OmSim and Omola Tutorial and User's Manual

Version 3

Andersson, Mats

ISSN 0280–5316
ISRN LUTFD2/TFRT--7504--SE

OmSim and Omola
Tutorial and User's Manual

Version 3

Mats Andersson

Department of Automatic Control
Lund Institute of Technology
April 1993

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

Document name
INTERNAL REPORT

Date of issue
April 1993

Document Number
ISRN LUTFD2/TFRT--7504--SE

Author(s)
Mats Andersson

Supervisor

Sponsoring organisation
NUTEK

Title and subtitle
OmSim and Omola Tutorial and User's Manual

Abstract

Omola is an object-oriented language for representation of continuous time and discrete event dynamical models. Differential and algebraic equations are used for representing continuous time behaviour. OmSim is an implemented environment for modelling and simulation based on Omola. OmSim contains a graphical editor for defining and displaying structured models. OmSim is introduced to the novice user through a set of prepared examples and exercises. The basic features of Omola are also presented.

Key words
Modelling, simulation, simulation languages

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
39

Recipient's notes

Security classification

Contents

1. Introduction

OmSim is an interactive environment for defining and simulating dynamical models based on the modelling language Omola. The current version of OmSim runs on Sun SparcStations under the X window system. It contains the following important tools.

The Omola parser is invoked to load Omola model definitions into the environment. The parser checks the syntax and report errors.

The class browser allows the user to view the contents of a loaded library and to select an object from it. The browser is the main window in the environment and it is used to invoke and open up other tools.

The model editor is a graphical editor used for displaying and defining Omola models.

The simulator compiles an Omola model and translates it into a suitable form. The simulator has a number of built-in numerical integration routines for simulating the compiled model. It also has a number of subtools to access and to display variables and parameters, to display simulation results, to debug models, etc.

The command language interpreter Omola Command Language (OCL) is language for writing command procedures for setting up and running simulation experiments in OmSim. When an OCL definition is loaded into OmSim it is immediately executed by the OCL interpreter.

This document is meant to be a hands-on guide on how to use OmSim and an introduction to Omola. Chapter 2 consists of an OmSim tutorial, describing how models can be defined and simulated by a set of practical examples. Chapter 3 is a short introduction to Omola, describing the main concepts of the language. The two parts can be studied independently of each other.

1.1 Further information

More information about OmSim and its specific tools is found in a set of manual pages accessible through the Unix man facility. The following manual pages exists: ACCESS, BROWSER, FUNCTIONS, MED, OCL, OMSIM, PLOT and SIMULATOR.

A general presentation of the design and the ideas behind Omola is found in [Andersson, 1990]. Overviews of Omola and OmSim and some modelling examples are found in [Mattsson and Andersson, 1993] and [Mattsson *et al.*, 1993]. A preliminary description of combined discrete events and continuous time models in Omola is found in [Andersson, 1992]. Extensive examples using Omola for modelling chemical processes are found in [Nilsson, 1989].

1.2 References

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

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

MATTSSON, S. E. and M. ANDERSSON (1993): “Omola — an object-oriented modeling language.” In JAMSHIDI and HERGET, Eds., *Recent Advances in Computer-Aided Control Systems Engineering*, volume 9 of *Studies in Automation and Control.* Elsevier Science Publishers.

MATTSSON, S. E., M. ANDERSSON, and K. J. ÅSTRÖM (1993): “Object-oriented modelling and simulation.” In LINKENS, Ed., *CAD for Control Systems.* Marcel Dekker, Inc.

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

1.3 Acknowledgement

Omola and OmSim are results from work in the Computer Aided Control Engineering (CACE) project. The author would especially like to thank the project manager Sven Erik Mattsson, and the fellow co-workers Dag Brück, Bernt Nilsson and Tomas Schönthal. Gratitude is also directed to Leif Andersson for helping with LaTeX problems. Finally the author would like to thank Prof. Karl-Johan Åström for initiating the project and for providing strong support and encouragement.

