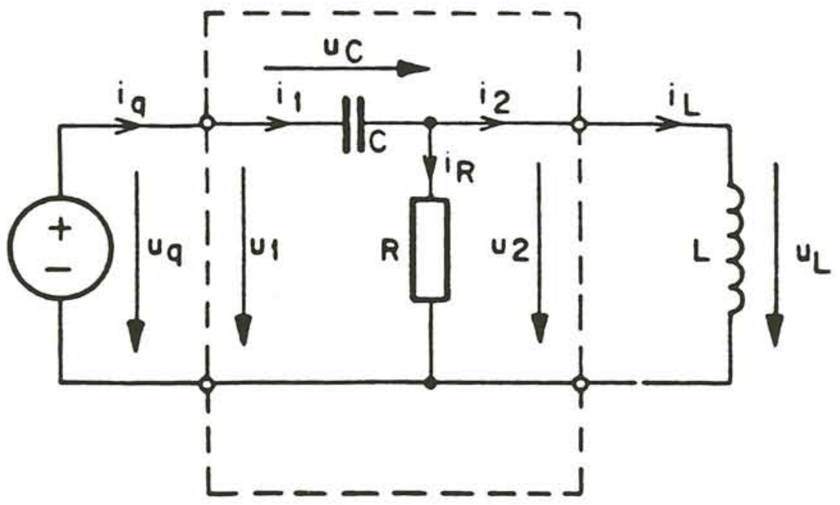


Fig. 2.1 RLC-network with voltage source

```
MACRO RC1 (U2, I1 <- U1, I2, R, C);
  MACVAR
    STATE UC;
    ALGEBR IR;
  MACCONTIN
    UC' = I1/C;
    I1 = I2 + IR;
    IR = U2/R;
    U2 = U1 - UC
  MACEND (* CONTINUOUS *)
MACEND (* RC1 *);
...
CONTINUOUS
  ...
  UQ = f(TIME);
  RC1 (UL, IQ <- UQ, IL, R, C);
  IL' = UL/L;
```

Fig. 2.2 Model of a RLC-network with voltage source

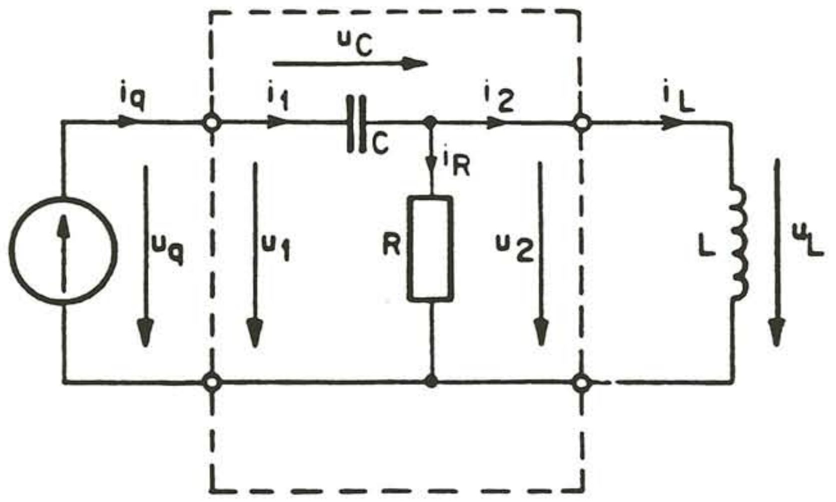


Fig. 2.3 RLC-network with current source

```
MACRO RC2 (U1, U2 <- I1, I2, R, C);
  MACVAR
    STATE UC;
    ALGEBR IR;
  MACCONTIN
    UC' = I1/C;
    IR = I1 - I2;
    U2 = R*IR;
    U1 = U2 + UC
  MACEND (* CONTINUOUS *)
MACEND (* RC2 *);
...
CONTINUOUS
  ...
  IQ = f(TIME);
  RC2 (UQ, UL <- IQ, IL, R, C);
  IL' = UL/L;
```

Fig. 2.4 Model of a RLC-network with current source

In present CSSL-type languages, the submodel concept is usually implemented through the concept of a macro. Coupling of the two submodels is done by use of formal parameters. Internal veriables of the macros are being renamed in the macro replacement process to avoid naming conflicts. However, macros are not truly modular for two reasons:

a) By using a macro, all parameter values must be passed onto higher and higher hierarchical levels until the calling sequence becomes totally unmanageable. The macro facility provides the ability to decompose the program structure but not the data structure. Therefore, macros are not modular with respect to the incorporation of their data structures.

b) From the example shown in Figures 2.1-2.4, it is clear that the same network needs two different model descriptions for the two different environments. Thus, macros are not truly modular with respect to the program structure either.

It can therefore be concluded that macros are not modular (Cellier 1979, 1988).

In discrete event models, the coupling transmits the information among submodels. In a physical system described usually by continuous models, the terminal variables are either "across" variables, measured by placing a meter across the terminals of the device, or "through" variables, measured by placing a meter in series with the device so that the measured quantity is transmitted through the meter. (The above definition is actually slightly simplified. For every network, there exists an adjugate network in which the role of all across and through variables is interchanged.) The coupling of continuous systems must obey physical laws. It is not simply a matter of putting together the terminal ports of the elements when their ports match. It seems to be a good idea to use a connection mechanism in the modeling environment that reflects the way how the physical system coupling mechanism works.

In most CSSL-type languages, a modeler has to explicitly specify the physical laws to describe the coupling equations which relate variables of the individual submodels to each other. This fact prevents models from being truly modular. A clear separation between a model's internal description and its terminal coupling description is paramount to any truly modular hierarchical modeling scheme.

There exist several graphical modeling software systems, such as EASE+ (Expert-EASE systems, Inc., 1988) EASY5 (Boeing Computer Services, 1988), and SYSTEM-BUILD (Integrated Systems, Inc., 1985). In these systems, submodels are maintained in a model library which is similar to a model base. Each of these models is associated with an icon stored in an icon library. Selecting an icon will invoke a submodel. Connections between submodels are done by drawing a line between two terminals of two icons. This is similar to the hierarchical modeling idea. Nevertheless, most of these systems are based on the concept of block diagram modeling. They do not provide for the representation of through variables. They do not provide a hierarchical decomposition of data structures, either. The software SPICE (MicroSim Corp., 1987) and its graphical front-end WORKVIEW (Viewlogic Sytems, Inc., 1988) support the concept of truly hierarchical modeling. These systems use the topological modeling approach. Models of subcircuits are modular at the abstraction level of a topological circuit description (Cellier, 1988). However, SPICE is restricted to simulating electrical networks only.

Zeigler (1984) described five levels of system specification. Among them, level five, i.e., the coupling system specification, supports hierarchical modeling. The current CSSL-type languages usually support the structured system specification.

From the above discussion, it becomes evident that CSSL's are not appropriate for modular modeling, and that another tool is needed to represent continuous models in *proper modular form.* The continuous modeling language DYMOLA, which will be introduced later in this chapter, has been chosen to serve as a generalized macro processor (program generator) for several continuous simulation languages.

2.5 Software Environment

This section presents a brief introduction to the software environment that this research project is based on.

2.5.1 DEVS-SCHEME

DEVS-SCHEME is a software environment for modeling and simulation of discrete event models. DEVS-SCHEME is written in PC-SCHEME, a LISP dialect, that runs on IBM or PC compatible TI microcomputers under MS-DOS. DEVS-SCHEME implements the discrete event system specification formalism developed by Zeigler (1984). It supports the modular hierarchical model specification of discrete event models. The simulation of discrete event models is done by implementing the abstract simulator principles developed as part of the theory (Zeigler, 1984). The ESP-SCHEME software, underlying DEVS-SCHEME, realizes the System Entity Structure (SES) concept. Moreover, DEVS-SCHEME is implemented as a shell in such a way that all the underlying LISP-based and objected-oriented programming language features are available to the user. The result is a powerful tool for combining symbolic and hierarchical modular discrete-event modeling approaches (Zeigler, 1987).

2.5.2 DYMOLA

DYMOLA is a continuous modeling language developed by Elmqvist (1978). It is also a program generator which can be used to generate programs for several different continuous simulation languages, namely SIMNON (Elmqvist, 1975) and now DESIRE (Korn, 1989-b). It can also generate a FORTRAN subroutine for general purpose usage. DYMOLA can therefore be used as a front end to simulation languages. By using a concept called *cut*, DYMOLA realizes the modular and hierarchical modeling of continuous models. DYMOLA has been implemented on VAX/VMS and UNIVAC in the past. Now it can also be used on the PC under MS-DOS.

2.5.3 DESIRE

DESIRE (Direct Executing Simulation In REal time) is a continuous system simulation language. DESIRE has been designed for maximum interactivity and extremely fast execution speed. It is not based on a target language. The time-consuming differential equation part of the program is translated by a small and fast compiler, which produces very efficient code for 8087 or 80287 math co-processors. DESIRE runs on VAX and on PC compatibles. References for DESIRE are Korn (1989-a, 1989-b).

CHAPTER 3
THE ADVANTAGES OF DYMOLA AND ITS ENHANCEMENTS

3.1 Overview

In Chapter 2, the differences between continuous models and discrete event models have been discussed. It has been shown that present CSSL-type languages do not provide means for true modularity. From the perspective of hierarchy of system specifications, model descriptions in CSSL-type languages are actually of the *structured system level* (Zeigler, 1984). Therefore the coupling specification has been absorbed into the resultant system description. In order to apply the hierarchical modeling strategy to continuous systems, a higher level of coupled model specification is required. Appropriate types of "wires" are needed for the continuous models to connect the components. These "wires" enable the continuous models to be raised from the structured system representation level to the *coupled model specification* level. A modeling language, DYMOLA, developed by Hilding Elmqvist (1978), has provided us with this type of powerful "wires".

DYMOLA (Dynamic-modeling Language) is a continuous system modeling language, but it is not a simulation language, as it does not have its own simulation engine. Rather than being a simulation tool, DYMOLA provides the user with a more readable and better modularized hierarchically structured model description (Cellier, 1983, 1988).

DYMOLA acts as a front end to several simulation languages. The input of the DYMOLA translator is the hierarchically structured model; the output will be the flattened model equations. These equations are sorted and grouped into systems of equations. A switch instructs DYMOLA for what target language the output is to be generated. At present, DYMOLA supports DESIRE, SIMNON, and FORTRAN as target languages. It would not be a difficult task to modify DYMOLA to add other simulation languages, such as ACSL, to the list of supported target languages.

DYMOLA uses the submodel concept. Coupling models in DYMOLA are formed by first selecting the submodels, and then connecting the ports of the submodels together.

DYMOLA solves the drawbacks of the CSSL-type languages discussed in Chapter 2 by introducing a new concept, the *cut*. Symbolic formula manipulation sorts the model equations and converts the model equations into assignment statements.

Two different versions of DYMOLA exist. One is coded in PASCAL, the other in SIMULA. The SIMULA version runs on UNIVAC computers; the PASCAL version runs on VAX/VMS and also on PC compatibles, using TURBO PASCAL (Version 4).

The intention of this chapter is to emphasize the modular and hierarchical modeling features of DYMOLA. In addition, some augmentation in DYMOLA will be discussed. For more detailed information about DYMOLA, please, refer to Elmqvist (1978) and Cellier (1983, 1988). The DYMOLA syntax presented in this chapter will be described using the Extended Backus-Naur Form (EBNF) (Bongulielmi and Cellier, 1984) and the syntax diagrams will be illustrated in Appendix 6.

3.2 Modular Hierarchical Model Description in DYMOLA

3.2.1 The "Cut" Concept

When considering a model, its boundaries are determined first. To describe the interaction of a subsystem with its environment, it is necessary to introduce variables describing what happens at the boundaries. In recent CSSL-type languages, submodels are connected through individual variables. This is done by hierarchically calling macros with a set of formal parameters representing the connection variables. Since access is needed to these connection variables, the number of formal parameters keeps growing when advancing to higher and higher levels of the hierarchy. No mechanisms are foreseen to group variables together such as in real systems where wires are grouped into cables, and cables are grouped into trunks or buses.

In DYMOLA, this is accomplished by introducing a connection grouping mechanism, called a "cut". Cuts, in DYMOLA, correspond to complex connection mechanisms of physical systems like shafts, pipes, electrical wires, etc. (Elmqvist, 1978)

Variables declared as *cut* in DYMOLA are actually ports. Depending on the coupling relations between the models, the individual variables forming these ports are interpreted as either input or output. The declaration of *cut* takes the form:

syntax:

$$'\text{cut}' \ \{ \ cut-identifier \ [\ cut \ ] \ \}.$$

where

$$\begin{aligned}
cut-identifier &= identifier.\\
cut &= '(' \ variable-cut \ ')' \mid '[' \ hierarchical-cut \ ']'\\
&\quad \mid cut-spec.\\
variable-cut &= [\ \{ \ across-variable \mid '.' \ \} \ ]\\
&\quad [\ '/' \ \{ \ through-variable \mid '.' \ \} \ ].\\
across-variable &= identifier.\\
through-variable &= identifier.\\
hierarchical-cut &= \{ \ cut \ \}.\\
cut-spec &= identifier.
\end{aligned}$$

semantics:

symbol "." represents a dummy variable.

Examples are:

```
cut  A  (v1  v2  /  v3  v4);
cut  B  (v  /  .);
cut  C  (v1  v2  v3).
```

How does *cut* make a continuous model achieve the purpose of modularity? Consider a very simple case, a model of a resistor. Figure 3.1 and Figure 3.2 illustrate the different model descriptions before and after using the concept of *cut*.

It is seen from the examples that switching the input and output variables will not change the model description with *cut* declarations. However, this is not the major advantage of *cut*, since an I/O variable could be declared as a terminal variable in DYMOLA. The important advantage of *cut* is that, as in the real physical

```
    model name : resistor
    input: I
    output: V
    parameter :R
    equations: V = I*R

or  model name: resistor
    input : V
    output  :I
    parameter: R
    equations:  I= V/R
```

Fig. 3.1 Models of a resistor using input output declaration

```
model name : resistor
cut : A(Va / I) B(Vb / -I)
local :V
parameter :R
equations : V=Va -Vb
            V=I*R
```

Fig. 3.2 Models of a resistor using *cut* declaration

connection mechanism, it associates with two kinds of variables, i.e., across variables and through variables. The equations describing the physical laws at the connection mechanism will be generated automatically from the declaration of *cut* and the connection statements. For example, suppose there are three submodels defined as "R1", "R2" and "R3" (Figure 3.3). A and B are the *cut* variables declared in all three models. VA and I are the across variable and the through variable associating with cut A respectively. By connecting these three models at cut A,

connect $R1 : A$ at $R2 : A$ at $R3 : A$,

the following equations will be generated

$$R1.VA = R2.VA \tag{3.1}$$

$$R2.VA = R3.VA \tag{3.2}$$

$$R1.I + R2.I + R3.I = 0 \tag{3.3}$$

Equations 3.1, 3.2, and 3.3 describe what happens at the boundary of the subsystems where two or more elements are connected.

Several cuts can be grouped together to form a ***hierarchical cut***, which is the same way as wires grouped together into a cable. For instance, in both model "M1" and "M2", the cuts and hierarchical cuts are declared as:

cut A (v1), B (v2), C (v3)

cut G [A B C].

The hierarchical cut G groupes the three ports (A, B, and C). Using the statement in the parent model

connect $M1 : G$ at $M2 : G$,

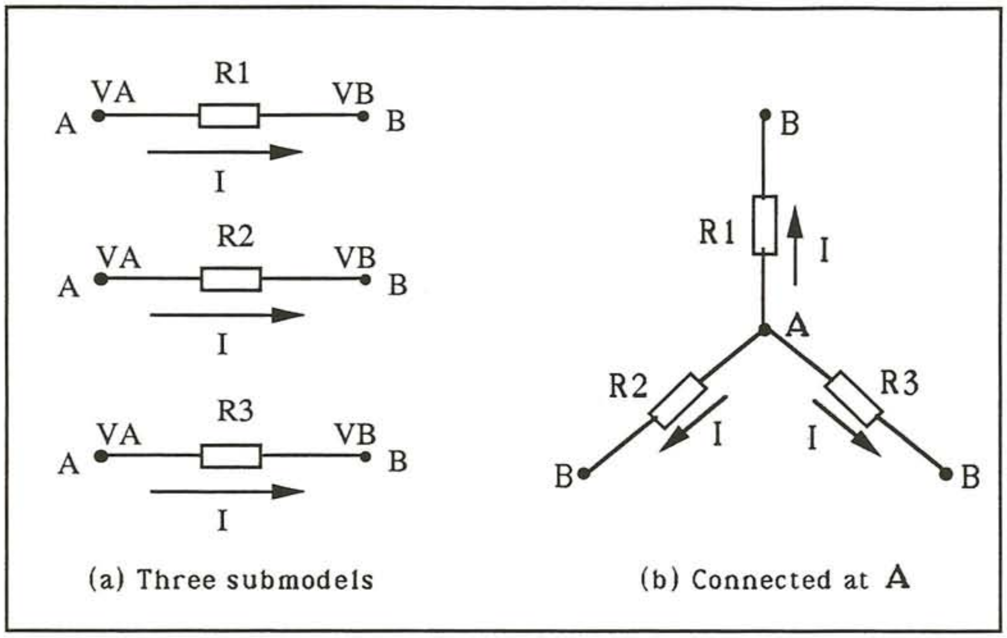


Fig. 3.3 Three submodels connected at port A

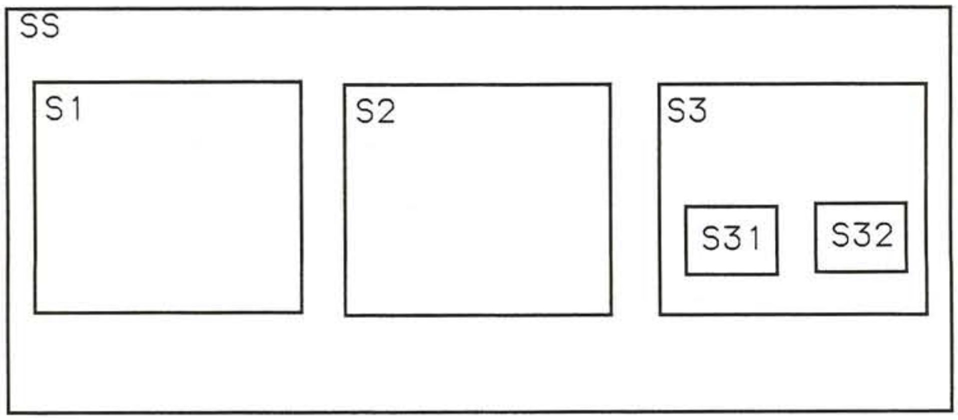


Fig. 3.4 A hierarchically structured system in DYMOLA

results the same as using

$$\text{connect } M1:A \quad \text{at } M2:A,$$
$$\text{connect } M1:B \quad \text{at } M2:B,$$
and $$\text{connect } M1:C \quad \text{at } M2:C.$$

From the above examples, it is clear that by means of *cut*, a model in a continuous system can avoid the model description change caused by the variation of I/O variables. In addition, using *cut* separates the physical laws describing the static or dynamic properties of the model itself from the physical laws dominating at the connecting points of several subsystems. As a result, the interaction between a model and its environment is done through these predeclared ports.

It can be concluded that by introducing the concept *cut*, models in DYMOLA can be described in *proper modular form.* This gives DYMOLA the special power needed for building models in a hierarchical modular manner.

3.2.2 Formula Manipulation

The model description will not change with respect to the environment by the use of *cut* declarations. An equation $U = I * R$ can be coded as $I = U / R$ or $U - I * R = 0$. The modeler has the freedom to choose any of the three equations in a model coded in DYMOLA. This was already demonstrated in Figure 3.2 where the variable V appeared twice on the left hand side of the equal sign. The DYMOLA preprocessor determines for which variable each of the equations must be solved and performs this task automatically. In addition, DYMOLA allows the syntax: $expression = expression$, instead of the commonly used syntax: $variable = expression$. The question of what should be the input variables and what should be the assigned variables is left to DYMOLA instead of the user. The

```
Model SS
   model S1
   ...
   end
   model S2
   ...
   end
   model S3
      model S31
      ...
      end
      model S32
      ...
      end
   ...
   end
...
end
```

Fig. 3.5 Description of the hierarchical structure of a system

```
Example1 : A RESISTOR

model resistor
  cut A (VA / I) B (VB / -I)
  local V
  parameter R=1
  V = VA-VB
  R*I = V
end


Example 2 : A CAPACITOR

model capacitor
  cut A (VA / I) B (VB / -I)
  local V
  parameter C=1
  V = VA-VB
  C*der(V) = I
end
```

Fig. 3.6 Examples of atomic models in DYMOLA

formula manipulation routines in DYMOLA produce the required form of the equations automatically, depending on the context in which the equation is used. This gives the language an important characteristic: it makes the models independent from the operations that are performed on them.

3.2.3 Hierarchical Model Structure in DYMOLA

A system named "SS" is considered here as an example. It is composed of three subsystems, "S1", "S2" and "S3". The subsystem "S3" is further decomposed into "S31" and "S32". The configuration of "SS" is shown in Figure 3.4.

Figure 3.5 illustrates one way to describe the hierarchical structure of system SS. Although Figure 3.5 presents a hierarchical structure, it has a serious drawback. For instance, if two of the subsystems have the same model, the model description will be duplicated. Section 3.2.5 will introduce an alternative approach to representing the hierarchical model.

3.2.4 Submodels in DYMOLA

In DYMOLA, a submodel can be an atomic model, i.e., a model without coupling, as well as a coupled model. The specification of atomic models has the following syntax:

syntax:

$$
\begin{aligned}
&\text{'model'} \quad [\text{'type'}] \quad model - identifier\\
&declaration - of - variables\\
&[declaration - of - connection - mechanism(*ports*)]\\
&model - descriptions\\
&\text{'end'}.
\end{aligned}
$$

where

$$model - identifier \ = \ identifier.$$

Examples of atomic models are illustrated in Figure 3.6.

On the other hand, the specification of coupled models in DYMOLA has the syntax described below:

syntax:

$'\text{model}' \quad ['\text{type}'] \quad model - identifier$

$declaration - of - submodels$

$declaration - of - variables$

$[declaration - of - connection - mechanism(*ports*)]$

$connection - descriptions - and - equations$

$'\text{end}'.$

where

$$model - identifier \ = \ identifier.$$

```
model rcp
  submodel resistor(20)
  submodel capacitor(10)
  cut A (VA / I)
  cut B (VB / -I)
  connect  resistor:B at capacitor:B
  connect  resistor:A at capacitor:A
  connect A at resistor:A
  connect resistor:B at B
end
```

Fig. 3.7 Example of a coupled model *rcp*

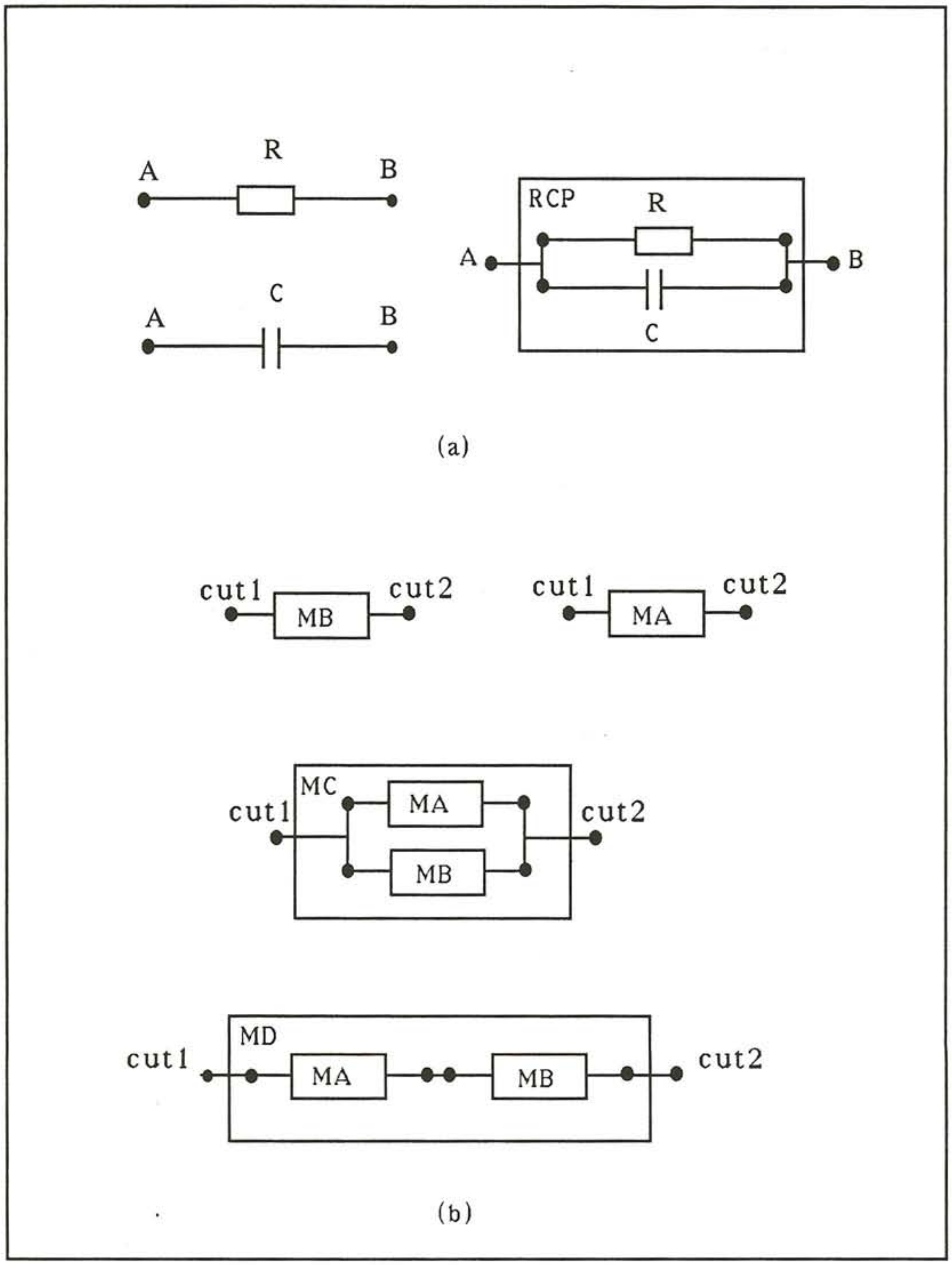


Fig. 3.8 Coupled models in DYMOLA

An example of a coupled model is shown in Figure 3.7. In this example, the submodels are one resistor and one capacitor, which are in modular form. The resistor's only parameter R (Figure 3.6) is assigned to be 20 (indicated by the (20) in the submodel statement); the capacitor's only parameter C (Figure 3.6) is assigned to be 10.