This work has been supported by NUTEK project number 92-003500P.

2. OmSim Tutorial

This tutorial gives a general introduction to the OmSim environment. It is based on some simple modelling and simulation exercises.

2.1 How to use this tutorial

The text in this tutorial is structured such that the presentation of OmSim and the practical examples are interrupted by instructions to the reader to issue commands in OmSim. Command instructions are printed in column wide boxes. It is possible to run the complete examples just by reading and following the boxed instructions.

The examples in this tutorial are based on a set of given files with some libraries and example models. You must copy these files into your local directory before you can start doing the exercise. The files are located in a subdirectory of the OmSim installation called `tutorial`. [1]

Notation

Commands to OmSim are selected from menus or issued by clicking on graphical buttons. In this tutorial special notations are used for denoting buttons, menus, and menu items. The notation is defined the following table.

Description	*Meaning*	*Example*
Round box	Button	(Cancel)
Bould box	Menu	**[File]**
Box	Menu item	[Quit]
Boxes with arrow	Menu selection	**[File]**→[Quit]
Boxes with arrows	Pull-right selection	**[Tools]**→[Show tree]→[Inheritance]

2.2 Starting and stopping OmSim

OmSim is started by the command `omsim` in Unix. After a few seconds you are requested to position a window by moving the mouse to a desired position and then clicking a mouse button. The window is titled "Omola Class Browser" and it is the main window of OmSim. It is used for selecting models and for starting other tools.

> Prepare a working directory and copy the tutorial model files to it. Issue the command `omsim`. Position the window on the screen.

The newly started OmSim contains a library of basic Omola definitions. The library is called Base and its contents are listed in the class browser. The classes in Base are fundamental to OmSim and should not be changed or redefined.

In addition to the class browser an OmSim log window appears in the bottom left corner of the screen. This window is used by OmSim to output log messages and error messages.

1 At the Department of Automatic Control's Spark Stations the tutorial is found in `/home/cace/lib/tutorial`.

The class browser has two pull-down menus: $\boxed{\text{File}}$ and $\boxed{\text{Tools}}$. $\boxed{\text{File}}$ has an option, $\boxed{\text{Quit}}$, to stop OmSim.

> Select $\boxed{\text{File}}\rightarrow\boxed{\text{Quit}}$ in the class browser.

2.3 Running a simple simulation

The purpose of this section is to demonstrate how a simple model can be loaded and simulated in OmSim.

The model that is used in this example is the van der Pol oscillator defined by the second order non-linear equation

$$\frac{d^2y}{dt^2} + a(y^2 - b)\frac{dy}{dt} + y = 0.$$

The equation can be solved for different initial conditions and different parameter values by simulating the following Omola model.

```
VanDerPol ISA Model WITH
  a, b ISA Parameter WITH default:=1.0; END;
  y TYPE REAL;

  y'' + a*(y*y-b)*y' + y = 0;
END;
```

Starting OmSim with an Omola model file

A *model file* is a text file with Omola definitions. It can be loaded into OmSim and it may define any number of models. A list of model files may be given as argument to the OmSim startup command. The files will be loaded in given order into OmSim at startup time.
The van der Pol model is stored in the model file '`tutorial.om`'. Get a copy of that file to your working directory.

> Start OmSim by issuing the Unix command: '`omsim tutorial.om`'.

More model files can be loaded into OmSim by selecting $\boxed{\text{File}}\rightarrow\boxed{\text{Load}}$ in the main window.

The main OmSim window contains a browser for viewing the contents of *libraries* and for selecting models and other classes for further treatment. A library is a collection of Omola class definitions. It is usually so that each model file defines a library. If a library name is selected in the left column of the browser, its contained classes are listed in the right column.

Two libraries are now present in the environment: Base and Tutorial.

> Select Tutorial by mouse clicking on the name in the left part of the class browser. Click on `VanDerPol` in the right part of the same window, so that it becomes high-lighted.

VanDerPol is now the currently selected class. Figure 2.1 shows current appearance of Omola Class Browser.