A coupled model "rcp"(parallel connected resistor and capacitor) is generated by coupling the two atomic models together. The result, a coupled model "rcp", is once again in proper modular form, and can be employed to construct yet larger models, in the same manner as with the "resistor" and the "capacitor". The concept of coupled models in DYMOLA is depicted in Figure 3.8.

3.2.5 Model Type and Model Library

In Section 3.2.3, the shortcomings of the hierarchical description (Figure 3.5) have been discussed. A method to describe coupled models was shown in Section 3.2.4. By coupling the models recursively, a large system can be represented in a hierarchical manner. In order to avoid duplicated descriptions of similar submodels, DYMOLA employs a term "model type". A model specified as "model type" represents a generic model of a general class of objects. In some cases, several models have different parameters, but these models are of the same type. In some other cases, a superior model needs several submodels which are the same. The user can declare a "model type" in DYMOLA. This "model type" can be used to generate several models with a submodel statement so that duplication will be avoided.

For example, model types can be defined for the "resistor" and the "capacitor" presented in Section 3.2.4. In the model specification, the "model resistor" and "model capacitor" are changed to "model type resistor" and "model type capacitor", respectively. The simplified model specification for "rcp" using model types is shown in Figure 3.9.

```
model type rcp
  submodel (resistor) rtwo (20)
  submodel (capacitor) cone (10)
  cut A (VA / I)
  cut B (VB / -I)

  connect  rtwo:A at cone:A
  connect  rtwo:B at cone:B
  connect A at rtwo:A
  connect rtwo:B at B
end
```

Fig. 3.9 Model specification for *rcp* using model type

At this point, it is natural to establish libraries of models, each one containing all the model types of a system. When a system is modeled, the library is set up first, then the hierarchy of the coupling relations can be specified.

3.3 DYMOLA ON PC

One version of DYMOLA has been coded in Pascal to run under VAX/VMS. In the first phase of this project, DYMOLA has been ported onto PC compatible computers. The compiler used is Turbo Pascal Version 4.0 or higher. In addition to the porting, some enhancements and modifications were made to meet some new challenges which have arosen in the current research. These changes will be discussed in this section.

3.3.1 Enhancement of DYMOLA to Generate DESIRE Models

The DYMOLA commands, "output simnon model" and "output fortran model", will generate Simnon and Fortran models, respectively. Now, DYMOLA has been enhanced to be able to generate DESIRE models by the command

output desire model.

Note that before outputting the models, the user has to issue the command "partition", that manipulates all the equations generated by the model description and the connection mechanism, in the following fashion. The partitioning starts with structure information, i.e., whether a variable is present in an equation or not. It then determines for which variable each equation must be solved, then partitions the equations into minimal systems of equations that must be solved simultaneously. At last, *partition* sorts the equations into the correct computational order. An example in Figure 3.11 demonstrates how this works. Suppose there are two files "trivial.lib" and "trivial.sys", with "trivial.lib" containing the type models (here, a capacitor, shown in Figure 3.6 before) needed for the system trivial, and "trivial.sys" being the system description. The file "trivial.sys" is shown in Figure 3.10

```
model trivial
  submodel (capacitor) cc(10)
  input u
  output y
  cc.VB = 0
  u = cc.I
  y = cc.V
end
```

Fig. 3.10 File "trivial.sys"

```
DYMOLA
> enter model
- @trivial.lib
- @trivial.sys
> output equations
  cc                  V = VA - VB
                      C*derV = I
  trivial             cc.VB = 0
                      u = cc.I
                      y = cc.V
> partition
> output sorted equations
  trivial             [cc.VB] = 0
  cc                  V = [VA] - VB
  trivial             u = [cc.I]
  cc                  C*[derV] = I
  trivial             [y] = cc.V
> output solved equations
  trivial             cc.VB = 0
  cc                  VA = V + VB
  trivial             cc.I = u
  cc                  derV = I/C
  trivial             y = cc.V
> output desire model
  ----------------------------
  -- CONTINUOUS SYSTEM trivial
  ----------------------------
  -- STATE V
  -- DER dV
  -- OUTPUT y
  -- INPUT u
  -- PARAMETERS and CONSTANTS:
  C = 10
  -- INITIAL VALUES OF STATES:
  V = 0
  ----------------------------
  DYNAMIC
  ----------------------------
  -- Submodel: trivial
  VB = 0
  -- Submodel: cc
  VA = V + VB
  -- Submodel: trivial
  I = u
  -- Submodel: cc
  d/dt V = I/C
  -- Submodel: trivial
  y = V

 note: 1. ">" is the system prompt for users to type in a command.
       2. "-" is the system prompt for users to type in the file name of
          the models. Users have to type "@" for the input of model files.
       3. The system outputs of this example are shown on the screen.
          they can also be saved into files.
```

Fig. 3.11 Example of system *trivial*

With these two files existing, the user then runs DYMOLA by typing in "dymola". The result is shown in Figure 3.11.

3.3.2 Creation of a DESIRE Simulation Program

Running the simulation of a continuous system requires at least the basic information for simulation control, such as simulation step, communication points, simulation time, etc. It can be seen that the "output desire (fortran, simnon) model" only generates the program of the model itself, since the DYMOLA preprocessor compiles only the model body. Thus, an executable simulation program cannot be produced without the user's additional work on the output model from DYMOLA. In our Knowledge Based Modeling and Simulation Environment, it was expected that the output of DYMOLA should be directly executable. For this purpose, a special type of model called simulation control model has been introduced in DYMOLA.

3.3.2.1 Description of the Simulation Control Model

The simulation control model must be stored into a file with the same filename as that of the system being controlled. The file extension "ct" indicates a simulation control model. This control model consists of three parts:

1. basic part;
2. run control block;
3. output block.

The structure of the control model is as follows:

syntax:

'cmodel'

basic − part

$['\text{ctblock}'$

$control - descriptions$

$'\text{ctend}']$

$['\text{outblock}'$

$output - descriptions$

$'\text{outend}']$

$'\text{end}'.$

As specified, *ctblock* and *outblock* are optional. The information contained in the basic part is:

1. simulation time;
2. simulation step size;
3. number of communication points;
4. inputs (optional) .

The format for the basic part is depicted as follows:

syntax:

$'\text{simutime}' \quad simulation - time$

$'\text{step}' \quad step - size$

$'\text{commupoints}'\ number - of - communication - points$

$[input - specification].$

where

$simulation - time \ = \ number.$

$step - size \ = \ number.$

$number - of - communication - point \ = \ integer.$

$integer \ = \ \{digit\}.$

$number \ = \ ['+' \mid '-']\ unsigned - number.$

$unsigned - number \ = \ integer\ ['.'\ [integer]]\ ['E'\ ['+' \mid '-']\ integer].$

Note that the definition for number and integer will remain the same throughout this thesis. If the input of a system is not specified in the model, it should be provided in the simulation control model. The format is:

syntax:

$$'\text{input}'\ \ number - of - inputs\ ','\ \ \{\ variable\ (\ '\text{depend}'\ |$$
$$'\text{independ}'\ ','\ expression\ )\ \$\ [','\,]\ \}.$$

where

$$number - of - inputs\ =\ integer.$$
$$variable\ =\ identifier.$$

and *depend/independ* specifies whether the input variable is time dependent or not; *expression* specifies the input expression.

For example:

input 2, u1(depend, sin(3*t)), u2(independ, 10).

means that there are two inputs, u1 and u2. u1 is a function of time (equals sin(3*t)); while u2 is a constant (equals 10). The DYMOLA compiler puts the time dependent input into the DYNAMIC segment part of a DESIRE program and a constant input into the job control block of a DESIRE program. The inputs provided in the simulation control model should be consistent with the inputs declared in the system model.

The run control block encloses the run control statements which can appear in the run-time output part of a DESIRE program (section 2.3.3; Korn 1989-a).

Simulation output requirements should be placed in the output block. The supported output statements are "dispt", "dispxy", "type", and "stash", with the syntax described below:

(a) Run-time CRT graphs can be obtained with

syntax:

'dispt' {*var* $ ','}.

where

$var = identifier.$

semantics:

number of *var* is up to 4, versus *t*.

or

syntax:

'dispxy' *var* ',' {*var* $ ','}.

where

$var = identifier.$

semantics:

number of *var* is up to 3, versus the first one.

These definitions will remain the same in this section.

(b) The statement

syntax:

'type' {*var* $ ','}.

where

$var = identifier.$

semantics:

number of *var* is up to 4.

produces runtime tabulated output with respect to the independent variable t on the screen.

Note that only one "dispt", "dispxy" or "type" statement is allowed in each program.

(c) The statement

syntax:

$$'\text{stash}' \quad \{var \ \$ \ ','\}.$$

where

$$var \ = \ identifier.$$

semantics:

number of *var* is up to 20.

stores successive values of t, vi . . . at each communication point into a file with the name of the current running file and the file extension "tim". (Refer to Korn, 1989-a.)

All the statements in the output block are interpreted only at communication points. This reduces the computation time significantly.

3.3.2.2 Obtaining Executable DESIRE Programs

The command:

output desire program

is used to create executable DESIRE programs. The program will first verify the existence of the simulation control model associated with the system. If this model exists, an executable DESIRE program will be generated; otherwise an error message will be displayed. Figure 3.12 continues the example of system "trivial". It is demonstrated how an executable DESIRE program for this system is generated.

1. The simulation control model of trivial (file "trivial.ct") contains:

```
cmodel
 simutime     20
 step         0.1
 commupoints  50
 input  1,u(depend,sin(0.5*t))
 outblock
 dispt u,y
 outend
end
```

2. In DYMOLA (after partition):

```
> output desire program

-----------------------------------------
-- CONTINUOUS SYSTEM trivial
----------------------------------------
-- STATE V
-- DER dV
-- OUTPUT y
-- INPUT u
-- PARAMETERS and CONSTANTS:
C=10
-- INITIAL VALUES OF STATES:
V=0
----------------------------------------
TMAX=20 | DT=0.1 | NN=50
drun
----------------------------------------
DYNAMIC
----------------------------------------
u = sin(0.5*t)
-- Submodel: trivial
VB = 0
-- Submodel: cc
VA = V + VB
-- Submodel: trivial
I = u
-- Submodel: cc
d/dt V = I/C
-- Submodel: trivial
y = V
----------------------------------------
dispt u,y
----------------------------------------
/--
/PIC 'tri.PRC
/--
```

Fig. 3.12 Example of system *trivial* (continued)

3.3.3 Other New Features

3.3.3.1 Initial Conditions

The initial conditions of all integrators in DYMOLA are zero by default. Originally, a state variable could only assume an initial value different from zero if the state variable had been declared as a local variable, e.g., "local V=2.0". This however meant that the values of the initial conditions could not be changed in submodel invocations.

The syntax of the submodel has therefore been changed to the form shown below which overcomes this disadvantage of DYMOLA:

syntax:

$$\text{'submodel'}\ [\ '('\ model-type-identifier\ ')'\ ]\ \{model-identifier$$
$$[\ '('\ parameter-list\ ')'\ ]\ [\ '(\ ic'\ initial-condition-list'\ )'\ ]\}.$$

where

$$model-type-identifier\ =\ identifier.$$
$$model-identifier\ =\ identifier.$$
$$parameter-list\ =\ \{number\ \$\ [','\,]\}\ |\ \{(parameter\ '='\ number)\ \$\ ['.']\}.$$
$$parameter\ =\ identifier.$$
$$initial-condition-list\ =\ \{(variable\ '='\ number)\ \$\ ['.']\}.$$
$$variable\ =\ identifier.$$

Examples are:

submodel (capacitor) c1 (5.6) (ic V=0.5), c2 (1), c3(10) (ic V=10);

submodel inductor (ic I=9).

The parameter list or initial condition list following the model identifier sets or changes the default values of parameters, or the initial conditions of this model, respectively.

3.3.3.2 Special Functions

The syntax for the statement description in DYMOLA is

expression = *expression.*

To increase the compatibility of DYMOLA with modern simulation languages, a second form of statement was introduced which will be discussed below. This modification contributes to DYMOLA the flexibility of managing various kinds of functions that appear in the target languages.

1. The Tabular Function

syntax:

'func' *variable* '=' *expression.*

When "func" is specified, the function expressed in the statement denotes a function generator operation.

Example:

func y = TAB(x)

in which x is the independent variable; while y is dependent on x. The values of TAB are given in the simulation control model by the declaration of an array with the dimension n, where n must be even. The array contains n/2 breakpoint abscissas from TAB[1] to TAB[n/2], followed by n/2 breakpoint ordinates from TAB[(n/2)+1] to TAB[n]. The values of y are produced by table look up and linear interpolation between function breakpoints, and

$$y = TAB[(n/2)+1] \quad \text{when } x \leq TAB[1];$$
$$y = TAB[n] \quad \text{when } x \geq TAB[n/2].$$

2. The Storage Functions

syntax:

'store' *variable* '=' *expression.*

and

'get' *variable* '=' *expression.*

can store and get the time history of certain variables into and from prespecified one dimensional arrays, which must be declared in the simulation control model. Examples:

store tab1 = x

stores the values of x, at each communication point, into the array tab1, which can then be reused for further processing, printing, etc. The dimension of tab1 should be the same as the number of communication points.

get y = tab2

makes the value of y at and between successive communication points equal to the successive elements of the one-dimensional array tab2, which is predefined in the simulation control model, such that

$$y = \text{tab}[k]$$

when (k-1)*communication-interval $\leq$ t-t0 $<$ k*communication-interval,

where

t = simulation time;

t0 = the initial simulation time;

k = 1, 2, 3, . . .

If the time exceeds the dimension of the array multiplied by the communication interval, y keeps the last value of array tab2.

get is a fast time-function generator (without linear interpolation); it also permits computations involving stored time histories (Korn, 1989-a).

3.4 Unsolved Problems

As previously stated, DYMOLA uses a different approach from other continuous modeling and simulation languages to the modeling of continuous systems, which enables the user to invoke reusable models without studying the details of the submodels when building a large system. It also has been furnished with more properties that fulfil the purpose of the ongoing project, which shall be presented in the coming chapters. A fair amount of program development and study are still needed to make DYMOLA a productional code. Listed below are some suggestions for future research (Cellier, 1988):

1. DYMOLA has the ability to eliminate equations of type $a = b$ by replacing all occurrences of a by b. There exists a problem that if variable a is to be displayed, which is specified in the output block of the simulation control model, it is expected that b will be eliminated instead of a. DYMOLA has not taken this situation into consideration yet. Besides, "DYMOLA should

also be able to eliminate variables from equations of type $a \pm b = 0$." (Cellier, 1988).

2. "DYMOLA should be able to recognize equations that have been specified twice, and eliminate the duplication automatically to avoid redundant equations." (Cellier, 1988). This property is especially important for hierarchically connected submodels. Meanwhile, special care must be taken when relating through variables. For instance, one of the node equations for current is redundant when connecting electrical components. In one of the submodels, a dummy through variable must be used, such as cut A(Va / I) B(Vb /.), to solve this problem. This necessity somewhat jeopardizes the generality of the model descriptions.
3. "DYMOLA should be able to handle superfluous connections," i.e., suppose $w1$, $w2$ are two angular velocities and $b1$, $b2$ are their corresponding angles; "if we specify that $b2 = -b1$, it is obviously true that also $w1 = -w2$. However, DYMOLA will not let us specify this additional connection at the current time. Superfluous connections should simply be eliminated during model expansion." (Cellier, 1988)
4. "DYMOLA should recognize that connections of outputs of integrators can always be converted into connections of inputs of these integrators, i.e., if we have specified that $ia3 = ia2$, it is obviously true that $iadot3 = iadot2$. This reformulation can help to eliminate structural singularities." (Cellier, 1988)
5. Groups of linear algebraic equations are currently grouped together and printed out by DYMOLA without being solved. DYMOLA should be able to rewrite the system of equations into a matrix form, since DESIRE can handle matrix expressions efficiently and future versions of DESIRE will include efficient

algorithms for inverting matrices. For simple algebraic loops, it might be a good idea to shift to the interactive model and ask the user for help (Cellier, 1988).

6. In nonlinear equations like

$$X^2 + Y^2 + 2 * Z - 5 = 0,$$

problems will arise if DYMOLA wants to solve for X or Y. Cellier provides several possible solutions for handling nonlinear equations. (Cellier, 1988)

7. Since DESIRE can solve discrete time systems, it would be good if DYMOLA could handle discrete time models as well.

8. DYMOLA should have the ability to manage statements like:

$$identifier \quad expression,$$

where the identifier would be one of the keywords in DESIRE or other simulation languages. These kinds of statement are now treated in the output block of the simulation control model in order not to affect partitioning the model equations. It would be more flexible if these statements could also be compiled together with the model body.

9. More powerful user interfaces, such as manual driven and graphic modeling, are needed. Most of the existing graphic modeling software tools, such as EASY5 (Boeing Computer Services, 1988), SYSTEM-BUILD (Integrated Systems, Inc., 1985), etc., are based on the concept of block diagrams. They do not provide the data structures needed for hierarchical decomposition (like *cuts*), and they do not provide the representation of through variables. A graphic preprocessor of DYMOLA, HIBLITZ, exists which supports all the modeling concepts of DYMOLA (Elmqvist, 1982). HIBLITZ currently runs on Silicon

Graphics (IRIS) machines only. It is hoped that HIBLITZ can be converted for use on 386 compatibles (Cellier, 1988).

CHAPTER 4
CONTINUOUS SYSTEM MODELS IN DEVS-SCHEME

4.1 Introduction

Models are multifacetted in nature and modeling is goal driven. The goal of the modeler directs the process of model construction (Zeigler, 1984; Rozenblit, 1985). Constructing models to meet new objectives can be accelerated by the so-called *multifacetted modeling methodology*, which is based on the combination of the System Entity Structure and Model Bases (Zeigler, 1989-b). In this approach, the lower layers of models, which are less important to the objective, are aggregated into atomic (or even coupled) units residing in a model base (Cellier, 1988). The *System Entity Structure*, which was previously introduced as a knowledge representation scheme in Chapter 2, and which will be discussed in more detail in Chapter 5, can be used to organize models and facilitate model synthesis. DEVS-SCHEME, a Knowledge Based Modeling and Simulation Environment, is a realization of the theory of multifacetted modeling methodology.

Several approaches to the modeling of continuous systems using DEVS formalism have been proposed (Zeigler, 1989-a). The discrete event description of a continuous model relates the continuous system counterpart by abstraction. The abstraction plays an important role in advanced robot and intelligence control where a continuous process must be interfaced with a symbolic reasoning system (Zeigler, 1989-a).

Furthermore, building a robot model in the "Simulation Environment for Laboratory Management by Robot Organization" involves the design of Model-Plan

Units (MPU), which in turn involves modeling of continuous processes by discrete event models. The MPU plans sequences of operations. For each action on the real processes, a normal state trajectory would be obtained by the continuous models. The normal process window, which indicates the range of the simulation time under normal conditions, would also be determined from running series of continuous process simulations with different parameters and initial conditions (Zeigler, Cellier and Rozenblit, 1988). DEVS-SCHEME should be, but was not, able to execute continuous simulation runs and to generate time histories, which can be referred to by a model in DEVS-SCHEME to determine appropriate thresholds for switching from one state to another state in an equivalent discrete event model. In order to manage continuous models in DEVS-SCHEME, two new classes, named *continuous-models* and *continuous-systems*, have been created in DEVS-SCHEME.

4.2 Classes in DEVS-SCHEME

As stated in Chapter 2, DEVS-SCHEME is built upon the object-oriented programming paradigm. Every object has its own variables and methods. A class definition in an object-oriented programming environment provides a template for generating any number of instances, each one being an identical copy of a basic prototype (Booch, 1986; Zeigler, 1989-b).

In DEVS-SCHEME, the root class is called *entities*. This class provides tools for manipulating objects within this class and all its subclasses. All other classes in DEVS-SCHEME are subclasses of this universal class. Scheme's "Scoop" facility offers inheritance mechanisms that are sufficiently strong to ensure that such general facilities be defined only once.

Models and *processors*, the two main subclasses of entities, offer the basic constructs for modeling and simulation. The major subclasses of models are *atomic-models* and *coupled-models*, which are further specialized into more specific classes. On the other hand, the class called processors has three specializations, namely *simulators*, *coordinators*, and *root-co-ordinators* (Appendix 1), which handle all the simulation processes.

$M = \langle X, S, Y, \delta_{ext}, \delta_{int}, \lambda, \tau \rangle$

X _ set of external events received as input

Y _ set of outputs, external events generated output

S _ set of states

δ_{ext} _ external transition function

δ_{int} _ internal transition function

λ _ output function

τ _ time of next event

Fig. 4.1 Formalism of atomic models of discrete event system

The atomic-models class realizes the atomic level of the DEVS model formalism (Figure 4.1) by declaring variables corresponding to each of the parts in the formalism. The coupled-models class embodies the hierarchical model composition constructs of the DEVS formalism. *Digraph-models* and *kernel-models* (Appendix

1), the two main subclasses of coupled-models, enable descriptions of coupled models in particular ways. In the DEVS formalism, a coupled model is defined by its component models and through coupling relations which establish desired communication links.

The continuous-models class, representing the atomic continuous system models in DEVS-SCHEME, is a subclass of the atomic-models class, and the continuous-systems class is a subclass of the digraph-models class.

Figure 4.2 illustrates the overall taxonomic hierarchy of classes in DEVS-SCHEME. Brief explanations of some of the classes are given in Appendix 1. Detailed descriptions, usages, and studies about various classes in DEVS-SCHEME can be found in Zeigler (1986, 1987, 1989-b), Kim (1988), Zhang (1988), and Christensen (1989).

4.3 The Class of Continuous-models

The continuous-models class is a specialization of atomic-models. An instance of continuous-models in MBASE (the DEVS model base) serves mainly as a pointer, pointing to its corresponding continuous model in DYMOBASE (the DYMOLA model base).

The continuous behavior knowledge of a model is captured by a DYMOLA model. On the other hand, a DEVS continuous model is the abstraction of the dynamic model from the continuous behavior into a different kind of event-based behavior according to the requirement.

Note that throughout this thesis, the DEVS models, with filenames such as "modelname.m", are stored in the discrete event model base MBASE, while DYMOLA models, with filenames the same as the model file name but without the file extension, are stored in the dynamic model base DYMOBASE.

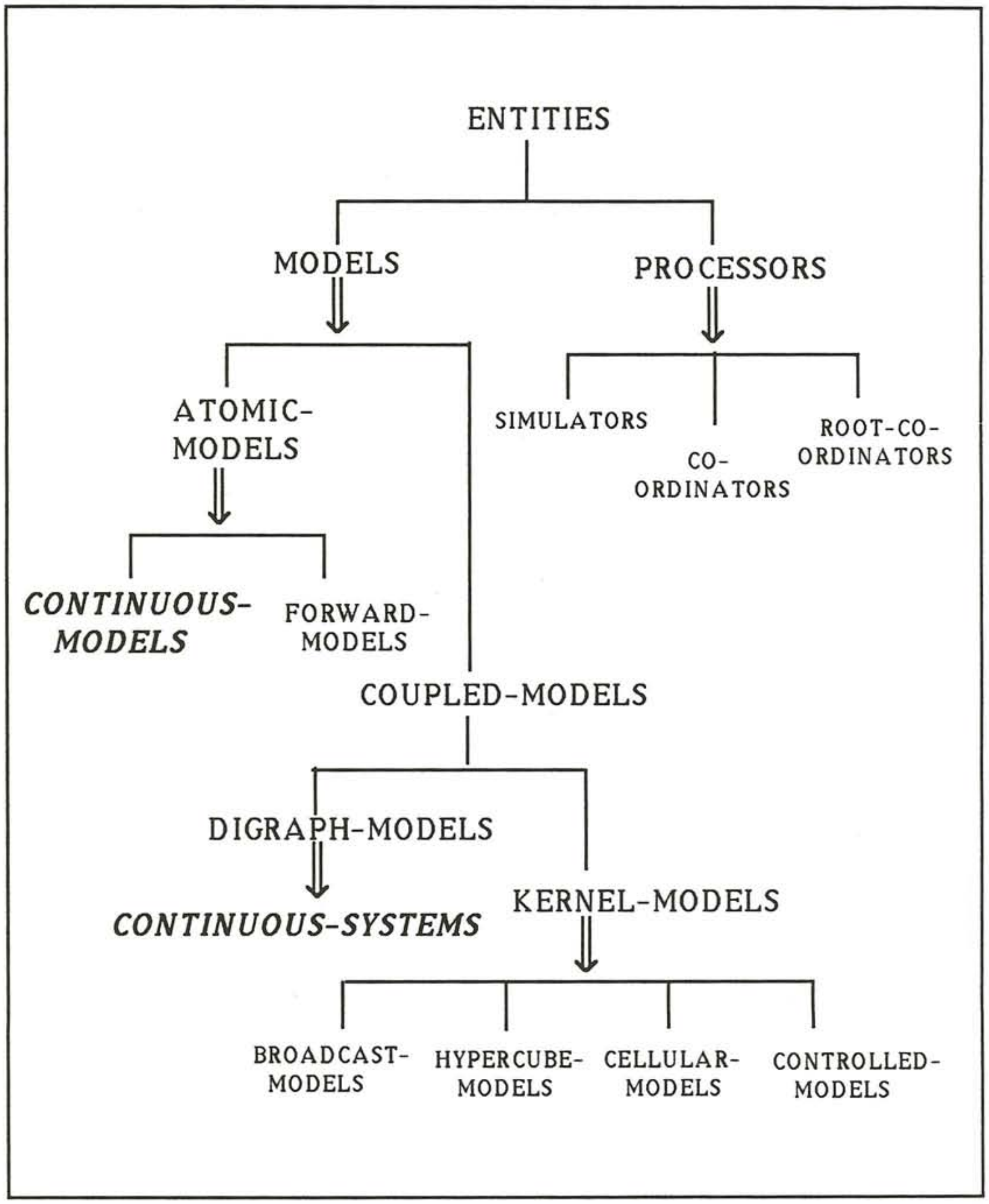


Fig. 4.2. Class hierarchy of DEVS-SCHEME

4.3.1 Descriptions of Continuous-models

It was explained in Chapter 3 that the term "type model" refers to a generic model of a general class of components or devices, such as resistor. Then a model of class continuous-models might be an instantiation of one of the type models (a particular resistor), or it might be a model which has no type model. The *instance variables* show which of the categories a model belongs to. Instance variables also hold the knowledge about the declaration part of a model in DYMOLA.