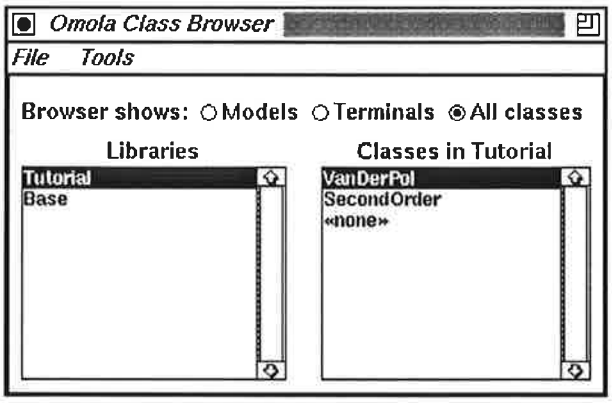


Figure 2.1 The Omola Class Browser with the Tutorial library.

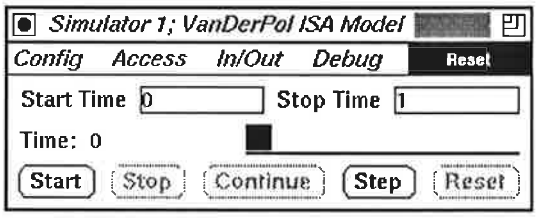

Simulator 1; VanDerPol ISA Model
Config Access In/Out Debug
Reset
Start Time 0 Stop Time 1
Time: 0
Start Stop Continue Step Reset

Figure 2.2 The simulator panel for the van der Pol example.

Creating a simulator

The OmSim Simulator is a tool that can be invoked from the Omola Class Browser's Tools menu. When a new simulator is created, it will be loaded with the currently selected model from the class browser. Several simulators can be created and each one will have its own control panel.

Select Tools→Simulate in the class browser and position the simulator control panel on the screen.

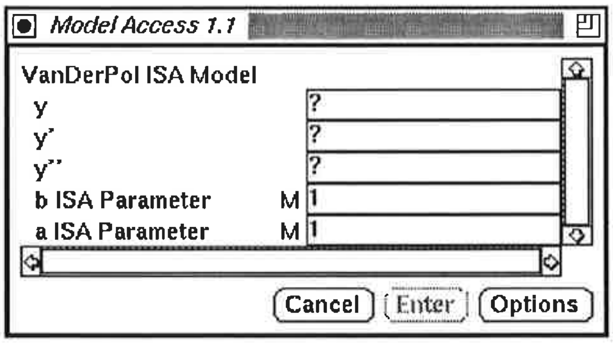
Creating a simulator means that the model is checked, equations are collected and analysed, and simulation code is generated. The simulator cannot perform any simulations if the selected model is found to be incorrect. This condition is shown by the status indicator in the upper right corner of the simulator showing "Error". If the model is correct the status "Reset" is shown.

The simulator panel has four pull-down menus for various kinds of operations and for opening of sub-tools such as plotters for displaying simulation results. The panel has two input fields where the user can specify simulation start and stop times. It also has a row of buttons used for the most common operations. Furthermore, it displays the progress of time during simulation and the current state of the simulation. The state of the current simulation activity is displayed in the upper right corner of the panel and it may show *Reset, Init, Active* or *Error*.

The simulator created for our example is shwon in Figure 2.2.

Creating an Access Tool

An *Access Tool* is a simulator sub-tool that can be opened from Access of the simulator panel. It is used for accessing the variables of the simulated model, for example, setting initial values or changing parameters. A few different



VanDerPol ISA Model

y		?
y'		?
y''		?
b ISA Parameter	M	1
a ISA Parameter	M	1

Figure 2.3 Model Access tool for the van der Pol example directly after creation.

kinds of access tools are available. The most generally useful access tool is obtained by selecting Access→Model Struct. It gives access to all variables of the simulation model.

Select Access→Model Struct in the simulator panel and position the Model Access window.