The instance variables associated with continuous-models are:

1. *name*: the name of the continuous-models component, inherited from entities;
2. *tflag*: indicates whether this model is a type model. Its value can only be true or false;
3. *tname*: If the model is a type model, its *tname* is the same as *name*, which is the generic type. If the model is a specific instantiation of a type model, then its *tname* is the *name* of that type model. If the model is neither a type model nor a specific instantition of one type, then its *tname* is "nil";
4. *ind-vars*: a structure. The name of the structure is *state*, and the default fields of this structure are *sigma* and *phase*, both inherited from atomic-models, and *cut, mcut, path, mpath, node, parameter, local, terminal, input,* and *output*, which represent the types of variables declared in DYMOLA.

To clarify the meanings for *name*, *tflag*, *tname*, suppose that a system includes two resistors, say "r1" and "r2", which are all instantiations of the "resistor" model type, and a model "m1". Then the instance variables are assigned as follows:

model	*name*	*tflag*	*tname*
resistor	resistor	true	resistor
r1	r1	false	resistor
r2	r2	false	resistor
m1	m1	false	nil

Other instance variables, such as *int-transfn, ext-transfn, outputfn* and *time-advancefn*, are inherited from atomic-models (Zeigler, 1986, 1987, 1989-b; Kim, 1988; Christensen, 1989).

```
;-- Class continuous-models
(define-class continuous-models
   (classvars)
   (instvars
     tflag
     tname
     (ind-vars '(sigma phase
                 cut mcut path mpath node
                 parameter
                 local terminal input output)
     )
   )
(mixins atomic-models)
  (options
    gettable-variables
    settable-variables
    inittable-variables)

)

;--define states for continuous-models
(define-structure state sigma phase
                        cut mcut path mpath node
                        parameter
                        local terminal input output)
```

Fig. 4.3 Class definition of continuous-models

The definition of the continuous-models class (Figure 4.3) indicates that this class has the newly created/defined instance variables *tflag, tname* and *ind-var.* It inherits all the attributes from its parent class atomic-models (defined by mixins in Figure 4.3), and all the instance variables of this class can be assigned with initial values, readable and modifiable (indicated by options). "(define-structure ...)" defines the structure named *state* for the instance variable *ind-vars.*

4.3.2 Methods for Continuous-models

The methods for continuous-models are:

1. set-sv (vname vvalue);
2. get-sv (vname);
3. set-type (tf tn);
4. valid? (tf tn);
5. make-new (mname);
6. change-parameter (p-list);
7. change-ic (ic-list);

A detailed explanation of these methods is given in Appendix 2.

Sending a method to an instance of a class in DEVS-SCHEME is done by

(send *name-of-method name-of-object parameter1 parameter2 ...*)

Suppose there is a DYMOLA model named "resistor" in DYMOBASE (Figure 3.6), and a corresponding DEVS model "resistor.m" in MBASE as defined in Figure 4.4.

```
(make-pair continuous-models 'resistor)
(send resistor valid? #t '())
(send resistor set-s (make-state 'cut '((A (VA / I)) (B (VB / -I)))
                                 'parameter '((R 1))
                                 'local '((V 0))
                     )
)
```

Fig. 4.4 An example of continuous-models

In this example, "(make-pair ...)" creates an entity called "resistor" of class continuous-models in DEVS-SCHEME, and attaches a simulator for this entity. Then the method *valid?* is sent to resistor to check whether it is a valid model of this class. Method *valid?* has two parameters, *tflag* specifying wether the model is a type model in DYMOLA, and *tname* specifying the type this model belongs to if it is not a type model itself. If a model is a type model then *tname* can be set to nil. Model "resistor " is a type model so its *tflag* equals true, and *tname* is nil. If the type model "resistor" is not residing in DYMOBASE, then this DEVS model is not valid and an error message will be displayed. The method *set-s* is inherited from the class atomic-models. "(set-s ...)" generates a structure named *state* by (make-state ...), and fills the values of the slots for the instance variable *ind-vars*.

It can be seen that the description of the "resistor" in MBASE written in DEVS contains the knowledge about its counterpart model in DYMOBASE. Furthermore, the instance variables *int-transfn, ext-transfn, outputfn,* and *time-advance* can still be assigned to this model upon necessity. The conclusion can be drawn that a DYMOLA model is considered a "base model", by Zeigler's definition of the five elements: the real system, the experimental frame, the base model, the lumped model, and the computer, in the theory of modeling and simulation (Zeigler, 1985) providing a relatively "complete explanation of the behavior of a real system"; while a DEVS continuous model plays the role of a "lumped model" constructed from the base model.

4.4 The Class of Continuous-systems

Continuous-systems is a subclass of digraph-models. Instances in class continuous-systems are generated by applying the structural knowledge of the particular systems. Inherited from digraph-models, it provides means to specify explicitly couplings between the components. A coupled model of the continuous-systems class will be generated by specifying:

1. the internal coupling, structural relations between components, i.e., components and their influencees;
2. the external coupling, structural relations between the coupled model and the outside world including:
   a. the external input coupling, i.e., inputsfrom the outside world to the coupled model;
   b. the external output coupling, i.e., outputs from the coupled model to the outside world.

At the same time, a coupled DYMOLA model will also be automatically generated and stored in DYMOBASE.

4.4.1 Descriptions of Continuous-systems

The instance variables of the continuous-systems class are:

1. *name:* the name of the continuous-models component, inherited from entities;
2. *tflag:* indicates whether this model is a type model. Its value can only be true or false;
3. *tname:* If the model is a type model, then its *tname* equals its *name*. If the model is not a type model but is an instantiation of a type model, then its *tname* equals its type model's name. If the model is not a type model, nor does it belong to any type then its *tname* equals "nil";
4. *tylist:* a list of names of the type models needed for the coupled model;
5. *vars:* a structure. The name of the structure is *cstate* and the default fields of this structure are *cut, mcut, path, mpath, node, parameter, local, terminal, input,* and *output.* These fields represent the types of variables declared in a DYMOLA coupled model.

Listed below are the instance variables inherited from digraph-models (and coupled-models). However, some of them are reassigned, and therefore, have a different meaning from those in the parent classes digraph-models and coupled-models:

1. *children*: specifies all the subcomponents of the coupled model (inherited from class coupled-models).
2. *receivers*: specifies the children which will receive an external input event from the parent coupled model (inherited from class coupled-models).
3. *influencees*: determines which siblings one component will be connected to, without considering the I/O direction. In contrast, in the coupled-models class it determines which siblings the output of one component will be sent to.
4. *composition-tree*: represents the structure of the hierarchical model. The root of the composition-tree is a coupled model. The children who are atomic

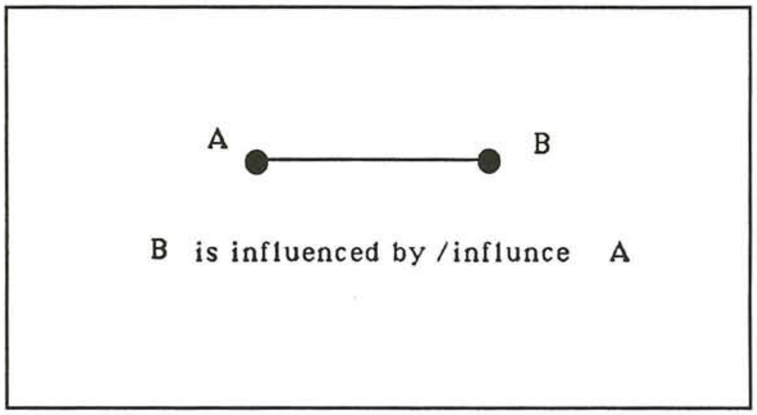


Fig. 4.5 Influence digraph of a model of class continuous-systems

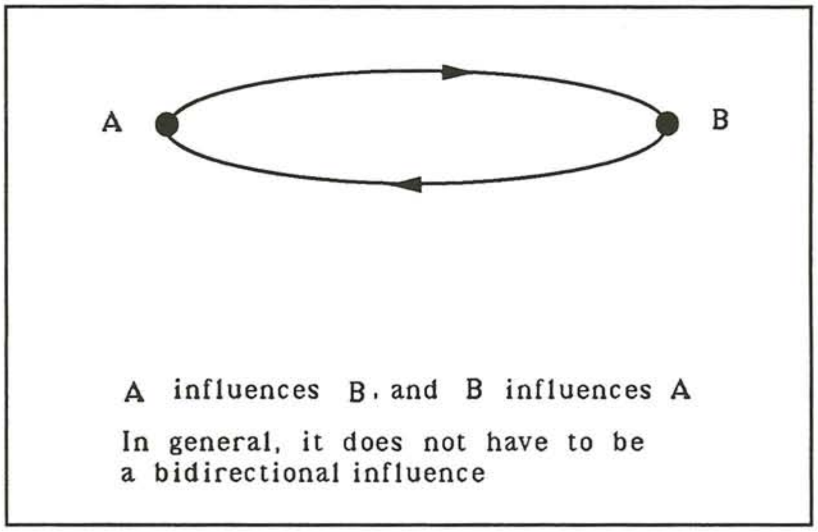

A
B
A influences B, and B influences A
In general, it does not have to be
a bidirectional influence

Fig. 4.6 Influence digraph of a model of class digraph-models

models become leaves in the tree. The children who are not atomic models are further decomposed into their children. All the leaves in the composition-tree are atomic models (inherited from digraph-models).

5. *influence-digraph*: represents the non-directed information flow graph between one component and its siblings. In contrast, in the parent class digraph-models, it represents the directed I/O graph between one component and its siblings. Therefore, the influence-digraph of a continuous-systems model reflects only the physical structural relations. It does not include the computational structural relations, i.e. the I/O directions between submodels, as the models of the parent class digraph-models do. This difference is illustrated in Figure 4.5 and Figure 4.6.

```
;--Class Continuous-systems
(define-class continuous-systems
 (classvars)
 (instvars
     tflag
     tname
     tylist
     (vars (make-cstate 'cut '()
                        'mcut '()
                        'path '()
                        'mpath '()
           )
     )
 )
 (mixins digraph-models)
(options
  gettable-variables
  settable-variables
  inittable-variables)
)

;--define structure cstate for continuous-systems
(define-structure cstate cut mcut path mpath node
                         parameter
                         local terminal input output)
```

Fig. 4.7 Class definition of continuous-systems

The definition of the class continuous-systems is presented in Figure 4.7. This class inherits all the attributes from digraph-models with the newly defined instance variables.

4.4.2 Methods for Continuous-systems

The methods for the class "continuous-systems" are listed below. Explanations for these methods can be found in Appendix 2:

1. make-new (mname);
2. change-parameter (p-list);
3. change-ic (ic-list);
4. set-xxx (vvalue) and get-xxx;
5. build-composition-tree (m list-of-children);
6. build-system-tree (m list-of-children);
7. set-inf-dig (list-of-influencees);
8. set-int-coup (ch1 ch2 list-of-port-pairs);
9. set-ext-inp-coup (child list-of-port-pairs list-of-variable-pairs);
10. set-ext-out-coup (child list-of-port-pairs list-of-variable-pairs);
11. valid? (tf tn);
12. set-type-model;
13. set-system;
14. set-tylist (children);
15. set-lib (children);
16. set-subcomponent(children);
17. write-connection (ch1 ch2 list-of-port-pairs);
18. write-ext-inp (child list-of-port-pairs list-of-variable-pairs) and
    write-ext-out (child list-of-port-pairs list-of-variable-pairs);

19. write-statement.

It should be noticed that using *set-ext-inp-coup* or *set-ext-out-coup* would have identical results unless the I/O ports need to be specified explicitly for a continuous-systems model. Since a DEVS continuous model is an abstraction of a DYMOLA model, the question of whether a particular coupling is an input coupling or an output coupling is of no importance. DYMOLA will take care of this question, and automatically assign the computational structure to the model. As indicated in Chapter 3, I/O variables of a DYMOLA model can be implicitly declared as (non-directed) terminal variables.

4.4.3 Building Continuous-systems Models

In this subsection, a coupled model, named "net", is illustrated as an example of the way in which a continuous-systems model is built. The electrical network "net" is depicted in Figure 4.8.

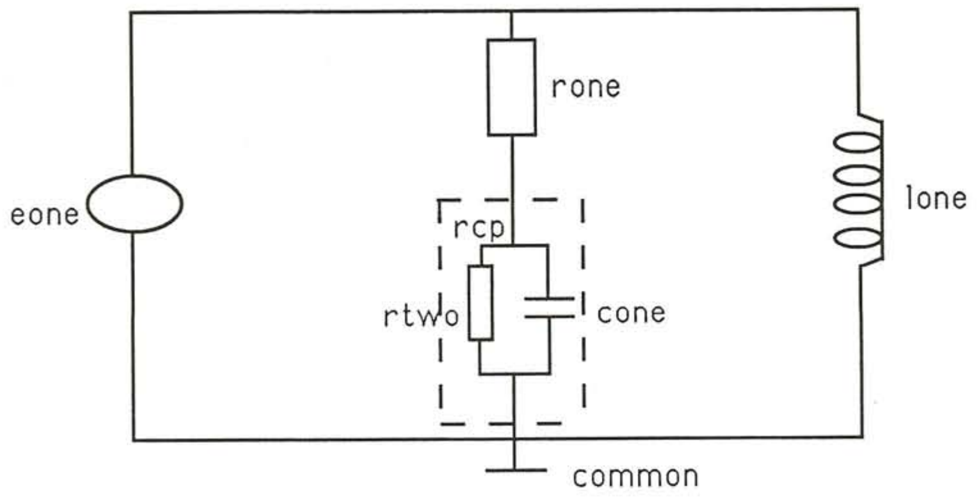


Fig. 4.8 The electrical network *net*

This network will also be used in Chapter 5 as an aid in explaining the System Entity Structure (SES) management of continuous models.

To clarify the hierarchy of model construction, the process of building a coupled type model "rcp" will be explained first. Then the model "net" will be built using "rcp" as one of its components.

Model "rcp" consists of an "rtwo" of type "resistor" and a "cone" of type "capacitor". The DEVS models for all the atomic components of "net" are shown in Figure 4.9. These models are separate files, with names of the form "modelname.m", e.g., resistor.m, capacitor.m, rone.m, etc., in the modelbase MBASE. For instance, in file "rone.m", it is first checked whether the type model "resistor" is already in the working memory by "(bound? ...)". If this is not the case, it loads the type model from MBASE into the working memory first. "(string-append ...)" is a function in SCHEME, concatenating several strings together. *ml* here is a defined global variable, representing the string of the directory of MBASE. Then a new model named "rone" which is the same as "resistor" is created by sending model "resistor" method *make-new*. *change-parameter* changes the value of parameter R of model "rone" from the default value of 0 (defined in resistor) to 10.

To define model "rcp" in DEVS-SCHEME (Figure 4.10), the component models are loaded first. *make-pair* generates a continuous-systems model in DEVS-SCHEME and attaches a coordinator (Appendix 1) to this model. After the model "rcp" is created, method *set-vars* is used to specify the variables for this model. Then method *build-composition-tree* is sent to "rcp" to establish its *composition-tree* and to begin setting up a type model named "rcp" with its subcomponents in DYMOBASE. Function *write-declare* writes the declaration part of a model into its DYMOLA model file. The *influence-digraph* of "rcp" is set by the method *set-inf-dig*, i.e., the information of submodel "rtwo" influences or is influenced by submodel

```
;-- RESISTOR --

(make-pair continuous-models 'resistor)
(send resistor valid? #t '())
(send resistor set-s (make-state 'cut '((A (VA / I)) (B (VB / -I)))
                                 'mcut '((C "[A B]"))
                                 'path '((PP "<A - B>"))
                                 'parameter '((R 0))
                                 'local '((V 0)))
)

;-- COIL --

(make-pair continuous-models 'coil)
(send coil valid? #t '())
(send coil set-s (make-state 'cut '((A (VA / I)) (B (VB / -I)))
                             'mcut '((C "[A B]"))
                             'path '((PP "<A - B>"))
                             'parameter '((L 0))
                             'local '((V 0)))
)

;-- CAPACITOR --

(make-pair continuous-models 'capacitor)
(send capacitor valid? #t '())
(send capacitor set-s (make-state 'cut '((A (VA / I)) (B (VB / -I)))
                                  'mcut '((C "[A B]"))
                                  'path '((PP "<A - B>"))
                                  'parameter '((C 0))
                                  'local '((V 0)))
)


;-- VOLTAGE --

(make-pair continuous-models 'voltage)
(send voltage valid? #t '())

;-- COMMON --

(make-pair continuous-models 'common)
(send common valid? #t '())


;-- RONE --

(if (unbound? resistor)
    (load (string-append ml "resistor.m")))
(send resistor make-new 'rone)
(send rone change-parameter '((R 10)))
```

Fig. 4.9 Examples of continuous-models in MBASE (for model *net*)

```
;-- RTWO --

(if (unbound? resistor)
    (load (string-append ml "resistor.m")))
(send resistor make-new 'rtwo)
(send rtwo change-parameter '((R 20)))

;-- LONE --

(if (unbound? coil)
    (load (string-append ml "coil.m")))
(send coil make-new 'lone)
(send lone change-parameter '((L 10)))

;-- CONE --

(if (unbound? capacitor)
    (load (string-append ml "capacitor.m")))
(send capacitor make-new 'cone)
(send cone change-parameter '((C 10)))

;-- EONE --

(if (unbound? voltage)
    (load (string-append ml "voltage.m")))
(send voltage make-new 'eone)
```

Fig. 4.9 Examples of continuous-models in MBASE (for model *net*)

(continued)

"cone". The coupling relations between submodels, and between "rcp" and its submodels, are done by methods *set-int-coup* and *set-ext-inp-coup/set-ext-out-coup*, respectively. Besides, method *set-int-coup* also converts the internal coupling relations into the connect statements of the corresponding DYMOLA model, while method *set-ext-inp/out-coup* writes onto the DYMOLA model file the external connect statements if the parameter *list-of-variable-pairs* is set to be nil; otherwise it writes the external coupling equations. To end the DYMOLA model file, method *write-statement* is used.

```
;-- Load the component models from model base

(load (string-append ml "rtwo.m"))
(load (string-append ml "cone.m"))

;-- Create the continuous-systems model rcp and its coordinator

(make-pair continuous-systems 'rcp)

;-- Define variables for rcp

(send rcp set-vars (make-cstate 'cut '((A (VA / I)) (B (VB / -I)))
                                'mcut '((C "[A B]"))
                                'path '((PP "<A - B>"))
                   )
)

;-- Specify the components for rcp

(send rcp build-composition-tree rcp (list rtwo  cone))

;-- Write declaration parts

(write-declare rcp)

;-- Specify the influencees of the model

(send rcp set-inf-dig (list (list rtwo cone)))

;-- Specify the internal couplings

(send rcp set-int-coup rtwo cone (list (cons 'A 'A)
                                       (cons 'B 'B)
                                 )
)

;-- Specify the external couplings

(send rcp set-ext-inp-coup rtwo (list (cons 'A 'A)) '())
(send rcp set-ext-out-coup rtwo (list (cons 'B 'B)) '())

;-- Close up the model

(send rcp write-statement)
```

Fig. 4.10 A continuous-systems model *rcp*

```
;-- Load the component models from model base

(load (string-append ml "eone.m"))
(load (string-append ml "common.m"))
(load (string-append ml "rone.m"))
(load (string-append ml "lone.m"))
(load (string-append ml "rcp.m"))

;-- Create the continuous-systems net and its coordinator

(make-pair continuous-systems 'net)

;-- Define variables for net

(send net set-vars (make-cstate 'input '((U '()))
                                'output '((Y1 '()) (Y2 '()))
                   )
)

;-- Specify the components for net

(send net build-system-tree net (list eone common rone lone rcp))

;-- Write declaration parts

(write-declare net)

;-- Specify the influencees of the model

(send net set-inf-dig (list (list eone rone lone rcp common)
                            (list rone rcp)
                      )
)

;-- Specify the internal couplings

(send net set-int-coup eone rone (list (cons 'B 'A)))
(send net set-int-coup eone lone (list (cons 'B 'A) (cons 'A 'B)))
(send net set-int-coup eone rcp (list (cons 'A 'B)))
(send net set-int-coup eone common (list (cons 'A 'COMMON)))
(send net set-int-coup rone rcp (list (cons 'B 'A)))

;-- Specify the external couplings

(send net set-ext-inp-coup eone (list (cons 'in 'B)) (list (cons 'U 'VB)))
(send net set-ext-out-coup rcp (list (cons 'A 'out1)) (list (cons 'VA 'Y1)))
(send net set-ext-out-coup lone (list (cons 'A 'out2)) (list (cons 'VA 'Y2)))

;-- Close up the model

(send net write-statement)
```

Fig. 4.11 A continuous-systems model *net*

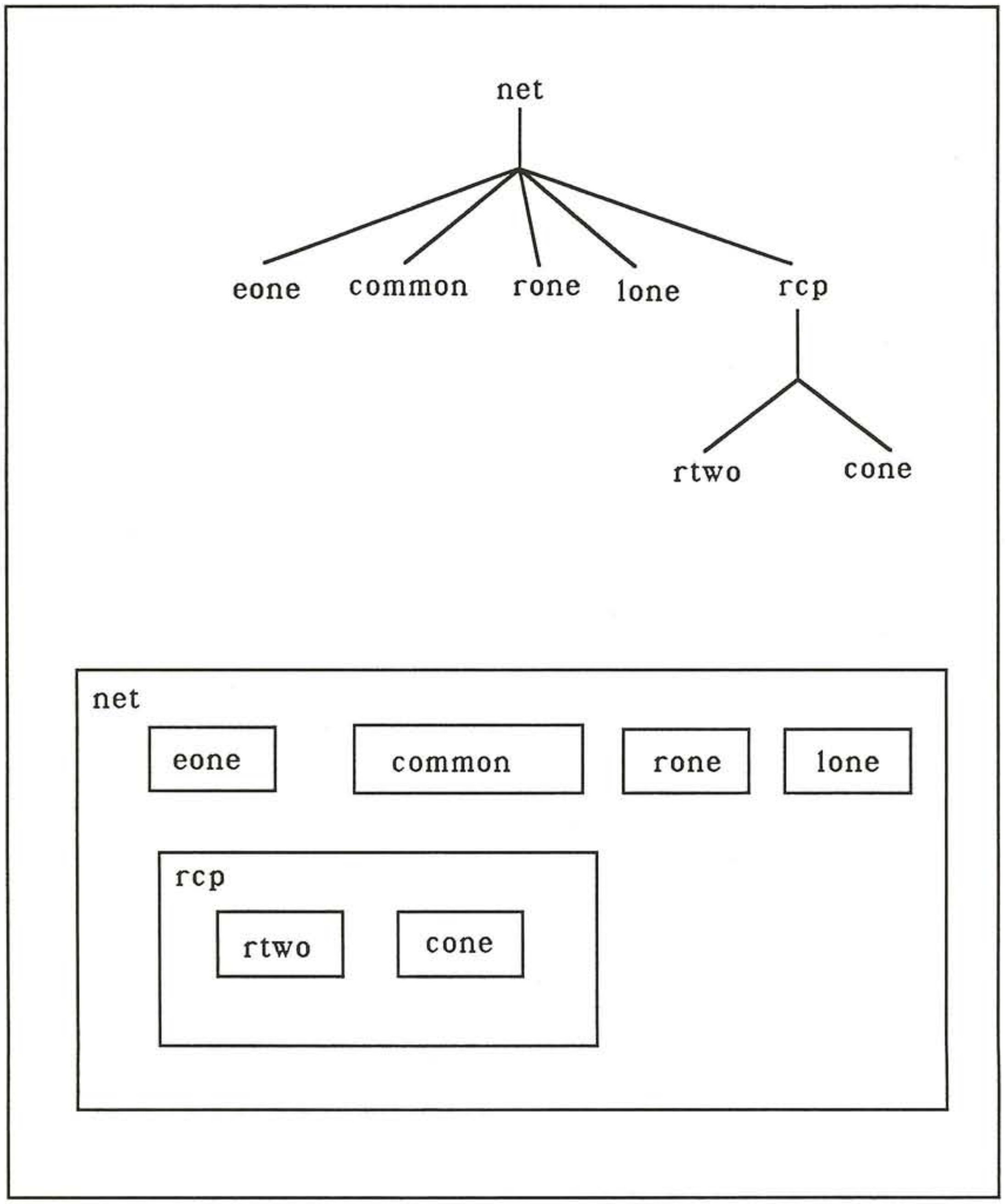


Fig. 4.12 Composition tree of *net*

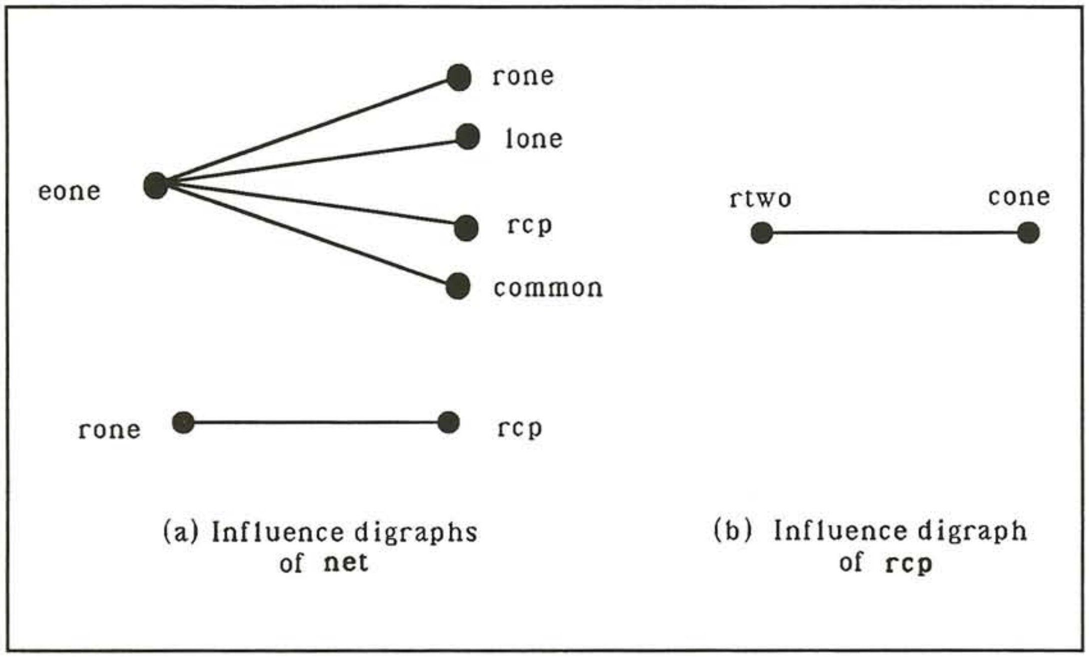


Fig. 4.13 Influence-digraph of *net*

```
A) File net.lib:

{  VOLTAGE  }
model type voltage
  cut A (VA / I) B (VB / -I)
  main cut C [A B]
  main path P <A - B>
  terminal V
  V = VB-VA
end

{  COMMON  }
model type common
  main cut COMMON (V / .)
  V = 0
end

{  COIL  }
model type coil
  cut A (VA / I) B (VB / -I)
  main cut C [A B]
  main path P <A - B>
  local V
  parameter L
  V = VA-VB
  L*der(I) = V
end
```

Fig. 4.14 Model *net* in DYMOBASE

```
{  RESISTOR  }
model type resistor
  cut A (VA / I) B (VB / -I)
  main cut C [A B]
  main path P <A - B>
  local V
  parameter R
  V = VA-VB
  R*I = V
end

{  CAPACITOR  }
model type capacitor
  cut A (VA / I) B (VB / .)
  main cut C [A B]
  main path P <A - B>
  local V
  parameter C
  V = VA-VB
  C*der(V) = I
end

model type rcp
  submodel (resistor) rtwo (20)
  submodel (capacitor) cone (10)
  cut A (VA / I)
  cut B (VB / -I)
  main cut C [A B]
  path PP <A - B>
  connect  rtwo:A at cone:A
  connect  rtwo:B at cone:B
  connect A at rtwo:A
  connect rtwo:B at B
end

B) File net.sys:

model net
  submodel (voltage) eone
  submodel common
  submodel (coil) lone (10)
  submodel (resistor) rone (10)
  submodel rcp
  input U
  output Y1
  output Y2
  connect  eone:B at rone:A
  connect  eone:B at lone:A
  connect  eone:A at lone:B
  connect  eone:A at rcp:B
  connect  eone:A at common:COMMON
  connect  rone:B at rcp:A
  eone.VB = U
  Y1 = rcp.VA
  Y2 = lone.VA
end
```

Fig. 4.14 Model *net* in DYMOBASE (continued)

Loading this coupled model from MBASE into DEVS results in a coupled model both in DEVS-SCHEME and DYMOBASE. The resultant DYMOLA model is the same as the one shown in Figure 3.9.

Model net consists of components "rone", "lone", "rcp" (the coupled model), "eone", and a common-point. The building procedure is explained in Figure 4.11. It should be noticed that, for model "net", the method *build-system-tree* is used instead of *build-composition-tree* used in "rcp". Both generate the *composition-tree* of the DEVS model. *build-system-tree* starts building a coupling model in DYMOLA, while *build-composition-tree* starts building a coupling type model.

In Figure 4.12 is the *composition-tree* of "net", and in Figure 4.13 arethe *influence-digraphs.* Comparing the network (Figure 4.8) with its *influence-digraphs* (Figure 4.13), it can be seen that the *influence-digraphs* depict the structural relations of "net" (not associated with ports). Also notice that different *influence-digraph* can be chosen as long as they are consistent and not repeated, so the ones in Figure 4.13 are not the only possible choice.

The outcome of loading model "net" is the generation of a coupled DYMOLA model consisting of two files, "net.lib" (library file for model "net") and "net.sys" (system file for model "net") (Figure. 4.14). With the existence of these two files and a simulation control model for "net", the executable DESIRE program can be generated.

4.5 Summary

In this chapter, two classes of continuous system models, as well as their instance variables and methods, have been introduced. The basic means for the management of continuous models in DEVS-SCHEME have been created. An example, model "net", is used to show the general approach of building a continuous coupled model. Appendix 3 provides information about functions and macros for managing continuous models in DEVS-SCHEME, which are not included in this chapter.

Even though it seems more involved to generate a coupled DYMOLA model from DEVS, the significant point is that, in DEVS-SCHEME, continuous models can be managed and then controlled. Moreover, it will be discussed in the next chapter that the user's work for generating a continuous coupled model will be considerably reduced by the System Entity Structure approach.

CHAPTER 5
THE SYSTEM ENTITY STRUCTURE MANAGEMENT OF CONTINUOUS SYSTEM MODELS

5.1 Introduction

A brief introduction of the System Entity Structure (SES) was given in Chapter 2, and the basic means for handling continuous models in DEVS-SCHEME was discussed in Chapter 4. In this chapter, a more detailed discussion of SES and its extension to the management of continuous models is being presented.

As stated in Chapter 2, the system entity structure is a knowledge representation scheme. While models possess the behavior knowledge of a system, the system entity structure holds the structural knowledge of a system. Furthermore, the system entity structure organizes the models in model bases in a systematic manner and provides means for retrieving and (re)using these models.

SES is a labeled tree with attached variable types. It satisfies the following axioms (Zeigler, 1984, 1989-b)

"a. uniformity: Any two nodes which have the same labels have identical attached variable types and isomorphic subtrees.

b. strict hierarchy: No label appears more than once down any path of the tree.

c. alternating mode: Each node has a mode which is either 'entity', 'aspect', or 'specialization'; if the mode of a node is entity then the modes of its successors are aspect or specialization, if the mode of a node is aspect or specialization, then the modes of its children are entity. The mode of the root is entity.

d. valid brothers: No two brothers have the same label.

e. attached variables: No two variable types attached to the same item have the same name.

f. inheritance: every entity in a specialization inherits all the variables, aspects and specializations from the parent of the specialization"

The SES is completely characterized by its axioms (Zeigler and Zhang, 1988).

Entity, aspect, and specialization are three kinds of nodes in SES to represent three types of structural knowledge about a system. An entity node corresponds to a model component that represents a real world object. The entity can have several aspects and/or specializations. An aspect node represents one decomposition of an entity out of many possibilities. A specialization node is used to represent the taxonomy of the system being modeled.

As DEVS-SCHEME realizes the DEVS formalism, ESP-SCHEME, underlying DEVS-SCHEME, realizes the SES formalism in the object-oriented environment. With operations defined in ESP-SCHEME, the creation, pruning and transformation of an SES are achieved.

Many descriptions of SES, its realization and applications may be found in Zeigler (1984, 1989-b), Rozenblit and Zeigler (1985) Rozenblit (1985), Rozenblit and Huang (1987), Rozenblit, Sevinc and Zeigler (1986), Kim (1988), Kim, Zeigler and Zhang (1988), Zeigler and Zhang (1988), Zhang (1988).

5.2 SES Representation of the Knowledge of a System

The simple electrical network "net" (Figure 4.8) is also used here to illustrate the SES representation of a system (Figure 5.1). In the SES tree, a single vertical line represents an aspect node, and a double vertical line represents a specialization node.

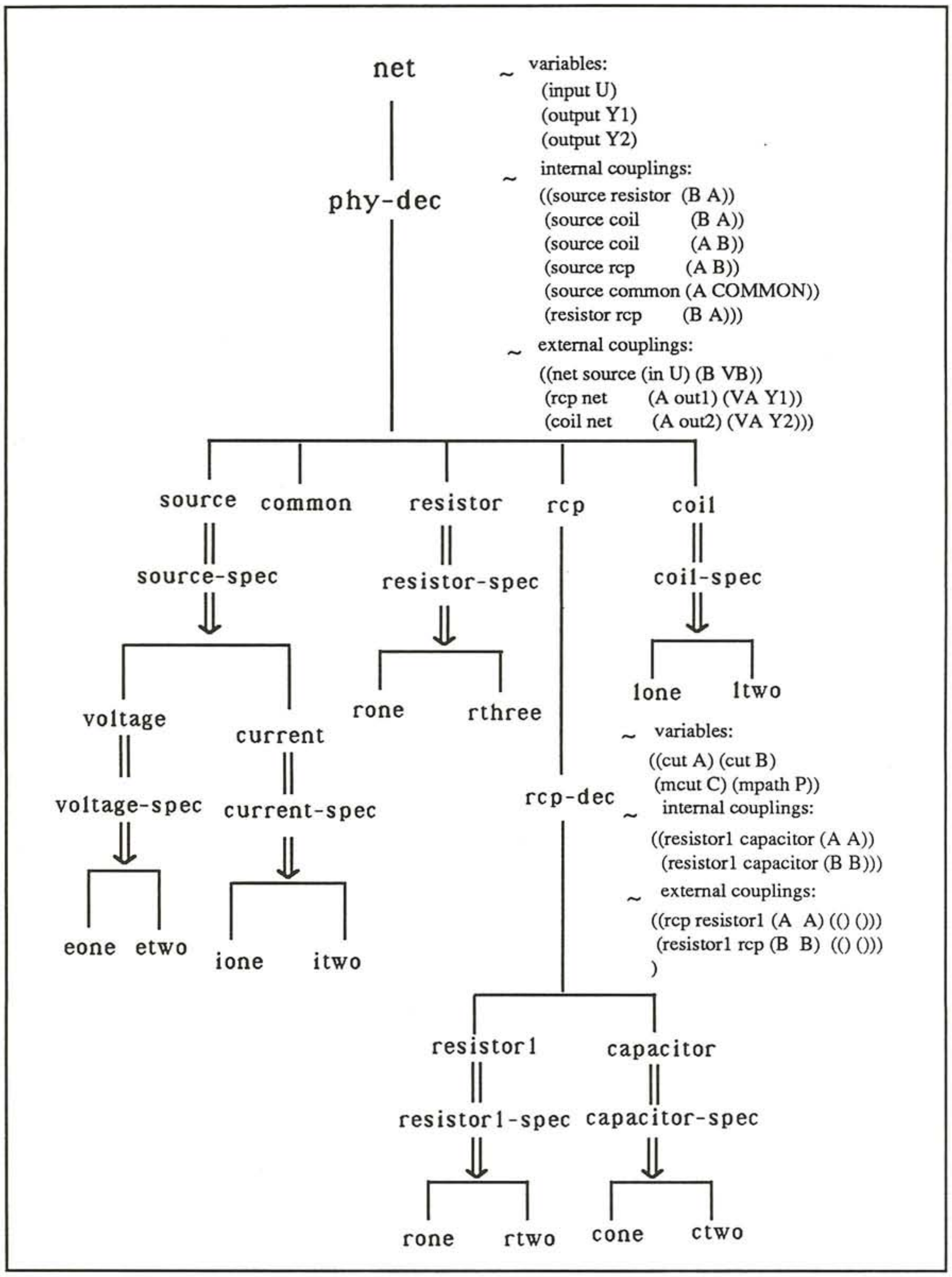


Fig. 5.1 The SES tree of *net*

In this example, the root entity is "net" and it has an aspect called *phy-dec* (short for physical decomposition). "net" is decomposed into five parts, i.e., "source", "common (point)", "resistor", "rcp" (parallel resistor and capacitor), and "coil". "source" is a generic type of the special types "voltage (source)" and "current (source)". "voltage" and "current" have their own distinctive attributes and inherit all the attributes (variables and structures) that "source" has. "voltage" and "current" are in turn the generic types with their specializations *voltage-spec* and *current-spec*, which are specialized into "eone", "etwo" and "ione", "itwo", respectively. The entity "rcp" has an aspect *rcp-dec*, specifying that it consists of "resistor1" and "capacitor", which are all generic types specialized into specific entities. "resistor" and "coil" are also generic types with entities "rone", "rthree" and "lone", "ltwo", respectively.

The coupling relationship defines how the entities (models) communicate with each other. Since aspects define the hierarchical decomposition of a system, the coupling relationship must be associated with aspects. In the example of the SES tree, it is shown that coupling relations are defined with the aspect nodes.

Specialization is a powerful way to represent many different variations of the same object. By specifying specializations, the SES organizes all possible alternatives of a system. The inheritance ensures that the specific individuals have the same coupling specification as their parents have.

The process *pruning* traverses the SES tree and selects one alternative among all according to the design objectives. A pruned entity structure (the one produced after *pruning*) is defined as an SES without specializations and with at most one aspect for each entity.

The creation of this SES tree in DEVS-SCHEME will be shown after discussing the operations for constructing an SES in the coming section.

5.3 Constructional Operations for the SES

The SES in ESP-SCHEME is realized by a module named *entity-structure* (Figure 5.2). A module is a package of local variables and operations. *items-list* and *branches-list* are main variables representing the tree structure of the SES. The variable *current-item* stands for the current node in the tree. Each item in the *items-list* is a structure (Figure 5.3), representing a node in the tree. Each branch in the *branches-list* is an ordered pair of items, specifying a node and its parent. Basic operations for the user to construct an SES are *set-current-item, add-item,* and *add-coupling.* Explanations and usages of these operations will be illustrated through an example. Other operations of constructing an SES for different classes of models can be found in Kim (1988) and Zeigler (1989).

As shown in Figure 5.3, the slots (fields) of an item structure are:

1. *name*: the name of a node;
2. *type*: the type of a node; this can be entity, aspect, or specialization (Section 5.1);
3. *coupling*: coupling relations associated with a node;
4. *priority-list*: sets up the priority of the node's children to deal with the situation that they can simultaneously send external events to their parent node.

The fields *sub-type, mult-coup-type,* and *num-mult-children*, which are related to the kernel-models (Appendix 1) will not be explained here. Interested readers are referred to Kim (1988) and Zeigler (1989).

Two new slots, *variable* and *cm-type* (shown italized in Figure 5.3) have been added to the structure item. Slot *variable* specifies the variables to be declared in DYMOLA models. Since there are two kinds of coupled DYMOLA models, a type

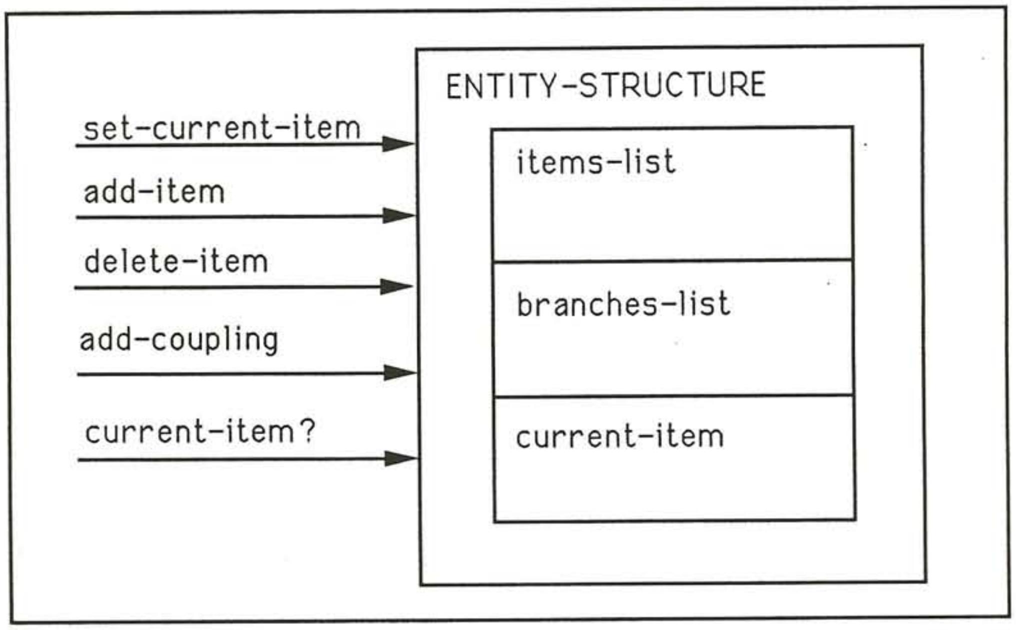


Fig. 5.2 Module *entity-structure*

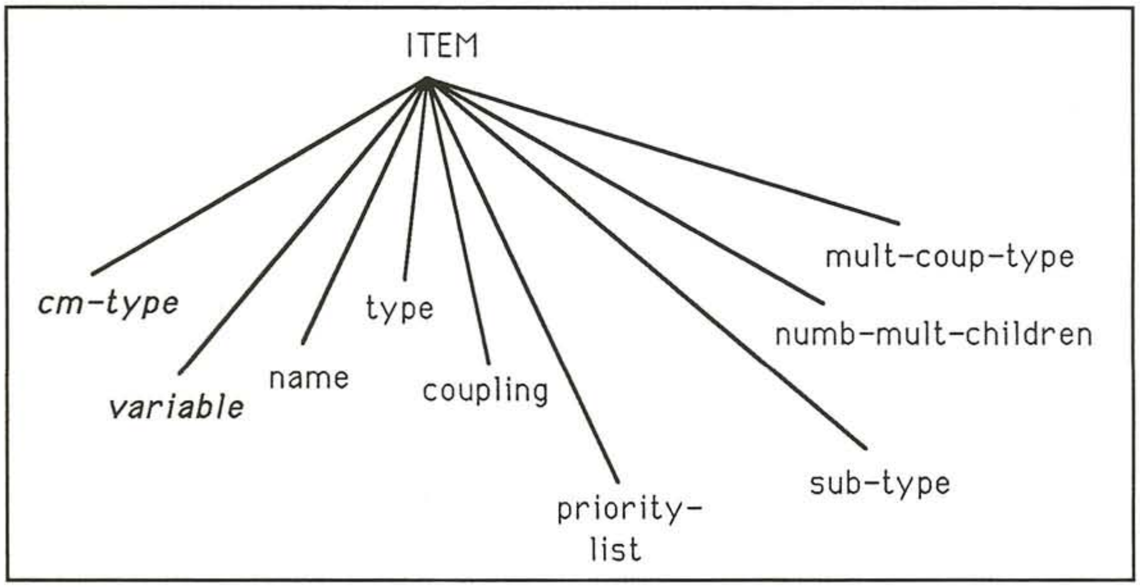


Fig. 5.3 Structure of *item*

model and a non-type model (Chapter 3 and Chapter 4), slot *cm-type* declares to which of the two a continuous coupled model in DYMOLA belongs.

Corresponding to the modified structure of an item, two new constructional operations, ***add-variable*** and ***set-c-system-type*** have been added to the original operations.

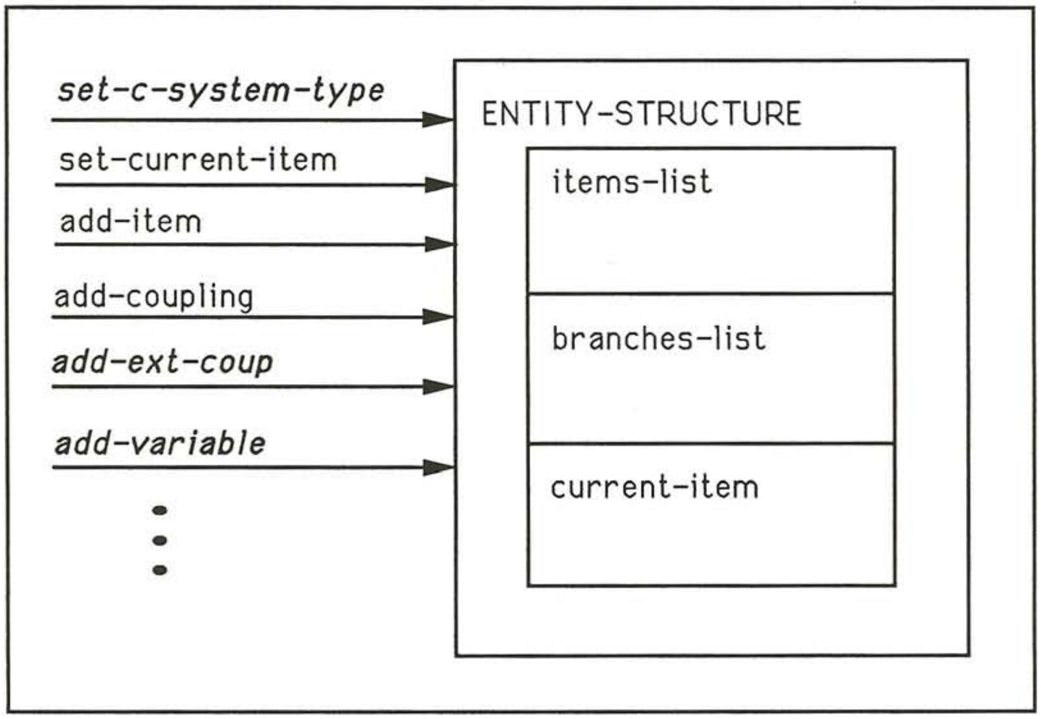


Fig. 5.4 Extended module of *entity-structure*

Originally, the ***add-couple*** was used to declare both internal and external couplings, i.e.,

(add-couple e:example 'node1 'node2 'out 'in),

where *example* is the name of the SES with the required prefix *e:*. *node*1 and *node*2 are a pair of nodes; *out* is a port of *node*1, and *in* is a port of *node*2. The difference between the internal and external coupling can be distinguished by the existence of a parent node in the pair. However, models of the class continuous-systems treat the external coupling differently from the models of class digraph-models (Chapter 4). The operation *add-ext-coup* has been introduced to specify the external coupling relationship for continuous-systems models. Figure 5.4 depicts the extended module of *entity-structure.*

Shown in Figure 5.5 is the realization of the SES of "net" (Figure 5.1) in DEVS-SCHEME. By first using *make-entstr*, the object "e:net" and the root entity "net" are created. The current item is now "net". At the root entity, *cm-type* is set to be *system* by *set-c-system-type.* This means that the model being built is not a type model. Then an aspect node called *phy-dec* is added by *add-item.* Operation *add-item* adds a new node to the tree, one parameter of this operation specifying the type of the node, the other giving a name to the node. *set-current-item* moves the pointer to a node. Notice that the pointer is moved to *phy-dec* first, then five new entity nodes of type *ent*, "source", "common", "resistor", "coil" and "rcp", are added to this node. A tree structure is built by using *set-current-item* and *add-item* repeatedly. Also notice that this tree must satisfy the axioms of the SES (Section 5.1).

As mentioned in the previous section, coupling specifications are associated with aspect nodes. Thus, adding couplings requires the current item be set to an aspect node first. The coupling specifications are accomplished by *add-couple* (for internal) and *add-ext-coup* (for external). The variables to be declared in the DYMOLA model are also added at the aspect node. Since these variables belong to the coupled model, they should be added in by *add-variable* before the system is decomposed.

```
;-- Entity Structure for NET

(make-entstr 'net)
(set-c-system-type e:net 'system)

(add-item e:net asp 'phy-dec)
(set-current-item e:net 'phy-dec)
(add-item e:net ent 'source)
(add-item e:net ent 'common)
(add-item e:net ent 'resistor)
(add-item e:net ent 'coil)
(add-item e:net ent 'rcp) ;rcp is a coupling model

;-- for source
(set-current-item e:net 'source)
(add-item e:net spec 'source-spec)
(set-current-item e:net 'source-spec)
(add-item e:net ent 'voltage)
(add-item e:net ent 'current)

;-- for voltage
(set-current-item e:net 'voltage)
(add-item e:net spec 'voltage-spec)
(set-current-item e:net 'voltage-spec)
(add-item e:net ent 'eone)
(add-item e:net ent 'etwo)

;-- for current
(set-current-item e:net 'current)
(add-item e:net spec 'current-spec)
(set-current-item e:net 'current-spec)
(add-item e:net ent 'ione)
(add-item e:net ent 'itwo)

;-- for resistor
(set-current-item e:net 'resistor)
(add-item e:net spec 'resistor-spec)
(set-current-item e:net 'resistor-spec)
(add-item e:net ent 'rone)
(add-item e:net ent 'rthree)

;-- for coil
(set-current-item e:net 'coil)
(add-item e:net spec 'coil-spec)
(set-current-item e:net 'coil-spec)
(add-item e:net ent 'lone)
(add-item e:net ent 'ltwo)

;-- for rcp
(set-current-item e:net 'rcp)
(add-item e:net asp 'rcp-dec)
(set-current-item e:net 'rcp-dec)
(add-item e:net ent 'resistor1)
(add-item e:net ent 'capacitor)
```

Fig. 5.5 The SES of model *net*

```
(set-current-item e:net 'resistor1)
(add-item e:net spec 'resistor1-spec)
(set-current-item e:net 'resistor1-spec)
(add-item e:net ent 'rone)
(add-item e:net ent 'rtwo)

(set-current-item e:net 'capacitor)
(add-item e:net spec 'capacitor-spec)
(set-current-item e:net 'capacitor-spec)
(add-item e:net ent 'cone)
(add-item e:net ent 'ctwo)

;-- Coupling and Variable specifications

;-- for NET
(set-current-item e:net 'phy-dec)

;-- Internal coupling

(add-couple e:net 'source 'resistor 'B 'A)
(add-couple e:net 'source 'coil 'B 'A)
(add-couple e:net 'source 'coil 'A 'B)
(add-couple e:net 'source 'rcp  'A 'B)
(add-couple e:net 'source 'common 'A 'COMMON)
(add-couple e:net 'resistor 'rcp 'B 'A)

;-- External coupling

(add-ext-coup e:net 'net 'source 'in 'B 'U 'VB)
(add-ext-coup e:net 'rcp 'net 'A 'out1 'VA 'Y1)
(add-ext-coup e:net 'coil 'net 'A 'out2 'VA 'Y2)

;-- Variable

(add-variable e:net 'input 'U '())
(add-variable e:net 'output 'Y1 '())
(add-variable e:net 'output 'Y2 '())

;---- for RCP
(set-current-item e:net 'rcp-dec)

(add-couple e:net 'resistor1 'capacitor 'A 'A)
(add-couple e:net 'resistor1 'capacitor 'B 'B)

(add-ext-coup e:net 'rcp 'resistor1 'A 'A '() '())
(add-ext-coup e:net 'resistor1 'rcp 'B 'B '() '())

(add-variable e:net 'cut 'A '(VA / I))
(add-variable e:net 'cut 'B "(VB / .)")
(add-variable e:net 'mcut 'c "[A B]")
(add-variable e:net 'mpath 'p "<A - B>")

;-- END --
```

Fig. 5.5 The SES of model *net* (continued)

The SES of "net" (Figure 5.1 and Figure 5.5) organizes all the knowledge about the electrical network "net". The "net" shown in Figure 4.8 is one possible realization of the SES which can be generated by pruning the SES.