The access tool, shown in Figure 2.3, displays the three variables (y and it's first and second derviatives) and the two parameters of VanDerPol.

Each row in a model access tool shows either the name of a substructure (a component of the model), or the name and the current value of a variable. Clicking on a substructure name opens or closes the display of its contents. Pressing a mouse button on a variable name pops up a menu of operations that can be performed on the variable.

A variable display line sometimes also shows a letter or some other symbol indicating the source of the current value. For a freshly created simulator, values are marked with an 'M', indicating that the current values come from the model definition. Unknown values are indicated by question marks.

Setting up a plotter

Plotter is another important simulator tool. A new plotter is created from In/Out in the simulator panel. The plotter window has two pull-down menus: File and Config, and two buttons for common operations: Erase and Rescale.

Select In/Out→Plotter in the simulator panel and position the new plotter window on the screen. Don't let it obscure any of the other OmSim windows.

The access tool is used to select variables to plot. This is done by pushing a mouse button on a desired variable name in the access tool, moving to the Connect... item, and selecting an appropriate plotter from the pull-right submenu. Plotters and other tools are identified by names displayed in the window header.

We would like to display y and dy/dt in the plotter.

From the menu of the y variable in the access tool, select Connect→Plotter1.1. Do the same thing for y'.

Setting initial values

The access tool shall be used for entering initial values. When the simulator is in reset mode, the value boxes in the access tool show the initial value of each variable. The values can be changed by clicking and typing in the value boxes. When one or more values have been changed the new values have to be sent to the simulator by clicking (Enter).

> Mouse click in the value box of y in the access tool. Use the keyboard to enter the value of 1. Then click on (Enter).

Values that have been changed by the user will be marked with 'E' to indicate an *Explicit* user value. If some variables are still marked with question marks when the simulation is started, the simulator tries to derive consistent initial values from the model. If this is not possible zeros will be used as initial values.

Running the simulation

We are now ready to run the simulation.

> We want to simulate from zero to 10 seconds. Enter '10' in the Stop Time box of the simulator panel. Click on (Start). Click on (Rescale) on the plotter to make the whole graph visible.

The resulting graph is shown in Figure 2.4.

The simulator has an options panel where the user can change a number of different attributes and flags controlling the simulator and its numerical routines. For example, it is possible to change integration method, distance between output points, and error tolerances for the numerical routines.

It is always a good idea to check the simulation of a new model by adjusting the error tolerances and trying different integration methods, and comparing the results.

We will check if the simulation result is sensitive to local errors in the used integration routine.

> Select [Config]→[Options...]. Increase the error tolerance one order of magnitude compared with the default setting, by entering '`0.01`' for the Relative Error and '`1e-05`' for the Absolute Error. Confirm the change by pushing (Enter).
> Repeat the simulation by pushing (Start) again.

We will also check if the result is sensitive to the choice of integration method. The van der Pol model can be simulated with any of the available integration methods.

> Select [Config]→[Options...]. Change to some other method, for example to Dopri45r, by clicking on the appropriate radio button. Confirm the change by pushing (Enter).
> Repeat the simulation by pushing (Start) again.

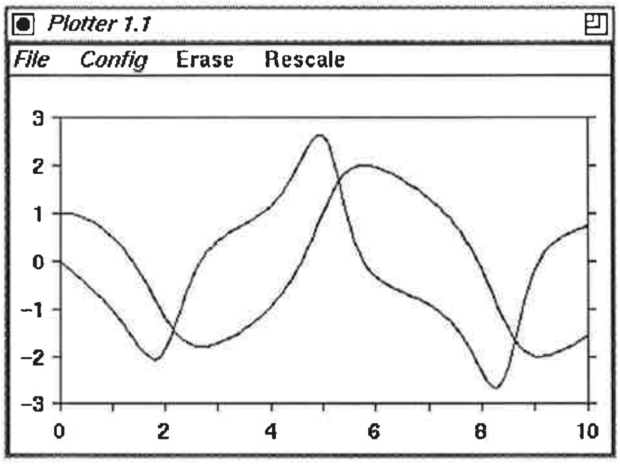


Figure 2.4 Graph obtained from the van der Pol simulation.