5.4 SES Organization of Model Bases

The SES representation of a system was illustrated by the example "net". It has been shown that the SES provides means to organize all structural knowledge about a system. The composition of a model using the SES approach requires that the leaf entities in the tree, related to the real atomic models, must be built previously and reside in model bases. Two model bases, MBASE and DYMOBASE, were introduced in the previous chapters. These bases store the continuous DEVS and DYMOLA models in such a way that the DEVS models point to the DYMOLA models and have the declarative knowledge of the DYMOLA models. Besides these two model bases, there are yet other knowledge bases, ENBASE, DEBASE, and TRAJECT, in the knowledge based modeling and simulation environment.

Here are some explanations of all the knowledge bases mentioned above:

1. ENBASE : the entity structure base. All entity structure files are in this data base. The names of these files are

   xx.s (an entity structure source file),

   xx.e (a compiled entity structure file),

   and xx.p (a pruned entity structure file).

2. MBASE: the model base. The DEVS model files xx.m are stored in this data base.

3. DYMOBASE: the DYMOLA model base. It stores the DYMOLA files The file names are:

   xx. (a model file with no extension),

   xx.lib (a library file for a coupled model),

   xx.sys (a coupled model file which needs its model library be loaded first),

   and xx.ct (a simulation control model file)

4. DEBASE: the DESIRE model base. The executable DESIRE files xx.lst are saved in this data base.

5. TRAJECT: the trajectory base. The resultant simulation trajectories with file names like xx.tr are saved in this data base.

These bases are all subdirectories of a directory named DEVS.

With the existence of the component models in MBASE and DYMOBASE, effective management and manipulation of these models is necessary for them being used in many different systems. Also a proper tool is desired to organize these model bases. Operations performed on a once constructed SES, as presented in the following section, provide approaches to organizing information among these model bases as well as to synthesizing, managing and manipulating the (re)usable models in the model bases.

5.5 Operations on a Constructed SES Tree

5.5.1 General Procedures

Once an SES tree file is set up (Figure 5.5), it is saved in ENBASE with a file name such as "net.s". The command (load-ents "net.s") searches the file "net.s" in ENBASE and loads it into the working memory. The command *save-entstr* will save the compiled SES, pruned or complete, into ENBASE. A compiled complete SES is saved with a file name, e.g., "net.e". A pruned SES is saved with a file name like "net-a.p", "net-b.p", etc., where "-a" and "-b" denote different pruned versions (different choices). *load-entstr* loads the compiled SES, pruned or complete, into the working memory. For example, (load-entstr e:net) loads the file "net.e", and (load-entstr p:net-b) loads the file "net-b.p". The procedure *print-entstr* prints out the entity structure on the screen, for instance, (print-entstr e:net) or (print-entstr p:net-b).

5.5.2 Pruning: Generating Alternatives

An operation called *prune* can be applied to a complete SES to generate alternatives. *pruning* traverses the SES in the depth-first manner, interactively querying the user to select one entity if there is more than one choice under a specialization. It continues until all leaf entities have been visited. An example of pruning the SES tree net is shown in Figure 5.6.

```
1. (load-entstr e:net)
2. (prune e:net)
   give extension for pruned-entstr name :
3. a
   select starting entity from the following:
   (CTWO CONE RTWO CAPACITOR RESISTOR1
    LTWO LONE RTHREE RONE ITWO IONE ETWO EONE CURRENT
    VOLTAGE RCP COIL RESISTOR
    COMMON SOURCE NET)
4. net
working from entity NET
make this a leaf? (y/n)
5. n
  select an aspect from the following: (PHY-DEC)
  aspect PHY-DEC selected
  working from entity SOURCE
  select a specialization from the following: (SOURCE-SPEC)
  specialization SOURCE-SPEC is selected
  select an entity from the following: (VOLTAGE CURRENT)
6.voltage
  entity VOLTAGE from specialization SOURCE-SPEC selected
  working from entity VOLTAGE
  select a specialization from the following: (VOLTAGE-SPEC)
  specialization VOLTAGE-SPEC is selected
  select an entity from the following: (EONE ETWO)
7. eone
  entity EONE from specialization VOLTAGE-SPEC selected
  working from entity COMMON
  working from entity RESISTOR
  select a specialization from the following: (RESISTOR-SPEC)
  specialization RESISTOR-SPEC is selected
  select an entity from the following: (RONE RTHREE)
8. rone
  entity RONE from specialization RESISTOR-SPEC selected
  working from entity COIL
  select a specialization from the following: (COIL-SPEC)
  specialization COIL-SPEC is selected
  select an entity from the following: (LONE LTWO)
9. lone
  entity LONE from specialization COIL-SPEC selected
  working from entity RCP
  make this a leaf? (y/n)
10. n
  select an aspect from the following: (RCP-DEC)
  aspect RCP-DEC selected
  working from entity RESISTOR1
  select a specialization from the following: (RESISTOR1-SPEC)
  specialization RESISTOR1-SPEC is selected
  select an entity from the following: (RONE RTWO)
11. rtwo
  entity RTWO from specialization RESISTOR1-SPEC selected
```

Fig. 5.6 Pruning the SES to construct one choice of *net*

```
    working from entity CAPACITOR
    select a specialization from the following: (CAPACITOR-SPEC)
    specialization CAPACITOR-SPEC is selected
    select an entity from the following: (CONE CTWO)
  12. cone
    entity CONE from specialization CAPACITOR-SPEC selected

    save the pruned entity structure? (y/n)
  13. y
    Pruned entstr P:net-a made.
```

Fig. 5.6 Pruning the SES to construct one choice of *net*

(continued)

In this example, line 3 gives the extension of a pruned entity structure to denote one of the many possible choices. It is also shown in the example that whenever there is more than one choice, the user is asked to make a decision, e.g., line 6, 7, 8, etc. At last, the pruned entity structure is saved in ENBASE by the user answering y(es) (line 13).

It is seen that the purpose of the modeler directs the pruning procedure. The pruned SES tree is called a *pure entity structure*, in which each entity node has no more than one aspect and no specializations. The resultant tree in this example represents the specific net depicted in Figure 4.8. This pure entity structure is printed by (print-entstr p:net-a) as shown in Figure 5.7.

Currently, the pruning choices are made by the user. It is expected that rules can be associated with the entity structure and the automatic pruning will be realized by the help of a rule-based expert system. Work is being done to integrate

```
[5] (print-entstr p:net-a)
-ENT : NET
--ASP : PHY-DEC
          ,:::::coupling -> ((LONE NET (A . OUT2) (VA . Y2)) (RCP NET (A
. OUT1) (VA . Y1)) (NET EONE (IN . B) (U . VB)) (RONE RCP (B . A) ()) (EONE
COMMON (A . COMMON) ()) (EONE RCP (A . B) ()) (EONE LONE (A . B) ()) (EONE
LONE (B . A) ()) (EONE RONE (B . A) ()))
          ,:::::variable -> ((OUTPUT Y2 ()) (OUTPUT Y1 ()) (INPUT U ()))
---ENT : EONE
---ENT : COMMON
---ENT : RONE
---ENT : LONE
---ENT : RCP
----ASP : RCP-DEC
          ,:::::coupling -> ((RTWO RCP (B . B) (())) (RCP RTWO (A . A) (()))
(RTWO CONE (B . B) ()) (RTWO CONE (A . A) ()))
          ,:::::variable -> ((MPATH P <A - B>) (MCUT C [A B]) (CUT B (VB /
.)) (CUT A (VA / I)))
-----ENT : RTWO
-----ENT : CONE
end of display
()
```

Fig. 5.7 One pruned entity structure for *net*

```
[6] (transform p:net-a)

-- Do you want to continue the transformation of  the models
-- to get the executable continuous system simulation files? (y/n)
y

=============
root-co-ordinator: R:NET
-model: NET---> processor: C:NET
--model: EONE---> processor: S:EONE
--model: COMMON---> processor: S:COMMON
--model: RONE---> processor: S:RONE
--model: LONE---> processor: S:LONE
--model: RCP---> processor: C:RCP
---model: RTWO---> processor: S:RTWO
---model: CONE---> processor: S:CONE
=============          #<ENVIRONMENT>
```

Fig. 5.8 The composition tree *net* built by *transform*

a rule base system assisting in the pruning process (Rozenblit and Huang, 1987; Huang, 1987).

5.5.3 Transform: SES Approach to Building Continuous-systems Models

The operation *transform* is applied only to a pure entity structure. *transform* traverses the pruned entity structure from the top of the tree and calls upon a retrieve processor to search for a model of the current entity. A model is used if it is found, and the transformation of its subtree is aborted. Otherwise, the transformation continues. A *construct-continuous-systems* procedure will be invoked if the model of a non-leaf node cannot be found in the working memory, MBASE or ENBASE. In this way, a hierarchical continuous-systems model is constructed in the working memory, which is identical to the model built by the procedure discussed in Section 4.4.3.

In summary, the transformation does the following for continuous-systems models:

a. It constructs the DEVS continuous-systems model.

b. For a non-type model, it retrieves (or, if its coupled type submodels do not exist then constructs) the coupled type submodels needed for this model, collects all of them into the library file of this model, and saves this library file into DYMOBASE.

c. It constructs the corresponding DYMOLA model file and saves it into DYMOBASE.

d. It interactively asks whether the user wishes to continue the transformation to obtain an executable DESIRE program. If the answer is yes, *transform* will

```
----------------------------------------
-- CONTINUOUS SYSTEM net
----------------------------------------
-- STATE lone$I cone$V
-- DER dloe$I  dcoe$V
-- OUTPUT Y1 Y2
-- INPUT U
-- PARAMETERS and CONSTANTS:
L=1
rtwo$R=2
C=1
rone$R=1
-- INITIAL VALUES OF STATES:
lone$I=0
cone$V=0

----------------------------------------
TMAX=15 | DT=0.01 | NN=256
scale=1
XCCC=1
label TRY
 drunr | if XCCC<0 then XCCC=-XCCC | scale=2*scale | go to TRY
   else proceed
----------------------------------------
DYNAMIC
----------------------------------------
U = sin(t)
-- Submodel: common
coon$V = 0
-- Submodel: net
eoe$VA = coon$V
eoe$VB = U
loe$VA = eoe$VB
roe$VA = loe$VA
rcp$VB = coon$V
-- Submodel: rcp
coe$VB = rcp$VB
-- Submodel: rcp::cone
coe$VA = cone$V + coe$VB
-- Submodel: rcp
rto$VA = coe$VA
rcp$VA = rto$VA
-- Submodel: net
roe$VB = rcp$VA
-- Submodel: rone
rone$V = roe$VA - roe$VB
rone$I = rone$V/rone$R
-- Submodel: net
eone$I = lone$I + rone$I
-- Submodel: eone
eone$V = eoe$VB - eoe$VA
-- Submodel: net
loe$VB = rcp$VB
-- Submodel: lone
lone$V = loe$VA - loe$VB
d/dt lone$I = lone$V/L
```

Fig. 5.9 DESIRE program for *net*

```
-- Submodel: rcp
rto$VB = coe$VB
-- Submodel: rcp::rtwo
rtwo$V = rto$VA - rto$VB
rtwo$I = rtwo$V/rtwo$R
-- Submodel: net
rcp$I = rone$I
-- Submodel: rcp
cone$I = rcp$I - rtwo$I
-- Submodel: rcp::cone
d/dt cone$V = cone$I/C
-- Submodel: net
Y1 = rcp$VA
Y2 = loe$VA
----------------------------------------
OUT
dispt Y1, Y2
----------------------------------------
/--
/PIC 'net.PRC
/--
```

Fig. 5.9 DESIRE program for *net* (continued)

call on DYMOLA and execute a DYMOLA batch file which has been generated during the transformation. Finally, an executable DESIRE program is generated and stored in DEBASE.

After all this work, *transform* creates a *root-coordinator* for the root entity and initializes it to the root entity's *co-ordinator*.

Figure 5.8 shows the composition tree generated by the transformation of the pruned entity structure "p:net-a". Figure 5.9 presents the generated DESIRE program for "net".

It has been shown that there are two methods to build a continuous-systems model: the bottom up method discussed in Chapter 4, and the top down approach using the SES. The SES, a knowledge representation scheme, reduces the modeler's amount of work significantly. More important, it systematically organizes the knowledge of a system and serves as an effective tool to manage information of the system among the model bases.

5.5.4 Simulation from the SES

The pruned entity structures are stored in ENBASE, for instance file "net-a.p". Simulating the model net directly from this pruned SES needs first (load-entstr p:net-a), then (transform p:net-a). The command (run p:net-a) will execute DESIRE, in which the user loads the program file "net-a.lst" and types "run" to simulate. The required simulation result, which was specified in the simulation control model of DYMOLA, will be saved in the TRAJECT data base.

5.5.5 Some Utilities to Simplify the Use of the SES

The simulation of a laboratory in a Space Station managed by robots requires that the continuous simulation results can be directly mapped into the DEVS model in a desired way. This can be achieved by assigning appropriate internal transition, external transition, and time advance functions to the DEVS models.

Some useful macros are provided in this subsection to facilitate the generation of a DEVS (discrete event) model that incorporates the continuous simulation results. These macros help to determine a *time-window*, which is selected by executing various continuous simulation runs using different parameter values and/or initial conditions, and/or disturbances. The time-windows concept employed by Zeigler (1989-a) in the event-based control is explained briefly in Appendix 5.

The macro *get-p-time* runs a continuous simulation from a pruned entity structure and returns its process time. This time information is also saved in a model state ***p-time***.

get-p-list runs a continuous simulation from a pruned entity structure and returns the parameter (value) list and process time. The parameter list and the time information are also saved in a model state *p-list.*

get-p-table runs series of simulations from a pruned entity structure and returns a table of parameter lists and simulation times of each simulation run. This table of parameter lists and simulation times is also saved in a model state *p-table.*

get-p-window runs series of simulations from a pruned entity structure and returns a time-window. The table of parameter lists and simulation times is saved in a model state *p-table.*

States *p-time, p-list,* and *p-table* are generated when the corresponding macros are called. When calling a macro, the user has to specify one of the models on the pruned entity structure tree, in which these states can be stored.

A system can be simulated by different simulation requirements. The conditions are provided in different simulation control models of the same system. Various simulation control models are distinguished by a numbered suffix. For instance, "net.1", "net.2", etc. While calling these macros, the user has to specify by number the simulation control model he/she wants. These macros will select the desired control model and rename it to the file name of the simulation control model, e.g., "net.ct". The termination condition for the simulation can also be specified in these macros. This helps a DEVS model, e.g., that of a robot, to control the real simulation in DESIRE.

The time-windows information generated by use of the above described macros can then be fed back into the DEVS (discrete event) models to allow DEVS to perform a qualitatively similar, more highly aggregated discrete event simulation of the formerly continuous model. Such a simulation can be executed using the

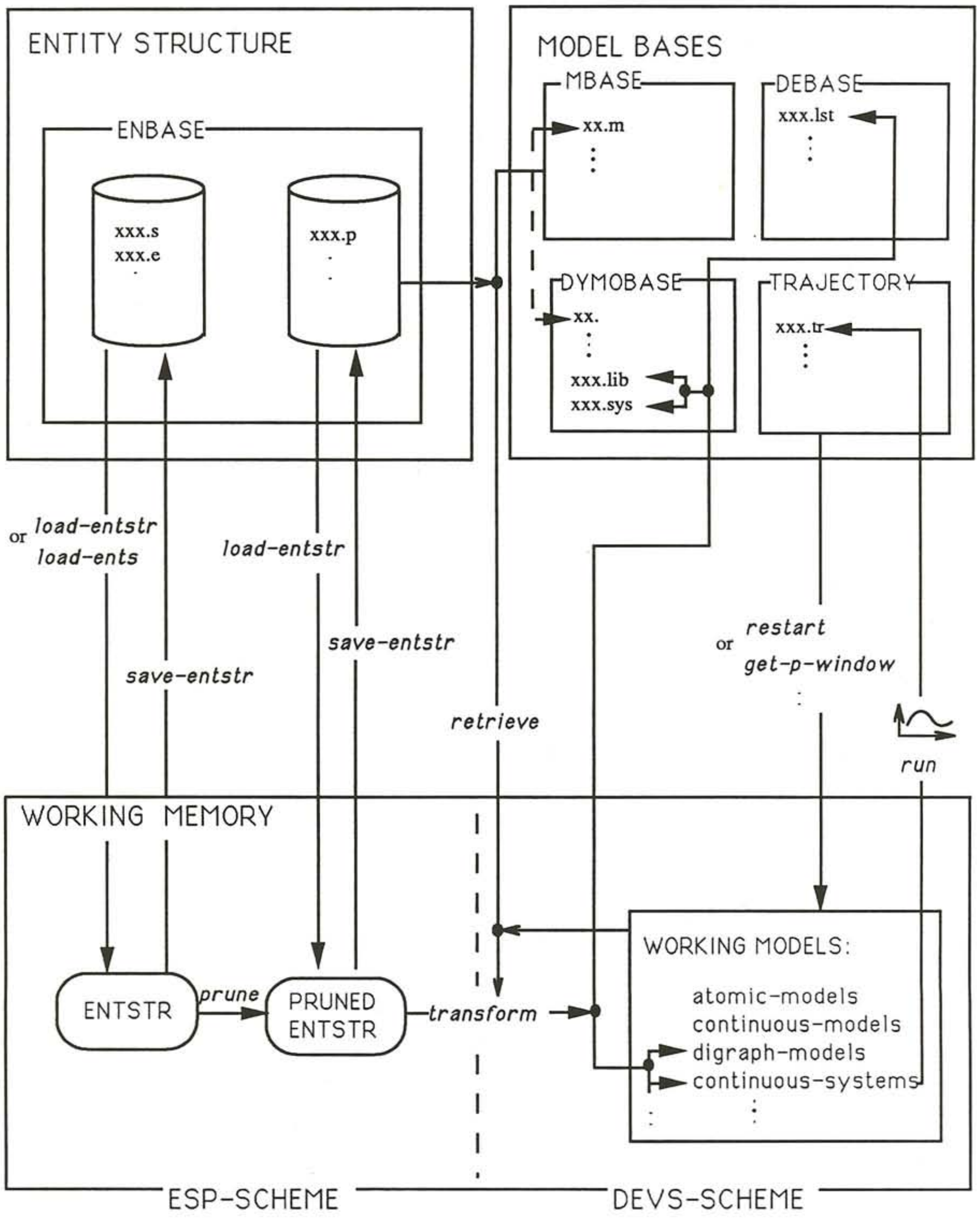


Fig. 5.10 The knowledge-based modeling and simulation environment

(restart r:net) command once the transition functions of the atomic discrete models (coded inside the continuous-models) are defined.

Detailed formats and explanations for each of these macros are provided in Appendix 4.

5.6 The Extended Knowledge-Based Modeling and Simulation Environment

The system entity structure and model base together form a knowledge-based modeling and simulation environment (Kim, 1988; Zeigler, 1989). With the research done in this thesis, an extended knowledge-based modeling and simulation environment has been realized, which encompasses the management of continuous system models. Figure 5.10 visualizes the resultant environment into a picture.

In this environment, the SES organizes models in the model bases; *pruning* produces a desired system in ENBASE; *transform* converts the pruned SES into a coupled DEVS model as well as a hierarchical DYMOLA model, and calls upon DYMOLA to flatten the hierarchy into a monolithic description, generating a DESIRE program; finally, *run* executes the DESIRE program which generates time histories to be stored in the trajectory base. Several macros are also provided to assist in simulating a continuous system model directly from a pure entity structure and mapping the time information into DEVS models.

CHAPTER 6
GENERATING TIME-WINDOWS FOR EVENT-BASED INTELLIGENT CONTROL

In designing a simulation environment capable of supporting the study of robot organizations for managing chemical, or similar, laboratories aboard Space Station Freedom, a thorough study of the problems to be encountered in assigning the responsibilities of managing a non-life-critical, but mission-valuable, process to an organized group of robots is needed. For instance, handling fluids in orbit will be essential to many experiments being planned in manufacturing and biotechnology. Therefore, fluid handling in microgravity has been chosen as the focus of the laboratory environment (Zeigler, Cellier and Rozenblit, 1988).

In the project, a robot model consists of three parts: a motion-system, a sensory-system, and a cognition-system. The cognition-system contains one selector and several Model-Plan-Units (MPUs). The selector is a controller which controls MPUs. MPUs are task specialists which are activated under the appropriate circumstances (Zeigler, 1989-c). For instance, one MPU is specialized for the task of fluid handling. The instruments needed are a pressurized bladder bottle and a syringe (Sarjoughian, 1989). To monitor and thus control the process, the robot has to have knowledge about certain states of the model of the bottle. In order to model the robot's recognition of the process, several models of the same bottle are needed. These models are related by abstraction (Zeigler, 1989-c).

Figure 6.1 shows that a MPU is decomposed into two parts, an "operator" and a "diagnoser". There are three models of the bottle: "btl-e", "btl-o", and "btl-d", for the fluid handing MPU. "btl-e" is the bottle model external to the MPU, representing the real bottle. Since the simulation of the process is continuous, "btl-e" is an abstraction of the model bottle in DYMOBASE. "btl-o" is the operational model of the bottle, similar to the way that a human being views a bottle being operated on. "btl-d" is the bottle model to be diagnosed, mimicking the way that a human being checks the states of the bottle when something abnormal happens. "btl-o" and "btl-d" are different abstractions from "btl-e" (Zeigler, 1989-c).

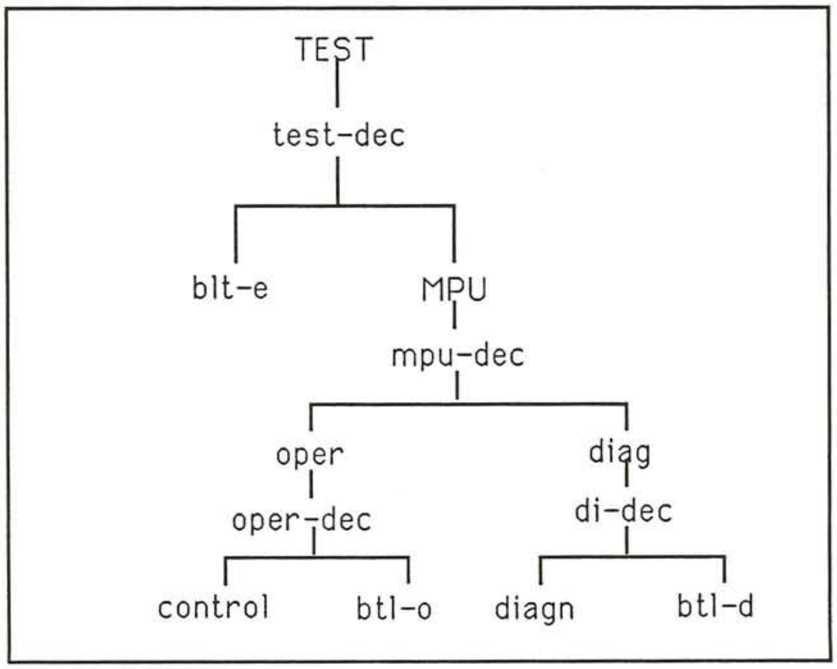


Fig. 6.1 SES for testing a bottle handling *mpu*

The application to be explained in this chapter demonstrates how the continuous models for the fluid handling process are organized and manipulated in the modeling and simulation environment, and how the continuous simulation results of the fluid handling process are mapped into a DEVS model. Here, time-windows will be generated from the series of simulations and will be referred to by the DEVS model "btl-e".

6.1 The Fluid Handling System with Experimental Frame

The model of a fluid handling process operated by a robot aboard the Space Station Freedom consists of a model of the physical equipment, and the experimental frame operating on that equipment (Section 2.2).

The equipment consists of a syringe (Figure 6.2) and a pressurized bladder bottle (Figure 6.3). Since air/liquid interfaces are not allowed under microgravity conditions (unless they are controlled by surface tension), standard earth-bound containers, such as beakers, cannot be used. The bottle, a space adapted "beaker", contains an inflatable bladder. Two processes, filling and emptying, are considered. Liquid is injected from the syringe into the bladder during the filling process, and extracted from the bladder into the syringe during emptying. The air pressure between the bladder and the wall of the bottle indicates the volume of the liquid in the bladder.

The experimental frame consists of a generator and a transducer. The robot pushes the plunger of the syringe with velocity V during the filling process, and pulls the plunger with velocity $-V$ during emptying. The generator generates this input, i.e., the velocity of the system, and the transducer gathers the outputs and analyzes the results. Different generators and transducers can be used for different experimental environments.

Shown in Figure 6.4 is the diagram of the system "fh" (fluid handling system with experimental frame).

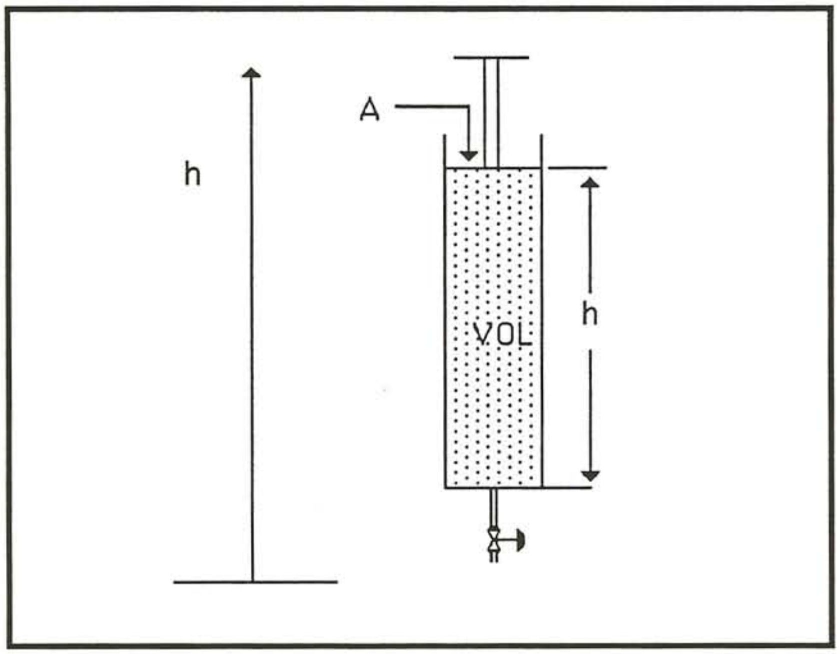


Fig. 6.2 The *syringe*

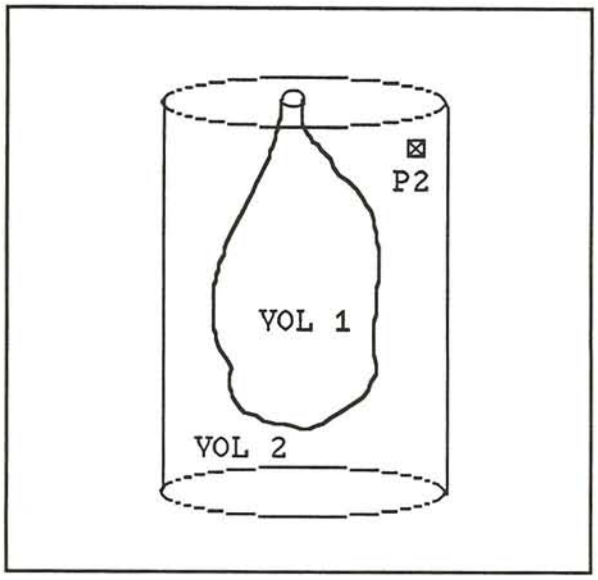


Fig. 6.3 The *pressurized bladder bottle*

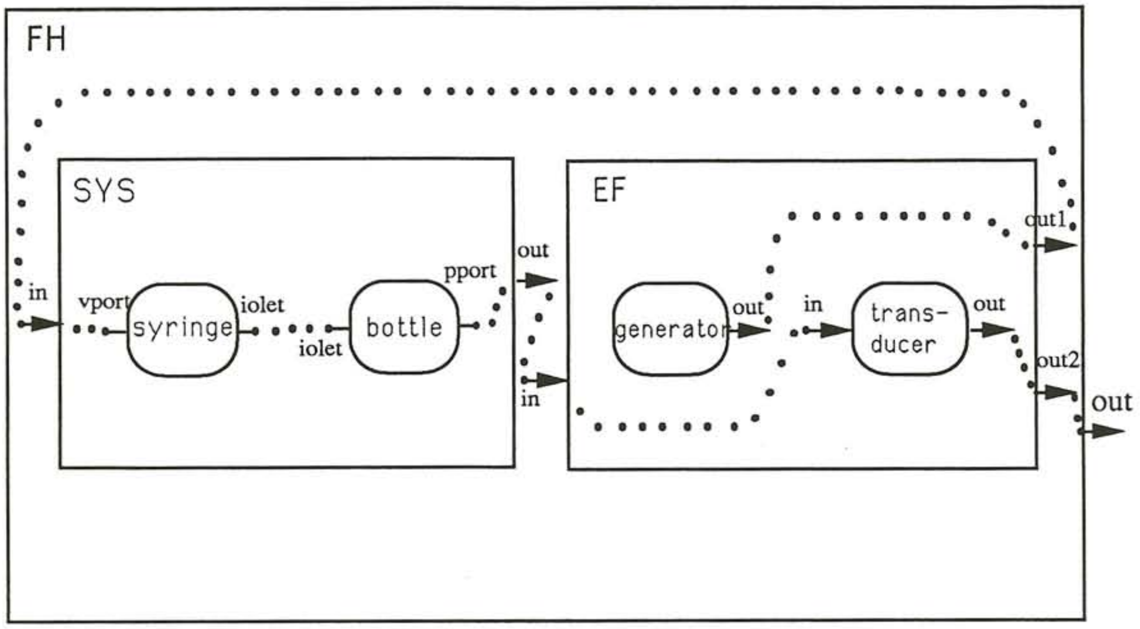


Fig. 6.4 The fluid handling system with experimental frame

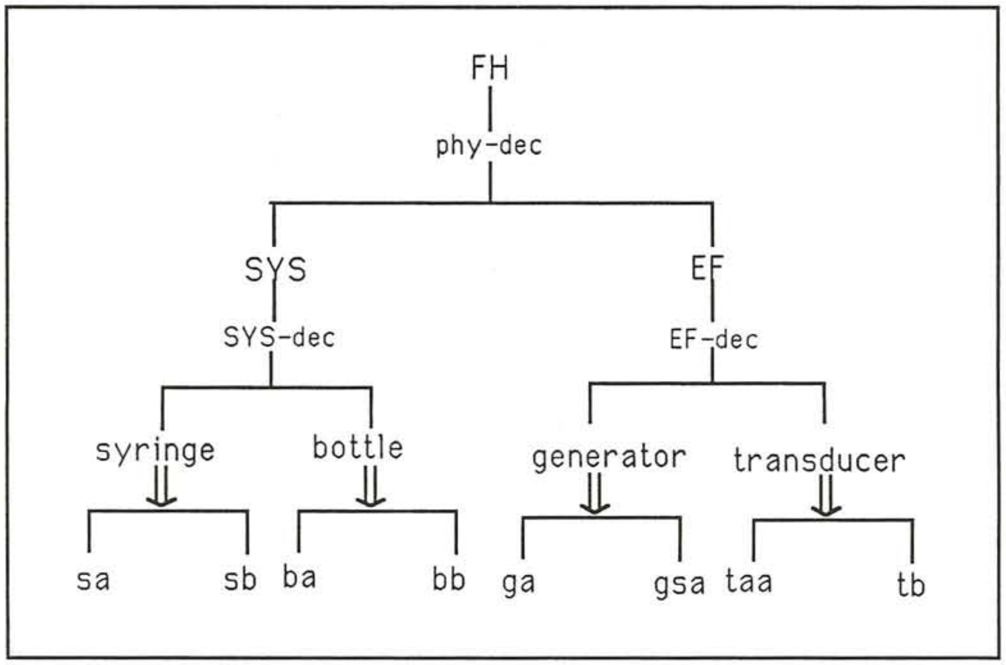


Fig. 6.5 The system entity structure of system *fh*

6.2 The System Entity Structure of FH

The SES of system "fh" is depicted in Figure 6.5, and its textual description is shown in Figure 6.6. It can be seen that all the component models in the system have two specializations. For instance, the generator "ga" generates a constant input to the system while the generator "gsa" generates a sine wave input to the system.

Figure 6.7 shows the pruned SES of "fh". The original and the pruned entity structures of "fh" are saved in ENBASE.

6.3 Component Models in the Model Bases

Only those models selected during pruning will be shown in this section.

6.3.1 Models in DYMOBASE

The component model files in DYMOBASE are listed below:

bottle.

syringe.

gen.

transedu.

To simplify the modeling process, it is assumed that the input, the nominal velocity V of the syringe plunger, is constant (in the pruned SES, "sa" is chosen). Accordingly, also the nominal flow rate of the syringe is constant during the process. To demonstrate the flexibility of the modeling scheme of extracting time-windows from continuous simulation runs which represent variations in simulation time due to parameter changes of a model, the model allows the actual flow rate into the bottle (actually into the bladder) to vary in a nonlinear fashion with the fluid volume in the bladder, with the nominal flow rate that is specified as an input parameter, and

```
;-- Entity Structure for FH

(make-entstr 'fh)
(set-c-system-type e:fh 'system)
(ai e:fh asp 'phy-dec)
(sci e:fh 'phy-dec)
(ai e:fh ent 'sys)
(ai e:fh ent 'ef)

;-- for sys
(sci e:fh 'sys)
(ai e:fh asp 'sys-dec)
(sci e:fh 'sys-dec)
(ai e:fh ent 'syringe)
(ai e:fh ent 'bottle)

(sci e:fh 'syringe)
(ai e:fh spec 'syringe-spec)
(sci e:fh 'syringe-spec)
(ai e:fh ent 'sa)
(ai e:fh ent 'sb)

(sci e:fh 'bottle)
(ai e:fh spec 'bottle-spec)
(sci e:fh 'bottle-spec)
(ai e:fh ent 'ba)
(ai e:fh ent 'bb)

;-- for ef
(sci e:fh 'ef)
(ai e:fh asp 'ef-dec)
(sci e:fh 'ef-dec)
(ai e:fh ent 'generator)
(ai e:fh ent 'transducer)

(sci e:fh 'generator)
(ai e:fh spec 'generator-spec)
(sci e:fh 'generator-spec)
(ai e:fh ent 'ga)
(ai e:fh ent 'gsa)

(sci e:fh 'transducer)
(ai e:fh spec 'transducer-spec)
(sci e:fh 'transducer-spec)
(ai e:fh ent 'taa)
(ai e:fh ent 'tb)

;-- Coupling and Variable specifications
;-- for FH
(sci e:fh 'phy-dec)

;-- Internal coupling
(acp e:fh 'ef 'sys 'OUT1 'IN)
(acp e:fh 'sys 'ef 'OUT 'IN)

;-- External coupling
(acpx e:fh 'ef 'fh 'OUT2 'OUT 'Y2 'Y)
```

Fig. 6.6 SES file of *fh*

```
;-- Variable
(av e:fh 'output 'Y '())

;---- for sys
(sci e:fh 'sys-dec)
(acp e:fh 'syringe 'bottle 'IOLET 'IOLET)
(acpx e:fh 'sys 'syringe 'IN 'VPORT 'U 'V)
(acpx e:fh 'bottle 'sys 'PPORT 'OUT 'P 'Y)
(av e:fh 'cut 'IN "(U / .)")
(av e:fh 'cut 'OUT "(Y / .)")

;---- for ef
(sci e:fh 'ef-dec)
(acpx e:fh 'ef 'transducer 'IN 'IN 'U 'U)
(acpx e:fh 'generator 'ef 'OUT 'OUT1 'Y 'Y1)
(acpx e:fh 'transducer 'ef 'OUT 'OUT2 'Y 'Y2)
(av e:fh 'cut 'IN "(U / .)")
(av e:fh 'cut 'OUT1 "(Y1 / .)")
(av e:fh 'cut 'OUT2 "(Y2 / .)")

;-- END --
```

Fig. 6.6 SES file of *fh* (continued)

```
-ENT : FH
--ASP : PHY-DEC
          ,:::::coupling -> ((EF FH (OUT2 . OUT) (Y2 . Y)) (SYS EF
                             (OUT . IN) ()) (EF SYS (OUT1 . IN) ()))
          ,:::::variable -> ((OUTPUT Y ()))
---ENT : SYS
----ASP : SYS-DEC
          ,:::::coupling -> ((BA SYS (PPORT . OUT) (P . Y)) (SYS SA (IN
                             . VPORT) (U . V)) (SA BA (IOLET . IOLET) ()))
          ,:::::variable -> ((CUT OUT (Y / .)) (CUT IN (U / .)))
-----ENT : SA
-----ENT : BA
---ENT : EF
----ASP : EF-DEC
          ,:::::coupling -> ((TAA EF (OUT . OUT2) (Y . Y2)) (GA EF (OUT
                             . OUT1) (Y . Y1)) (EF TAA (IN . IN) (U . U)))
          ,:::::variable -> ((CUT OUT2 (Y2 / .)) (CUT OUT1 (Y1 / .)) (CUT
                             IN (U / .)))
-----ENT : GA
-----ENT : TAA
end of display
```

Fig. 6.7 Screen printout of the pruned SES of *fh*

(one alternative)

with other extraneous influencing factors. Three specific extraneous effects were considered. One was that the bladder could have a leak, the fluid would then leak out of the bladder into the space between the bladder and the wall of the bottle. The second effect involved the angle at which the needle of the syringe penetrated the diaphragm covering the opening of the bottle. It was thought that if this angle were very obtuse (the needle of the syringe is almost parallel to the diaphragm), the needle would not penetrate the diaphragm completely, and therefore, a fraction of the fluid ejected from the syringe would escape, instead of being injected into the bottle. The third consideration was that the flow rate of the liquid into the bottle would slow down when the bladder was almost full. These assumptions might or might not be realistic; they were included to show the ability of the modeling scheme to handle situations of this kind. For simplicity, all these effects are included in the bottle model rather than the syringe model. The effects are implemented by changing the parameters of the bottle model, as described next.

The model descriptions in DYMOLA are shown in Figure 6.8. All these are generic models declared as model type (Section 3.2). Text enclosed between curly braces are comments.

In the model "bottle", it is shown that the ports of the bottle illustrated in Figure 6.4 are declared as *cuts.* Variable $VOL1$ denotes the volume of the bladder, and $VOL2$ denotes the volume between the bladder and the wall of the bottle. VOL, a constant, denotes the total volume of the bottle. Fluid can flow in or out through the port $IOLET$. The input/output variable at port $IOLET$ is the nominal flow rate W. The actual flow rate $WREAL$ is influenced by the factors mentioned above. The effect of the angle on the flow rate is described as a tabular function $TAB1$, and that of the fluid volume in the bladder is a tabular function $TAB2$ for filling, and another tabular function $TAB3$ for emptying. The effect of

```
{ MODEL BOTTLE }
model type bottle
  cut IOLET(W /.)
  cut PPORT(P /.)
  local VOL1 VOL2
  local FA FVF FVE RATE WREAL
  { SC = 1 : Filling }
  { SC = 0 : Empty   }
  parameter R=8.314 M=0.00224 TEMP=273.15 VOL=50.24
  parameter ANGLE=90 LR=0 SC=1
  RATE = VOL1/VOL
  func FA = TAB1(ANGLE)
  func FVF = TAB2(RATE)
  func FVE = TAB3(RATE)
  WREAL = W*FA*(SC*FVF + (1-SC)*FVE) - LR
  der(VOL1) = WREAL
  VOL2 = VOL - VOL1
  P = R*M*TEMP/VOL2*1000000
end

{  MODEL SYRINGE  }
model type syringe
  cut VPORT (V / .)
  cut IOLET (-W / .)
  parameter A=3.14
  local VOL=50
  W = -A*V
  der(VOL) = W
end

{  MODEL GEN  }
model type gen
{ SC > 1 : Y > 0 }
{ SC = 1 : Y = 0 }
{ SC < 1 : Y < 0 }
  cut OUT(Y /.)
  parameter X=5 SC=1.5
  Y=sgn(SC-1)*X
end

{  MODEL TRANSDU  }
model type transdu
  cut IN(U / .)
  cut OUT(Y / .)
  parameter X
  Y=X*U
end
```

Fig. 6.8 Component models of *fh* in DYMOBASE

leakage is described through the variable LR. Values of these tabular functions are declared in the simulation control model. According to the gas law, the pressure of the gas is proportional to the mass of the gas and the temperature, and is inversely proportional to the volume of the gas. The variable P at port $PPORT$ indicates the pressure between the bladder and the wall which is related to the volume of the fluid in the bladder. The generator "gen" generates the nominal velocity of the syringe plunger. The nominal flow rate of the syringe is the product of its cross-section area and the velocity of its plunger. The transducer "transdu" in our example simply rescales its input variable which is the pressure P.

A more detailed study of these models could be performed. Modifications can be made to the individual models, and different types of models can be chosen without changing the system entity structure.

6.3.2. Models in MBASE

Component model files in MBASE are:

bottle.m

ba.m

syringe.m

sa.m

gen.m

ga.m

transdu.m

taa.m

Figure 6.9 shows their model descriptions in DEVS.

```
(make-pair continuous-models 'bottle)
(send bottle valid? #t '())
(send bottle set-s (make-state 'sigma '-
                               'phase #t
                               'tflag #t
                               'tname 'bottle
                               'cut '((PPORT '(P)) (IOLET '(W)))
          'parameter '((R 8.314) (M 0.00224) (TEMP 273.15) (VOL 50.24)
                       (SC 1) (ANGLE 90) (LR 0))
                               'local '((VOL1 0) (VOL2 '())
                                        (FA '()) (FVF '()) (FVE '())
                                        (RATE '()))))
)

;-- MODEL BA --
(if (unbound? bottle)
    (load (string-append ml "bottle.m")))
(send bottle make-new 'ba)

;-- MODEL SYRINGE --
(make-pair continuous-models 'syringe)
(send syringe valid? #t '())
(send syringe set-s (make-state 'sigma '-
                                'phase #t
                                'tflag #t
                                'tname 'syringe
                                'cut '((VPORT '(V/.)) (IOLET '(-W/.)))
                                'parameter '((A 3.14))
                                'local '((VOL 50)))
)

;-- MODEL SA --
(if (unbound? syringe)
    (load (string-append ml "syringe.m")))
(send syringe make-new 'sa)

;-- MODEL GEN --
(make-pair continuous-models 'gen)
(send gen valid? #t '())
(send gen set-s (make-state 'sigma '-
                                'phase #t
                                'tflag #t
                                'tname 'gen
                                'cut '((OUT '(Y)))
                                'parameter '((X 5) (SC 1.5)))
)

;-- MODEL GA --
(if (unbound? gen)
    (load (string-append ml "gen.m")))
(send gen make-new 'ga)
```

Fig. 6.9 Component models of *fh* in MBASE

```
;-- MODEL TRANSDU --
(make-pair continuous-models 'transdu)
(send transdu valid? #t '())
(send transdu set-s (make-state 'sigma '-
                                'phase #t
                                'tflag #t
                                'tname 'transdu
                                'cut '((IN '(U)) (OUT '(Y)))
                                'parameter '((X '()))))
)

;-- MODEL TAA --
(if (unbound? transdu)
    (load (string-append ml "transdu.m")))

(send transdu make-new 'taa)
(send taa change-parameter '((X 0.1)))
```

Fig. 6.9 Component models of *fh* in MBASE (continued)

6.4 Getting Time-windows from Simulations of the System

With the existence of a pruned entity structure in ENBASE and the component models in MBASE and DYMOBASE, the utility macros introduced in Chapter 5 can be used to manage the continuous simulation and obtain the desired time trajectories.

In this application, different parameter values within the range of normal operating conditions are assigned to the model "syringe" and the model "bottle". Time-windows are then determined by the maximum and minimum simulation time recorded for various values of a model parameter. Two examples are shown in this section that were used to obtain the time-windows for the filling and emptying processes.

To make the simulation more efficient, i.e., save the time needed for transforming the pruned entity structure, the parameter changes were specified in the

```
cmodel
simutime 10
step 0.01
commupoints 100

ctblock
connect 'fh.tr' as output 2
dimension TAB1[12],TAB2[10],TAB3[10]
data 0, 25, 45,  80,  85, 90
data 0, 0,  0.2, 0.8, 1,  1
data 0, 0.9, 0.95, 0.99, 1
data 1, 1,   0.9,  0.1,  0
data 0, 0.01, 0.05, 0.1, 1
data 0, 0.1, 0.9, 1, 1
read TAB1
read TAB2
read TAB3
dimension lra[5]
data 0, 0.1, 0.04, 0.06, 0.08
read lra
dimension angle[5]
angle[1] = 90
angle[2] = 65
for i=3 to 5
 angle[i]=abs(ran(0))*30+70
next
for i=1 to 5
 ANGLE=angle[i]
 LR=lra[i]
 drun
 write #2,ANGLE,LR,t
 reset
next
disconnect 2
ctend

outblock
OUT
```

Fig. 6.10 Simulation control model *fh.1*

simulation control model (Figure 6.10). The file name of this simulation control model is "fh.1". Other simulation control models of the same system were named "fh.2", "fh.3", etc. When calling the macro *get-p-window*, a test number is specified to indicate the particular simulation control model to be used. The macro then renames the selected test to "fh.ct" which makes it the simulation control model used during the simulation (Chapter 3). In file "fh.1", array $TAB1$ specifies the influence of the angle of the syringe needle on the flow rate into and out of the bottle, and arrays $TAB2$ and $TAB3$ specify the influence that the volume of the liquid in the bladder has on the flow rate during fillying and emptying. Different values of parameter LR (leakage rate) are assigned in array *lra*. Different values of parameter $ANGLE$ (angular influence) are randomly generated within the acceptable range (Monte Carlo technique).

In example 1 (Figure 6.11), the initial conditions and parameters for the component models of system "fh" are set to meet the requirements of filling. Macro *get-p-window* (Chapter 5 and Appendix 4) sets the initial volume of the bladder to zero, the test number to 1 (file "fh.1" is chosen to be the simulation control model), and executes five separate simulation runs the parameter values of which are specified in "fh.1". The simulation terminates when the system output reaches a value of $2.14E + 6$. The system output is the output of "transdu", which is the rescaled pressure of bottle "ba" (refer to Figure 6.4). The value $2.14E + 6$ of the rescaled pressure indicates that the bladder in "ba" is full.

```
;-- Get filling time window

(define (example-1)
; loading the model into working memory
   (if (unbound? ba)
       (load (string-append ml "ba.m")))
   (if (unbound? sa)
       (load (string-append ml "sa.m")))
   (if (unbound? ga)
       (load (string-append ml "ga.m")))
; for filling the generator generates V > 0
   (send ga change-parameter '((SC 1.5)))
; set the initial volume of liquid in syringe to be full
   (send sa change-ic '((VOL 50.24)))
; set the parameter SC of bottle to be 1 for filling
   (send ba change-parameter '((SC 1)))

; set the initial condition for VOL1 of bottle to be zero and
  get time window
   (eval '(get-p-window p:fh-a ba '((vol1 0))
                                   '(ad lr) 1 5 "fh$Y-2.14E+6")))
```

Fig. 6.11 Procedure to get filling time-window

Typing "(example-1)" sets the necessary parameters and initial conditions for the component models and calls upon the macro. The macro does the transformation and simulation of the pruned entity structure and finally returns the time-window. The result is shown in Figure 6.12. To get the parameter values of the model for every simulation and the simulation time, method *get-sv* can be sent to model bottle (Figure 6.12).

Example 2 (Figure 6.13) shows the procedures for getting the time-window of emptying. For emptying, the same system is used without changing the component models. The initial conditions and some switching parameters have to meet the emptying conditions. Other parameter changes, which influence the flow rate, are chosen to be the same as for filling, so that the same simulation control model

```
[2] (example-1)

(FH ROOT-ASP)
(SYS PHY-DEC FH ROOT-ASP)
(SA SYS-DEC SYS PHY-DEC FH ROOT-ASP)
(BA SYS-DEC SYS PHY-DEC FH ROOT-ASP)
(EF PHY-DEC FH ROOT-ASP)
(GA EF-DEC EF PHY-DEC FH ROOT-ASP)
(TAA EF-DEC EF PHY-DEC FH ROOT-ASP)
-- Do you want to continue the transformation of  the models
-- to get the executable continuous system simulation files? (y/n)
y

=============
root-co-ordinator: R:FH
-model: FH---> processor: C:FH
--model: SYS---> processor: C:SYS
---model: SA---> processor: S:SA
---model: BA---> processor: S:BA
--model: EF---> processor: C:EF
---model: GA---> processor: S:GA
---model: TAA---> processor: S:TAA
=============

-- Do you want to save another trajectory
-- besides the basic one ?
   (y/n)
n

(3.64 6.97)

[3] (send ba get-sv 'p-table)

    (((ANGLE 9.00000E+01) (LR 0.00000E+00) 3.64000E+00)
     ((ANGLE 6.50000E+01) (LR 0.1) 6.97000E+00)
     ((ANGLE 7.00000E+01) (LR 0.04) 5.86000E+00)
     ((ANGLE 7.00010E+01) (LR 0.06) 5.96000E+00)
     ((ANGLE 7.66445E+01) (LR 0.08) 5.06000E+00)
    )
```

Fig. 6.12 Result from extracting filling time-window

(fh.1) can be used. The simulation terminates when the system output has decreased to a value of $1.0125E + 4$ (the bladder is now considered empty). Figure 6.14 shows the result, i.e., the returned time-window.

Besides returning the time-window, the macro also produces model files of the system, simulation files, and trajectory files in DYMOBASE, DEBASE, and TRAJECT. These results are shown in Figure 6.15, 6.16, and 6.17, respectively.

In this chapter, only the results of the filling process are shown. The emptying process generates similar files.

```
;-- Get emptying time window

(define (example-2)
; loading the model into working memory
   (if (unbound? ba)
       (load (string-append ml "ba.m")))
   (if (unbound? sa)
       (load (string-append ml "sa.m")))
   (if (unbound? ga)
       (load (string-append ml "ga.m")))
; for emptying the generator generates V<0
   (send ga change-parameter '((SC 0.5)))
; set the initial volume of liquid in syringe to be zero
   (send sa change-ic '((VOL 0)))
; set the parameter SC of bottle to be 0 for empty
   (send ba change-parameter '((SC 0)))
; set initial conditions as full for VOL1 of bottle and get time-window
   (eval '(get-p-window p:fh-a ba '((vol1 50))
                                   '(ad lr) 1 5 "-fh$Y+1.0125E+4")))
```

Fig. 6.13 Procedure to get emptying time-window

```
[5] (example-2)
(FH ROOT-ASP)
(SYS PHY-DEC FH ROOT-ASP)
(SA SYS-DEC SYS PHY-DEC FH ROOT-ASP)
(BA SYS-DEC SYS PHY-DEC FH ROOT-ASP)
(EF PHY-DEC FH ROOT-ASP)
(GA EF-DEC EF PHY-DEC FH ROOT-ASP)
(TAA EF-DEC EF PHY-DEC FH ROOT-ASP)
-- Do you want to continue the transformation of  the models
-- to get the executable continuous system simulation files? (y/n)
y
=============
root-co-ordinator: R:FH
-model: FH---> processor: C:FH
--model: SYS---> processor: C:SYS
---model: SA---> processor: S:SA
---model: BA---> processor: S:BA
--model: EF---> processor: C:EF
---model: GA---> processor: S:GA
---model: TAA---> processor: S:TAA
=============
-- Do you want to save another trajectory
-- besides the basic one ?
   (y/n)
n
(1 7.58)
```

Fig. 6.14 Result from extracting emptying time-window

```
Result in DYMOBASE:

(Generation of file "fh.lib", "fh.sys" and coupling type model files "sys.",
                         "ef.")
A) File fh.lib

{  MODEL SYRINGE  }
model type syringe
  cut VPORT (V / .)
  cut IOLET (-W / .)
  parameter A=3.14
  local VOL=50
  W = -A*V
  der(VOL) = W
end

{ MODEL BOTTLE  }
model type bottle
  cut IOLET(W /.)
  cut PPORT(P /.)
  local VOL1 VOL2
  local FA FVF FVE RATE WREAL
  { SC = 1 : Filling }
  { SC = 0 : Empty   }
  parameter R=8.314 M=0.00224 TEMP=273.15 VOL=50.24
  parameter ANGLE=90 LR=0 SC=1
  RATE = VOL1/VOL
  func FA = TAB1(ANGLE)
  func FVF = TAB2(RATE)
  func FVE = TAB3(RATE)
  WREAL = W*FA*(SC*FVF + (1-SC)*FVE) - LR
  der(VOL1) = WREAL
  VOL2 = VOL - VOL1
  P = R*M*TEMP/VOL2*1000000
end

model type sys
  submodel (syringe) sa(ic VOL=50.24 )
  submodel (bottle) ba
  cut IN (U / .)
  cut OUT (Y / .)
  connect  sa:IOLET at ba:IOLET
  sa.V = U
  Y = ba.P
end

{ MODEL GEN  }
model type gen
{ SC > 1 : Y > 0 }
{ SC = 1 : Y = 0 }
{ SC < 1 : Y < 0 }
  cut OUT(Y /.)
  parameter X=5 SC=1.5
  Y=sgn(SC-1)*X
end
```

Fig. 6.15 Generated models in DYMOBASE

```
{ MODEL TRANSDU  }
model type transdu
  cut IN(U / .)
  cut OUT(Y / .)
  parameter X
  Y=X*U
end

model type ef
  submodel (gen) ga
  submodel (transdu) taa(0.1)
  cut IN (U / .)
  cut OUT1 (Y1 / .)
  cut OUT2 (Y2 / .)
  taa.U = U
  Y2 = taa.Y
  Y1 = ga.Y
end

B) File "fh.sys"

model fh
  submodel sys
  submodel ef
  output Y
  connect  sys:OUT at ef:IN
  connect  ef:OUT1 at sys:IN
  Y = ef.Y2
end

C) File "sys."

model type sys
  submodel (syringe) sa(ic VOL=50.24 )
  submodel (bottle) ba
  cut IN (U / .)
  cut OUT (Y / .)
  connect  sa:IOLET at ba:IOLET
  sa.V = U
  Y = ba.P
end

D) File "ef."

model type ef
  submodel (gen) ga
  submodel (transdu) taa(0.1)
  cut IN (U / .)
  cut OUT1 (Y1 / .)
  cut OUT2 (Y2 / .)
  taa.U = U
  Y2 = taa.Y
  Y1 = ga.Y
end
```

Fig. 6.15 Generated models in DYMOBASE (continued)

```
Result in DEBASE:
(Generation of file "fh.lst")

----------------------------------------
-- CONTINUOUS SYSTEM fh
----------------------------------------
-- STATE sa$VOL VOL1
-- DER dsaVOL  dVOL1
-- OUTPUT fh$Y
-- PARAMETERS and CONSTANTS:
A=3.14
R=8.314
M=0.00224
TEMP=273.15
ba$VOL=50.24
ANGLE=90
LR=0
ba$SC=1
ga$X=5
ga$SC=1.5
taa$X=0.1
-- INITIAL VALUES OF STATES:
sa$VOL=50.24
VOL1=0

----------------------------------------
TMAX=10 | DT=0.01 | NN=100
connect 'fh.tr' as output 2
dimension TAB1[12],TAB2[10],TAB3[10]
data 0, 25, 45,  80,  85, 90
data 0, 0,  0.2, 0.8, 1,  1
data 0, 0.9, 0.95, 0.99, 1
data 1, 1,   0.9,  0.1,  0
data 0, 0.01, 0.05, 0.1, 1
data 0, 0.1, 0.9, 1, 1
read TAB1
read TAB2
read TAB3
dimension lra[5]
data 0, 0.1, 0.04, 0.06, 0.08
read lra
dimension angle[5]
angle[1] = 90
angle[2] = 65
for i=3 to 5
 angle[i]=abs(ran(0))*30+70
next
for i=1 to 5
 ANGLE=angle[i]
 LR=lra[i]
 drun
 write #2,ANGLE,LR,t
 reset
next
disconnect 2
```

Fig. 6.16 Generated program in DEBASE

```
---------------------------------------
DYNAMIC
---------------------------------------
-- Submodel: ef::ga
ga$Y = sgn(ga$SC - 1)*ga$X
-- Submodel: ef
Y1 = ga$Y
-- Submodel: fh
sys$U = Y1
-- Submodel: sys
V = sys$U
-- Submodel: sys::sa
sa$W = -A*V
d/dt sa$VOL = sa$W
-- Submodel: sys
ba$W = -sa$W
-- Submodel: sys::ba
VOL2 = ba$VOL - VOL1
P = R*M*TEMP/VOL2*1000000
func FA = TAB1,ANGLE
RATE = VOL1/ba$VOL
func FVF = TAB2,RATE
func FVE = TAB3,RATE
WREAL = ba$W*FA*(ba$SC*FVF + (1 - ba$SC)*FVE) - LR
d/dt VOL1 = WREAL
-- Submodel: sys
sys$Y = P
-- Submodel: fh
ef$U = sys$Y
-- Submodel: ef
taa$U = ef$U
-- Submodel: ef::taa
taa$Y = taa$X*taa$U
-- Submodel: ef
Y2 = taa$Y
-- Submodel: fh
fh$Y = Y2
---------------------------------------
OUT
term fh$Y-2.14E+6
---------------------------------------
/--
/PIC 'fh.PRC
/--
```

Fig. 6.16 Generated program in DEBASE (continued)

```
Result in TRAJECTORY:
(Generation of file "fh.tr")

 9.00000E+01  0.00000E+00  3.64000E+00
 6.50000E+01  1.00000E-01  6.97000E+00
 7.00000E+01  4.00000E-02  5.86000E+00
 7.00010E+01  6.00000E-02  5.96000E+00
 7.66445E+01  8.00000E-02  5.06000E+00
```

Fig. 6.17 Simulation results in TRAJECT

CHAPTER 7
CONCLUSIONS

An extended knowledge-based modeling and simulation environment has been demonstrated in this MS thesis. The management of continuous models in DEVS-SCHEME is realized to meet the requirements of the modeling and simulation of a robot-managed laboratory aboard Space Station Freedom.

By exploiting DYMOLA, a hierarchical structured continuous modeling language, it was possible to relate the abstracted DEVS models and the target continuous simulation language code (in DESIRE) to each other. This enables the user to switch back and forth between the discrete event and the continuous system modeling concepts, exploit the advantages of both, and ensure the consistency of his modeling effort across the barrier between the two modeling methodologies.

The system entity structure organizes all the models at different representation levels. Operations on the system entity structure provide a systematic way to synthesize and prune the system entity structure in accordance with the user's needs, to automatically generate the continuous system model in DYMOLA, to produce a continuous simulation program in DESIRE, and finally to execute the DESIRE simulation program, generate time histories for state and/or algebraic model variables, and store the time histories in the TRAJECT data base.

DESIRE simulations can be guided to automatically explore a set of parameter values, and generate time-windows for use in a subsequent equivalent discrete event DEVS simulation of the same model.

An event-based intelligent control can be realized in this knowledge-based multi-facetted modeling environment. Details can be found in Appendix 5. Thereby, a continuous process control can be interfaced with a symbolic reasoning system.

Continuous and discrete event system modeling and simulation can be merged with AI techniques in this environment.

The following is suggested for future research:

1. Elaboration of DYMOLA to make it a more powerful hierarchical continuous system modeling tool and simulation language generator (Section 3.4).
2. Realization of the automatic transformation of continuous models into equivalent discrete event models or vice versa, at different levels of the hierarchy, even at the atomic model level.
3. A more user-friendly interface.

APPENDIX 1

BRIEF EXPLANATIONS OF SOME CLASSES IN DEVS-SCHEME

Entities: the root class in DEVS-SCHEME, providing basic tools for manipulating objects in DEVS classes. All classes in DEVS-SCHEME are subclasses of this class.

Models: subclass of entities, providing basic means for modeling.

Processors: subclass of entities, providing basic means for simulation.

Atomic-models: subclass of models, realizing the atomic level of the DEVS model formalism (Zeigler, 1984).

Continuous-models: subclass of atomic-models. A continuous-models model is another level of representation for a real continuous atomic model written in DYMOLA.

Coupled-models: subclass of models, embodying the hierarchical model composition constructs of the DEVS formalism.

Digraph-models: subclass of coupled-models, providing ways of specifying a finite set of explicitly given components with explicitly specified coupling relations. Methods are available to build the composition-tree and influence-digraph which encode the external and internal coupling relations respectively.

Continuous-systems: subclass of digraph-models, providing means to manage continuous coupled models in DEVS.

Kernel-models: subclass of coupled-models, providing convenient facilities for constructing models with arbitrary numbers of components, which are all generated from a prototype, the kernel, and coupled in a uniform manner.

Broadcast-models: subclass of kernel-models. All children of a broadcast coupled model communicate directly with each other and with the outside world.

Cellular-models: subclass of kernel-models, providing means for coupling of a fixed or variable set of geometrically located cells, each of which is connected to other cells in a uniform way.

Hypercube-models: subclass of kernel-models. The number of a hypercube model is 2 to the order of n, and n is the dimension of the hypercube. Each child has its cell-position. Both internal and external couplings have to be specified explicitly, with the external coupling having two choices, broadcast or origin-only, which specify different coupling strategies.

Controlled-models: subclass of kernel-models, enabling the modeler to impose centralized control over a class of components in a dynamic fashion.

Simulators: subclass of processors. Simulation in DEVS is done by message passing. In order to perform simulations, simulators are attached to the atomic models. The simulators record the time-of-last-event and determine the time-of-next-event. They receive messages, process them by computing the transition functions of the associated atomic models, and then response by sending out messages.

Co-ordinators: subclass of processors. Co-ordinators are assigned to the coupled models. When a co-ordinator receives a message, it transmits the message to the processors of its associated coupled model's receivers or to one of its children, or to its parent, depending on the type of the message.

Root-co-ordinators: subclass of processors. A root-co-ordinator manages the overall simulation and is linked to the co-ordinator of the outermost coupled model.

More details about the definitions given in this Appendix can be found in Zeigler (1986, 1987, 1989-a).

APPENDIX 2
METHODS FOR CLASS CONTINUOUS-MODELS AND CLASS CONTINUOUS-SYSTEMS

A) Methods for Continuous-models:

1. **set-sv (vname vvalue):** sets the value of *vvalue* to one of the *ind-vars* named *vname* (inherited from atomic-models).
2. **get-sv (vname):** gets the value of one of the *ind-vars* named *vname* (inherited from atomic-models).
3. **set-type (tf tn):** assigns the value of the instance variable $tflag$ as tf and *tname* as *tn*. The value of tf can only be true or false, otherwise an error message is generated. If the model is neither a type model, nor of a certain type, *tn* should be assigned to nil.
4. **valid? (tf tn):** checks whether this model is valid or not. If the model does not belong to any type, its counterpart model should exist in DYMOBASE. Or, if the model is of a certain type, its type model should exist in DYMOBASE. Otherwise an error message will be displayed.
5. **make-new (mname):** makes a copy of the original model with a name of *mname*.
6. **change-parameter (p-list):** Changes the value of parameters. *p-list* is a list of pairs, each pair containing a parameter-name and the value the parameter is to assume. It is sufficient to use *change-parameter* for those parameters only the values of which need to be altered.

7. **change-ic (ic-list):** *ic-list* is a list of pairs, each pair containing a variable-name and a value of the initial condition to be set. *change-ic* changes the initial condition of variables that are declared as *local, input, output, terminal,* or that are associated with *cuts*.

B) Methods for Continuous-systems:

1. **make-new (mname):** makes a copy of the original model with a name of *mname*.
2. **change-parameter (p-list):** Changes the value of parameters. *p-list* is a list of pairs, each pair containing a parameter-name and the value the parameter is to assume. It is sufficient to use *change-parameter* for those parameters only the values of which need to be altered.
3. **change-ic (ic-list):** *ic-list* is a list of pairs, each pair containing a variable-name and a value of the initial condition to be set. *change-ic* changes the initial condition of variables that are declared as *local, input, output, terminal,* or that are associated with *cuts*.
4. **set-xxx (list), and get-xxx:** *xxx* stands for *cut, mcut, path, mpath, node, parameter, local, terminal, input,* or *output*. The method *set-xxx* assigns a list to the field, while *get-xxx* reads out the current list from the field. The list consists of pairs of names and values that are associated with the field.
5. **build-composition-tree (m list-of-children):** establishes the *composition-tree* of a type model, showing that the children are component models of model *m*. It also sets up a type model *m*, built from its subcomponents, in DYMOBASE.
6. **build-system-tree (m list-of-children):** establishes the *composition-tree* of a model showing that the children are component models of model *m*. It also

sets up a model, built from its subcomponents in DYMOBASE. The library of the system will be generated in DYMOBASE.

7. **set-inf-dig (list-of-influencees):** establishes the *influence-digraph.* This method is inherited from digraph-models.

8. **set-int-coup (ch1 ch2 list-of-port-pairs):** couples the ports of $ch1$ with the ports of $ch2$, where $ch1$ and $ch2$ are children of the coupled model. Corresponding ports are specified in the *list-of-port-pairs.* In addition, the connection relations between submodels in the DYMOLA model are set.

9. **set-ext-inp-coup (child list-of-port-pairs list-of-variable-pairs):** couples the input ports of the model with the input ports of its *child.* In addition, the relations between the ports will be converted into the DYMOLA model. Relations between variables associated with these ports will also be set up in the DYMOLA model if the *list-of-variable-pairs* is not nil.

10. **set-ext-out-coup (child list-of-port-pairs list-of-variable-pairs):** couples the output ports of a *child* with the output ports of the parent model. In addition, the relations between the ports will be converted into the DYMOLA model. Relations between variables associated with these ports will also be set up in the DYMOLA model if the $list-of-variable-pairs$ is not nil.

11. **valid? (tf tn):** If the user wants to use a coupled (type) model in mbase, *valid?* will check wether the coupled DYMOLA (type) model is available or not. If not, an error message is displayed.

12. **set-type-model:** starts setting up a coupled type model in DYMOLA.

13. **set-system:** starts setting up a coupled model in DYMOLA.

14. **set-tylist (children):** searches among the *children* for the type models needed for building a coupled model in DYMOLA.

15. set-lib (children): sets up the model library in DYMOLA for a model to be simulated.

16. set-subcomponent(children): sets up submodel statements for the DYMOLA model.

17. write-connection (ch1 ch2 list-of-port-pairs): sets up the connection mechanism in the DYMOLA model.

18. write-ext-inp (child list-of-port-pairs list-of-variable-pairs): writes the external coupling relations/equations for the DYMOLA model.

19. write-ext-out (child list-of-port-pairs list-of-variable-pairs): writes the external coupling relations/equations for the DYMOLA model.

20. write-statement: closes up a DYMOLA model.

APPENDIX 3

FUNCTIONS AND MACROS IN DEVS-SCHEME FOR MANAGING CONTINUOUS MODELS

A) Functions:

1. **(compare-parameter ma mb):** compares two models *ma* and *mb* of the same type for common parameters. Common parameters with different values will be put into the result list.
2. **(compare-local ma mb):** compares two models *ma* and *mb* of the same type for common local variables. Common variables with different values will be put into the result list.
3. **(write-par plist one fv):** writes the parameter list for the submodel statement into the DYMOLA model, where *plist* is a list of parameter pairs (name value); *one* indicates if there is just one element in plist; *fv* is the name of the result file.
4. **(write-ic iclist fv):** writes the initial condition list for the submodel statement into the DYMOLA model, where *iclist* is a list of state variable pairs (name value); *fv* is the name of the result file.
5. **(write-subm tname list-of-models fv):** writes DYMOLA submodel statements, where *tname* is a type model name; *list-of-models* is a list of models of type *tname*; *fv* is the name of the result file.
6. **(write-declare mn):** writes the declaration part from the model *mn* to its DYMOLA model, where *mn* is the model name.
7. **(to-devs):** returns the control to DOS in directory DEVS.

8. (to-enbase): returns the control to DOS in directory ENBASE.

9. (to-mbase): returns the control DOS in directory MBASE.

10. (to-dymobase): returns the control to DOS in directory DYMOBASE.

11. (to-debase): returns the control to DOS in directory DEBASE.

12. (to-traject): returns the control to DOS in directory TRAJECT.

13. (to-dev): changes the current directory to DEVS.

14. (to-en): changes the current directory to ENBASE.

15. (to-m): changes the current directory to MBASE.

16. (to-dymo): changes the current directory to DYMOBASE.

17. (to-des): changes the current directory to DEBASE.

18. (dym-to-des stname): transfers a file from DYMOBASE to DEBASE.

19. (des-to-tr-ent stname): transfers a file from DEBASE to TRAJECT.

20. (create-dymola-batch pp): creates a DYMOLA batch file for automatically running DYMOLA on system *pp* (generated from the pruned SES) in DEVS.

21. (save-trajectory): saves trajectories into TRAJECT.

22. (dymola): runs DYMOLA.

23. (dymola stname): runs DYMOLA in batch on system *stname*.

24. (desire): runs DESIRE.

25. (run stname): executes a DESIRE simulation for a pruned entity structure system stname.

B) Macros:

1. (write-state mname sname): writes the variables from the continuous-systems (DEVS) model to its DYMOLA counterpart, where *mname* is the name of a model; *sname* is the name of the model's state variable.

C) Other Library Functions:

1. **(file-clear fn):** clears a file.
2. **(file-append fresult fin):** appends the file with the file variable fin to the file with the file variable fresult.
3. **(member? a ls):** checks wether a is in the list ls.
4. **(atomize ls):** flattens out the hierarchical list ls.
5. **(purge ls):** purges a list so that elements which appear several times in the list are eliminated except for one single occurrence.
6. **(differ ls1 ls2):** returns a list of the different elements of the two lists ls1 and ls2.
7. **(string->number string):** converts a character string representing a number with or without expononential form into this number.

APPENDIX 4

MACROS TO MANIPULATE SYSTEM ENTITY STRUCTURES

1. **(get-p-time ename mname ic par ith-test tc):** returns the simulation time from running a continuous simulation of a pure entity structure. It first transforms the pruned entity structure, generating a DYMOLA model and the corresponding DYMOLA command file, then runs DYMOLA in batch mode and produces a DESIRE program. Finally, it executes the DESIRE program. The simulation results are saved in the TRAJECT base. It also generates a state variable *p-time* for model mname and saves the simulation time in this state variable,

where

ename: name of the entity structure;

mname: name of a model;

ic: list of pairs of initial conditions. Each pair includes the name of a variable and its initial value;

par: list of pairs of parameters or list of parameter names. If pairs are specified, each pair includes the name of a parameter and its value;

ith-test: a number, specifies a particular simulation condition;

tc: termination condition for the simulation, or list of simulation termination conditions if more than one. If the termination condition has already been specified in the simulation control model then *tc* is nil.

2. **(get-p-list ename mname ic par ith-test tc):** returns a list of parameters and the simulation time from running a continuous simulation of a pure

entity structure. It first transforms the pruned entity structure, generating a DYMOLA model and the corresponding DYMOLA command file, then runs DYMOLA in batch mode and produces a DESIRE program. Finally, it executes the DESIRE program. The simulation results are saved in the TRAJECT base. It also generates a state variable *p-list* for model *mname* and saves the simulation time in this state variable,

where

ename: name of the entity structure;

mname: name of a model;

ic: list of pairs of initial conditions. Each pair includes the name of a variable and its initial value;

par: list of pairs of parameters or list of parameter names. If pairs are specified, each pair includes the name of a parameter and its value;

ith-test: a number, specifies a particular simulation condition;

tc: termination condition for the simulation, or list of simulation termination conditions if more than one. If the termination condition has already been specified in the simulation control model then *ic* is nil.

3. **(get-p-table ename mname ic par ith-test times tc):** returns a table of lists of parameters and their simulation times from running several continuous simulations of a pure entity structure. This macro helps to study the process of running simulations with different parameters. The values of the parameters are generated in the simulation control model. They can be values taken from an array, or generated from functions. It first transforms the pruned entity structure, generating a DYMOLA model and the corresponding DYMOLA command file, then runs DYMOLA in batch mode and produces a DESIRE program. Finally, it executes the DESIRE program.

The simulation results are saved in the TRAJECT base. It also generates a state variable *p-table* for model *mname* and saves the simulation time in this state variable,

where

ename: name of the entity structure;

mname: name of a model;

ic: list of pairs of initial conditions. Each pair includes the name of a variable and its initial value;

par: a list of parameter names;

ith-test: a number, specifies a particular simulation condition;

times: a number, specifies how many simulation runs are required;

tc: termination condition for the simulation, or list of simulation termination conditions if more than one. If the termination condition has already been specified in the simulation control model then *tc* is nil.

4. **(get-p-window ename mname ic par ith-test times tc):** returns the time window, i.e., the minimum simulation time and the maximum simulation time from running several continuous simulations of a pure entity structure. Besides the returned value, *get-p-window* functions the same way as *get-p-table* does. Its parameters are also the same as in *get-p-table*.

APPENDIX 5
TIME-WINDOWS CONCEPT IN EVENT-BASED INTELLIGENT CONTROL *

Event-based control is a discrete eventistic form of control logic, in that the controller expects to receive conforming sensory responses to its control commands within definite time-windows determined by its DEVS model of the system under control. With this control paradigm, the classical control process can be readily interfaced with rule-based symbolic reasoning systems in advanced robotic and intelligent automation.

Continuous models can be advantageously mapped into DEVS representations. Such DEVS models provide a basis for the event-based control. Figure A5.1 shows the concept of event-based control, and Figure A5.2 presents the DEVS description of event-based control.

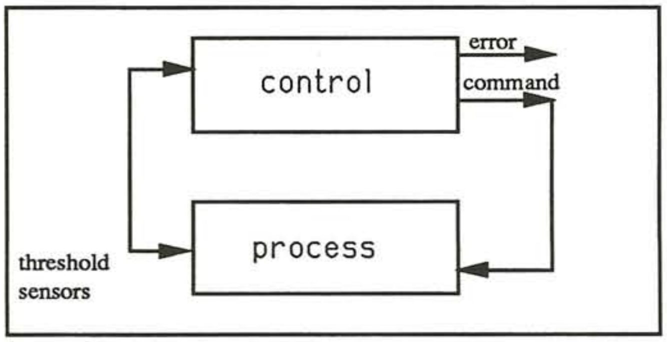


Fig. A5.1 Event-based control

* Most of the information given in this Appendix was extracted from Zeigler (1989-a).

```
initial phase: WAIT
initial sigma: tmin(P1)
initial checkstate: P1

------ external transition ------

when receive value on sensor-port
  case of: phase
    WAIT: hold-in EARLY 0
    WINDOW: if value = expected(Pi)
            then hold-in SEND-COMMAND 0
            else hold-in ERROR 0

------ internal transition ------

case of: phase
  WAIT: hold-in WINDOW window(Pi)
  WINDOW: hold-in LATE 0
  SNED-COMMAND: set checkstate = next(checkstate)
                hold-in WAIT tmin(next(Pi))
  ERROR: passive

------ output function ------

case of: phase
  EARLY: send "(Pi) input arrived too early" to error-port
  SEND-COMMAND: sends control command(Pi) to command port
  ERROR: send "(Pi) error in sensor value" to error-port
  LATE: send "(Pi) input arrived too late" to error-port
  else: send null message
```

Fig. A5.2 DEVS description of event-based control

It shows that the model starts in some *checkstate* $P1$ with *sigma* set to $tmin(P1)$. This means that it will stay in phase WAIT for a duration $tmin(P1)$. If a sensory input is received during this period, the external transition function recognizes this as an error, since it is too early for the expected sensory response. Once $tmin(P1)$ has elapsed without external interruption, the internal transition causes the model to change to phase WINDOW. The model is scheduled to stay in this phase for a duration given by $window(P1)$. If a sensory input is received during this period, the external transition function tests it for validity. If the test succeeds, an appropriate control command is issued from a transient phase SEND-COMMAND, *checkstate* is updated to $P2$, the WAIT phase is entered, and the model is scheduled to remain there for the appropriate duration, $tmin(P2)$. If the test fails, an error is reported. Finally, the internal transition function causes an error transition if the period, $window(P1)$, has elapsed without receipt of the expected sensory input (any subsequent input would arrive too late). It is seen that the model of event-based control moves through its *checkstate*s in concert with the received input, as long as that input arrives in the expected time-windows.

Time-windows can be determined from series of continuous simulation runs caused by the parameter changes of a process model under normal conditions. It is the time duration from the minimum allowed simulation time to the maximum allowed simulation time of a process. A macro in DEVS can help to determine a time-window directly from simulation runs (Section 5.5.5). Time-windows can also be derived by the DEVS models of the process. Let the DEVS model state $(q,\ x)$ represent a process state q which resides on a boundary, and an input x which the controller wishes to exert to drive the state to a second boundary. Then the value $ta(q,\ x)$ returned by the DEVS time-advance function is the time required to reach the desired boundary from state q under input regime x. Since the controller knows the process state only up to its being on the given boundary (i.e., only from the sensory outputs), the time to wait for a sensory response can only be narrowed down to lie between the smallest $ta(q,\ x)$ and the largest $ta(q,\ x)$ for states q on the boundary. Thus the window given by the DEVS models is the interval:

$$[\min\{ta(q,\ x)\}, \max\{ta(q,\ x)\}],$$

where the *min* and *max* operators are taken over states q on the boundary in question.

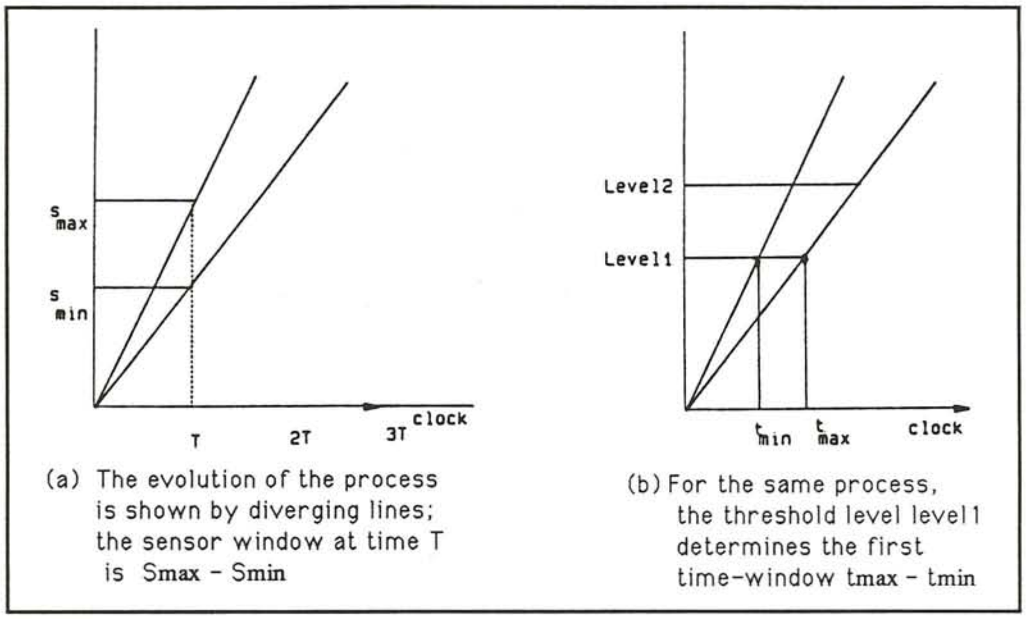


Fig. A5.3 Comparison of conventional and event-based control

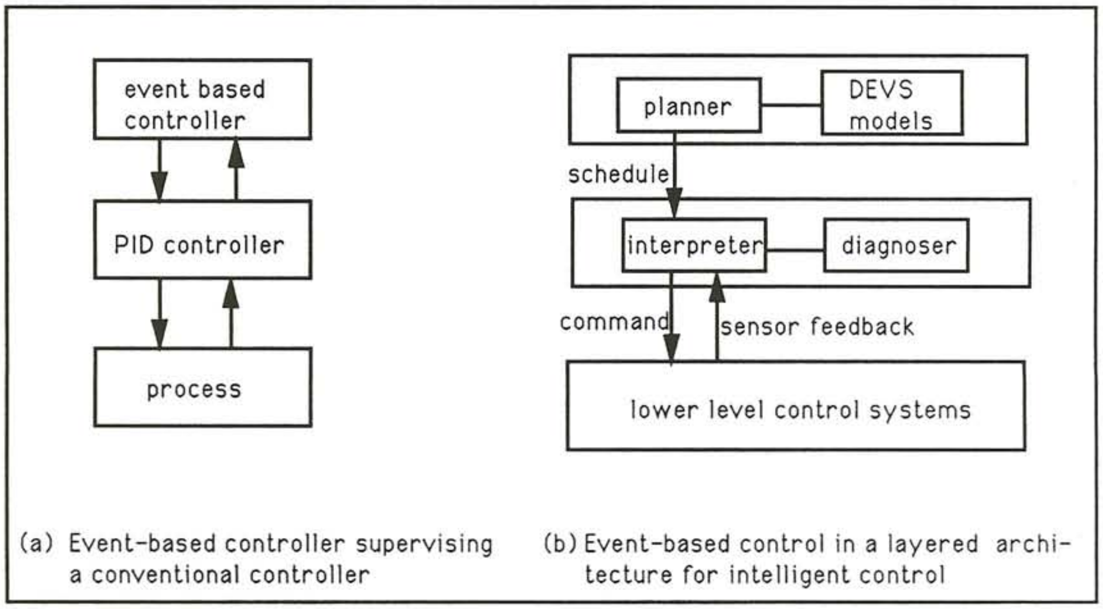

event based controller
PID controller
process
(a) Event-based controller supervising a conventional controller
planner
DEVS models
schedule
interpreter
diagnoser
command
sensor feedback
lower level control systems
(b) Event-based control in a layered architecture for intelligent control

Fig. A5.4 The role of event-based concepts in intelligent control

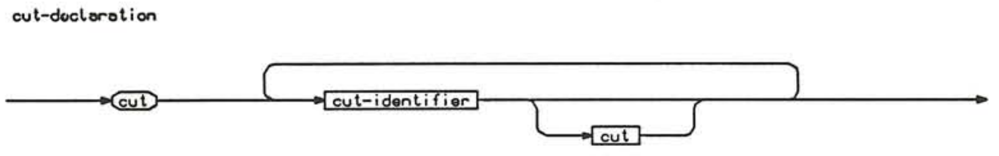
Figure A5.3 shows the difference between conventional control and the event-based control strategy. Figures A5.4 shows the role of event-based concepts in intelligent control.

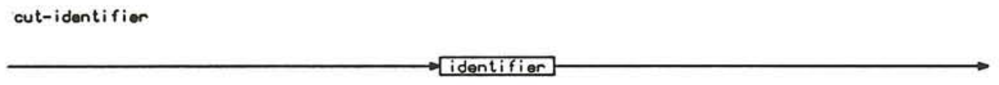
APPENDIX 6

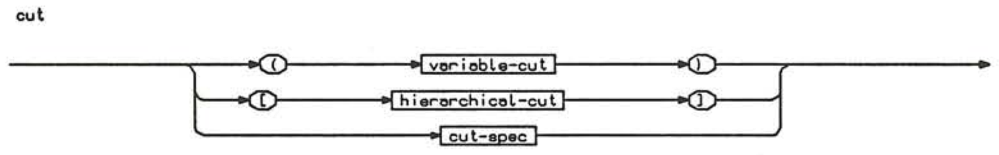
SYNTAX DIAGRAMS

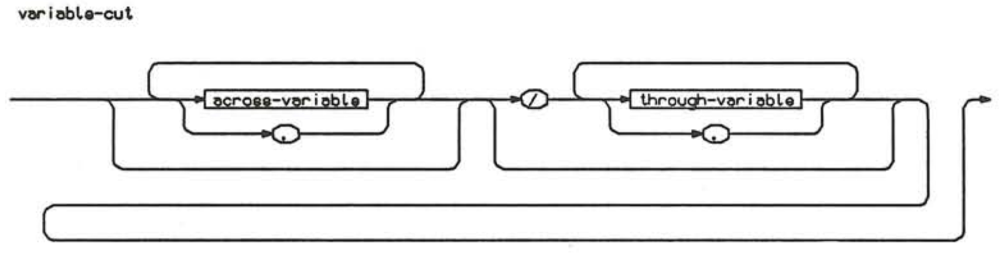
OF DYMOLA DESCRIPTIONS IN CHAPTER 3

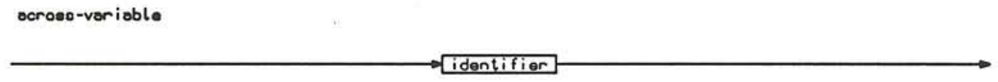
CUT DECLARATION



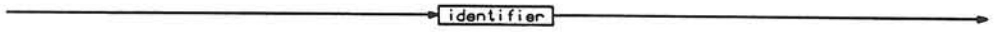




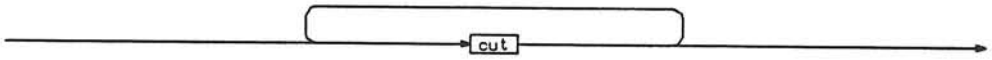




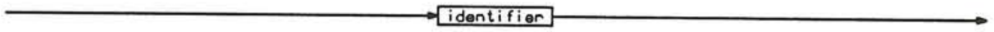
through-variable



hierarchical-cut

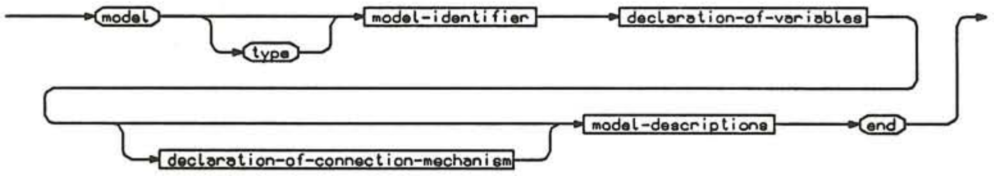


cut-spec

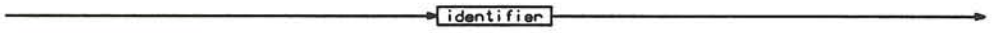


ATOMIC MODEL and COUPLED MODEL

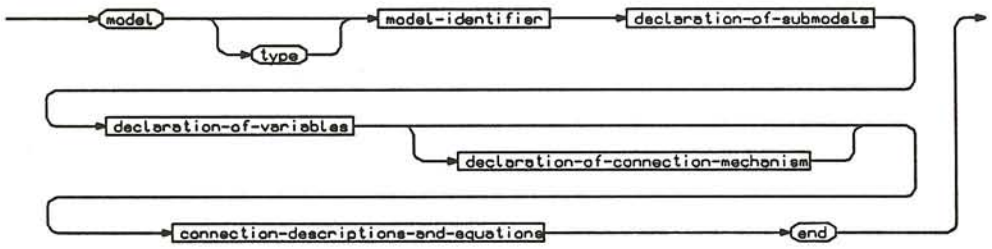
atomic-model



model-identifier

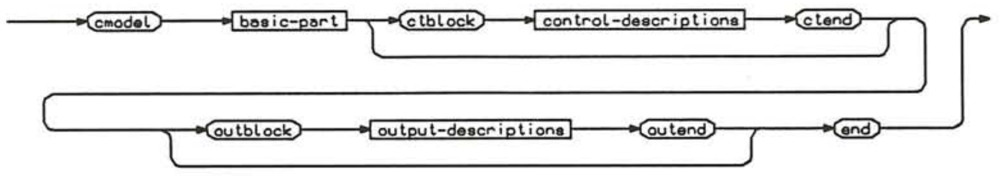


coupled-model



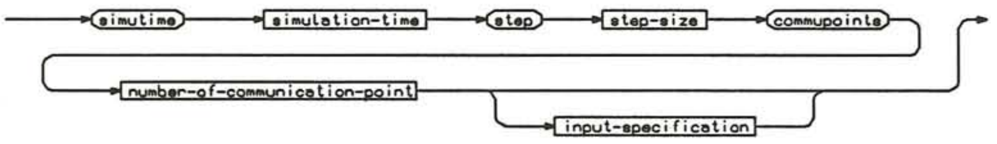
CONTROL MODEL

control-model

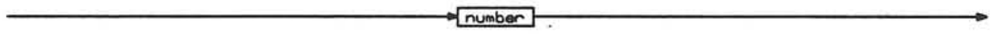


BASIC PART of CONTROL MODEL

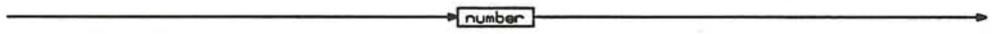
basic-part



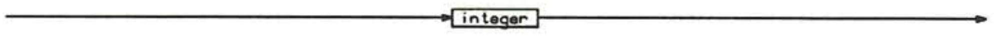
simulation-time



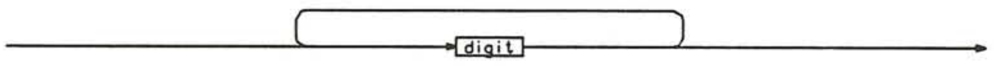
step-size



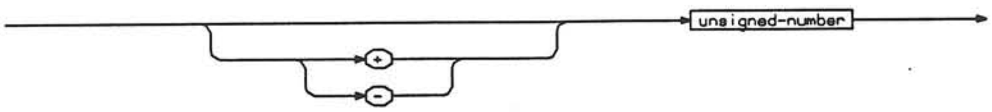
number-of-communication-point



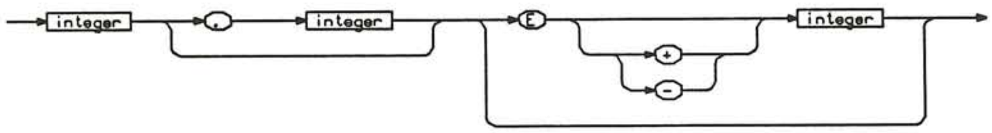
integer



number

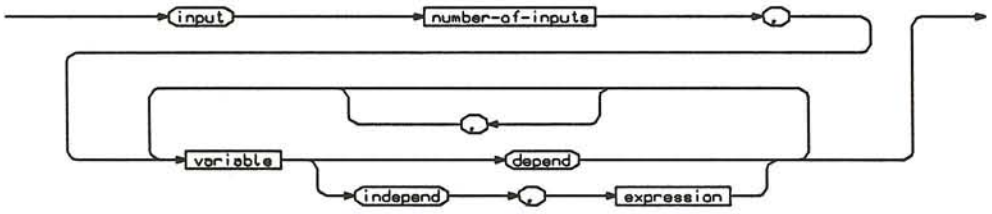


unsigned-number

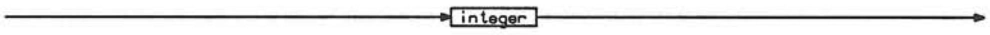


INPUT SPECIFICATION

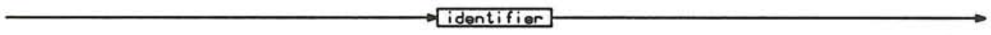
input-specification



number-of-inputs

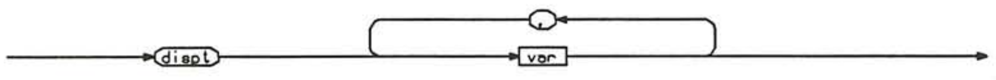


variable

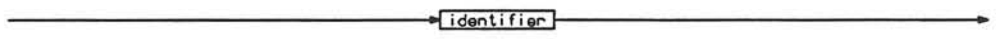


DISPT,DISPXY,TYPE,STASH STATEMENTS

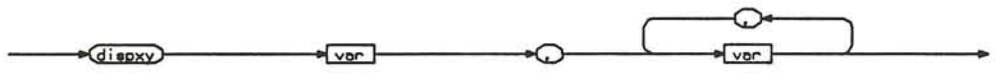
dispt-statement



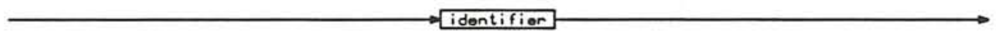
var



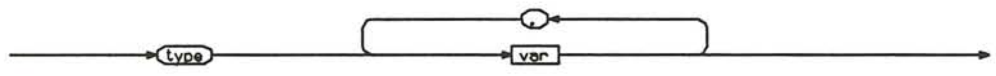
dispxy-statement



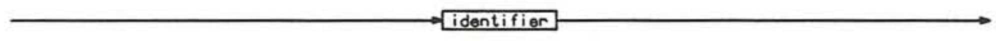
var



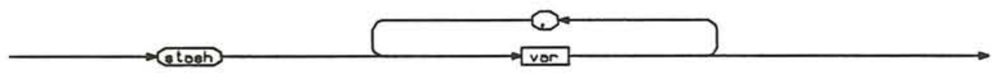
type-statement



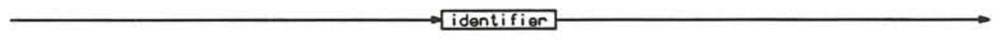
var



stash-statement

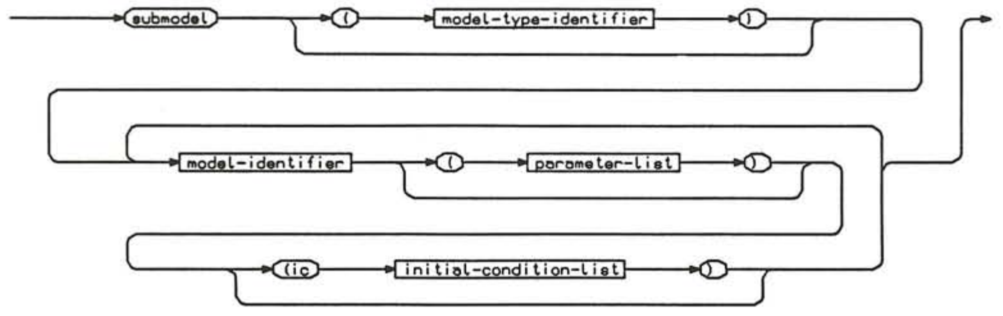


var

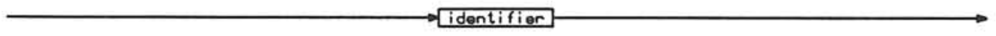


SUBMODEL DECLARATION

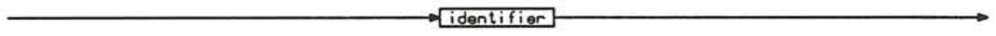
submodel-declaration



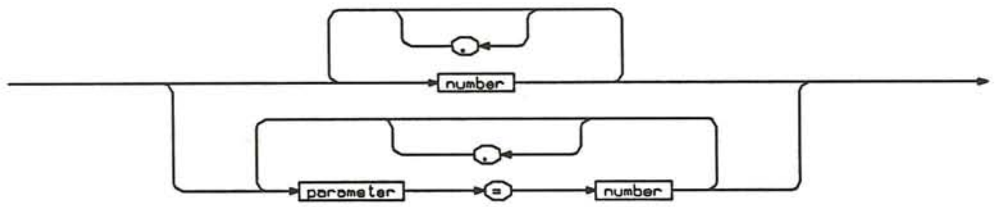
model-type-identifier



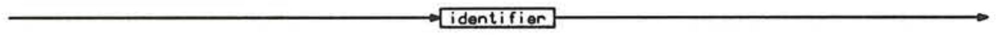
model-identifier



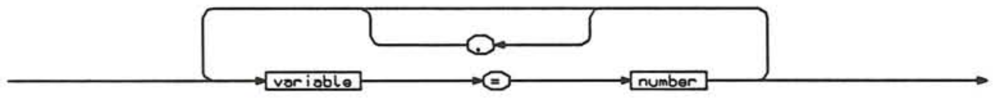
parameter-list



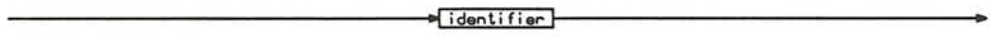
parameter



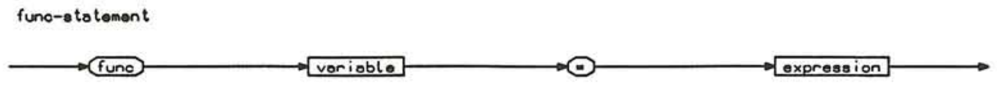
initial-condition-list

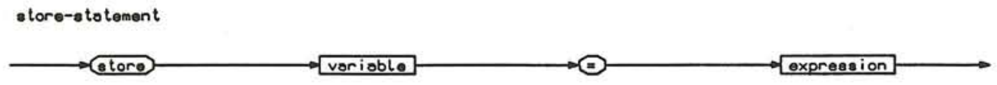


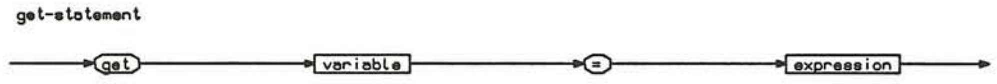
variable



FUNC, STORE, GET STATEMENTS







REFERENCES

Augustin, D. C. *et al.* (1967) "The SCi Continuous System Simulation Language (CSSL)," *Simulation*, 9, pp. 281-303.

Boeing Computer Services (1988), *Easy5/W - User's Manual,* Engineering Technology Applications (ETA) Division, P.O.Box 24346, MS 7L-23, Seattle, WA, 98124.

Bongulielmi, A. P. and Cellier, F. E. (1984) "On the Usefulness of Deterministic Grammars for Simulation Languages," *Simuletter*, **15**(1), pp. 14-36. Paper presented during the *Sorrento Workshop for International Standardization of Simulation Language (SWISSL)*, Sorrento, Italy, September 20-21, 1979.

Booch, G. (1986) "Object-Oriented Development," *IEEE Trans. Software Engr.*, vol. SE12, No. 2, pp.211-221, Feb. 1986.

Cellier, F. E. (1976) "GASP-V: A Universal Simulation Package," *Simulation of Systems*, L. Dekker, editor, North-Holland Publishing Company.

Cellier, F. E. (1979) *Combined Continuous/Discrete System Simulation by Use of Digital Computers : Techniques and Tools* PhD Dissertation, Swiss Federal Institute of Technology, Zurich/Switzerland.

Cellier, F. E. (1983) "Simulation Software: Today and Tomorrow," *Proc. Simulation in Engineering Sciences*, pp. 3-19, North-Holland.

Cellier, F.E. (1985) "Simulation Modelling Formalism : Ordinary Differential Equations", *Encyclopedia Systems and Control*, pp. 4356-4360.

Cellier, F. E. (1986) "Combined Continuous/Discrete System Simulation: Applications, Techniques, and Tools," *Proceeding of Winter Simulation Conference*, edited by J.R. Wilson *et al.*, pp. 24-33.

Cellier, F. E. (1988) "Continuous System Modelling and Simulation" *ECE 472 class notes*, Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz.

Christensen, Eric R. (1989) *DEVS-Sheme User's Manual,* Dept. of ECE, University of Arizona, Tucson, Ariz. 85721.

Close, Charles M. and Frederick, Dean K. (1978) *Modeling and Analysis of Dynamic Systems,* Houghton Mifflin Company, Boston

Elzas, M. S., Ören, T. R. and Zeigler, B. P. (1986) *Modeling and Simulation Methodology in the Artificial Intelligence Era*, North-Holland.

Elmqvist, Hilding (1978) *A Structured Model Language for Large Continuous Systems,* Ph.D. Dissertation, Report CODEN: LUTFD2/(TFRT-1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, Hilding (1979) "DYMOLA - A Structured Model Language for Large Continuous Systems," *Summer Computer Simulation Conference*, Toronto, Canada, July.

Elmqvist, Hilding (1981) "DYMOLA - A Structured Model Language for Large Continuous Systems - User's Manual," in Crosbie, R. E. and Cellier, F. E. (eds.), TC3-IMACS, *Simulation Software, Committee Newsletter No. 10.* (September)

Elmqvist, Hilding (1982) "A Graphical Approach for Documentation and Implementation of Control Systems," *Proc. of 3rd IFAC/IFIP Symposium on Software for Computer Control*, Madrid, Spain, October 5-8.

Expert-EASE Systems,Inc. (1988) *EASE+-User's Manual,* 1301 Shoreway Rd., Belmont, CA 94002

Franks, Roger G.E. (1972) *Modelling and Simulation in Chemical Engineering,* John Wiley and Sons, Inc., New York.

Huang, Yueh-Min (1987) *Building an Expert System Shell for Design Model Synthesis in Logic Programming,* MS Thesis. Department of Electrical and Computer Engineering, University of Arizona, Tucson, Arizona.

Integrated Systems, Inc. (1985), *System Build User's Guide,* 2500 Mission College Blvd., Santa Clara, CA 95054.

Kim, Tag Gon (1988) *A Knowledge-Based Environment for Hierarchical Modelling and Simulation,* Ph.D Dissertation. Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ.

Kim, Tag Gon, Zhang, Guoqing and Zeigler, B. P. (1988) "Entity Structure Management of Continuous Simulation Models," In *Proc. of 1988 Summer Simulation Conf.*, Seattle, WA.

Korn, Granino A. (1989-a) *NEUNET/DESIRE User's Manual,* 6801 E. Opata Rd., Tucson, AZ 85715.

Korn, Granino A. (1989-b) *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

MicroSim Corp. (1987), *PSPICE User's Manual,* 20 Fairbanks Rd., Irvine, CA 92718.

Moser, Thomas L. and Erickson, J. D. (1988) "The Evolution of Automation and Robotics in Manned Space Flight," in *Space Technology* vol. 8, No. 3, pp. 249-260, 1988, printed in Great Britain.

Ören, T. I. (1984) "GEST : A Modelling and Simulation Language Based on System Theoretic Concepts," In *Simulation and Model-Based Methodologies: An Interactive View* (Ören, T.I., Zeigler, B.P., and Elzas, M.S. eds.) North Holland, Amsterdam, The Netherlands, pp. 3-40.

Roffel, B. and Rijnsdorp, J. E. (1982) *Process Dynamics, Control and Protection,* Ann Arbor Science Publishers, Ann Arbor, MI.

Rozenblit, J. W. (1985) *A Conceptual Basis for Model-Based System Design*, Ph.D. Dissertation. Dept. of Computer Science, Wayne State University, Detroit, MI.

Rozenblit, J. W. and Zeigler, B.P. (1985) "Concepts of Knowledge Based System Design Environments," *In Proc. 1985 Winter Simulation Conf.*, San Franciso, CA, pp.223-231.

Rozenblit, J. W. and Huang, Y. (1987) "Constraint-Driven Generation of Model Structures," *In Proc. of 1987 Winter Simulation Conf.*, Atlanta, GA, pp. 604-611.

Rozenblit, J. W., Sevinc Suleyman and Zeigler, B.P. (1986) "Knowledge- Based Design of LANs Using System Entity Structure," *Proceeding of Winter Simulation Conference*, edited by J.R. Wilson, *et al.*, pp. 858-865.

Sarjoughian, Hessam S. (1989) *Intelligent Agents and Hierarchical Constraint Driven Diagnostic Units for a Teleoperated Fluid Handling Laboratory*, MS Thesis, Department of ECE, University of Arizona, Tucson, AZ 85721.

Standridge, Charles R. (1986) "An Approach to Model Composition from Existing Modules," In *Modelling and Simulation Methodology in the Artificial Intelligence Era*, M.S. Elazs, T.I. Ören and B.P. Zeigler (eds.), North Holland, Amsterdam, The Netherlands, pp. 113-120.

System's Designer's plc (1986) *Sysmod User Manual,* Release 1.0, D05448/14/UM, Ferneberga House, Alexandra Rd., Farnborough, Hampshire GU14 6D Q, U.K.

Texas Instrument (1985) *TI Scheme Language Reference Manual,* Dallas, TX.

Viewlogic Systems, Inc.(1988) *Workview Reference Guide* (Release 3.0), *View Reference Guide* (Version 3.0), 313 Boston Post Rd. West, Marlboro, MA 01752.

Zeigler, B. P. (1984) *Multifacetted Modelling and Discrete Event Simulation,* Academic Press, London, U.K. and Orlando, FL.

Zeigler, B. P. (1985) *Theory of Modelling and Simulation*, John Wiley, New York, 1976; reissued by Krieger, Malabar, FL.

Zeigler, B. P. (1986) *DEVS-Scheme: a Lisp-Based Environment for Hierarchical, Modular Discrete Event Models,* Technical Report AIS-2. CERL Lab., Dept. of ECE, University of Arizona, Tucson, Ariz. 85721.

Zeigler, B. P. (1987) "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment", *Simulation,* vol, 50, no.5, pp. 219-230.

Zeigler, B. P. (1989-a) "DEVS Representation of Dynamical Systems: Event-Based Intelligent Control," *Proceedings of the IEEE,* Vol. 77, No.1, pp. 72-80, January.

Zeigler, B. P. (1989-b) "Distributed Discrete Event Simulation" *ECE 574 class notes*, Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz.

Zeigler, B. P. (1989-c) *Hierarchical, Modular DEDS: Models, Knowledge and Endomorphy in Object-Oriented Simulation,* In preparation, Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz.

Zeigler, B. P., Cellier, F. E. and Rozenblit, J. W. (1988) "Design of a Simulation Environment for Laboratory Management by Robot Organizations", *J. Intelligent and Robotic Systems*, vol.1, pp. 299-309.

Zeigler, B. P. and Zhang, G. (1988) "The System Entity Structure: Knowledge Representation for Simulation Modelling and Design", *Artificial Intelligence, Modelling and Simulation,* edited by L.E. Widman *et al.*, John Wiley and Sons, New York.

Zhang, Guoqing (1988) *Knowledge Bsaed Simulation System - an Application in Controlled Environment Simulation System* MS Thesis, Department of Electrical and Computer Engineering, University of Arizona, Tucson, Arizona.