2.4 Working with structured models

In this section we will create a structured model based on library components. The example is based on two libraries with electrical components. However, we will start by introducing some basic Omola terminology. You may also turn to Chapter 3 for a more detailed description of Omola.

A structured model is a model whose behaviour is represented by a set of submodels. The submodels may be structured models themselves. A model that is not structured is primitive. It has no submodels and its behaviour is defined by equations only.

Structured and hierarchical modelling in Omola is based on *terminals* and *connections*. A terminal is a variable that is part of the model's interface. A connection defines a static relationship between terminals. A connection between terminals of different submodels means that the submodels are interacting.

Terminals (and their connections) can be simple or structured. A simple terminal involves a single quantity while a structured terminal contains multiple quantities.

Omola is an object-oriented modelling language. A model or a terminal in Omola is a *class definition*. Every class defines a general concept that can either be used as a basis for other classes or it can be instantiated into a simulation model. Every class in Omola has a *super class*. Attributes of the super class will be inherited by the new (derived) class.

Viewing Omola definitions in OmSim

Restart OmSim and load two model files by issuing the Unix command: '`omsim basicel.om elgen.om`'. This will load two electrical libraries called BasicElectrical and ElectricalGenerators.

OmSim has facilities for displaying models and libraries in different ways. Any Omola class definition can be presented as text. Inheritance and component hierarchies can be presented as tree diagrams. The model display facilities are found under $\boxed{\text{Tools}}\rightarrow\boxed{\text{Omola}}$ and $\boxed{\text{Tools}}\rightarrow\boxed{\text{Show tree}}$. There are three tree display possibilities: inheritance tree, component tree, and submodel tree. They all display a tree with the currently selected class as the root. Every node in a

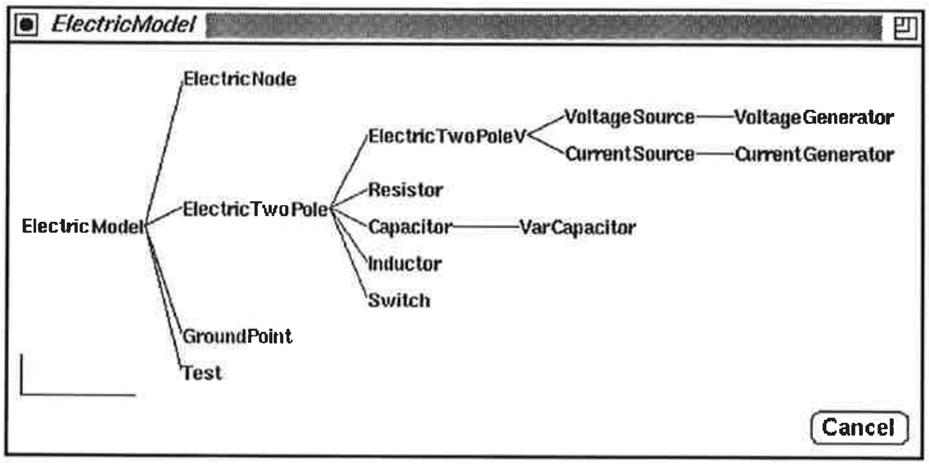


Figure 2.5 Inheritance tree for the electrical library.

tree diagram represent a class definition. The inheritance tree for the electric library is shown in Figure 2.5.

> Select library BasicElectrical and class ElectricModel. Select Tools→Show tree→Inheritance. Position the window and extend it to make it wider.

The nodes in a tree display are mouse sensitive. A mouse click on a node creates a window showing the Omola definition of that class. The same type of window for the currently selected class can be created by Tools→Omola. Names of other classes are printed in bold characters in Omola display windows. They are mouse sensitive and clicking results in a new window displaying that class.

> Click on ElectricTwoPole, in the inheritance diagram.

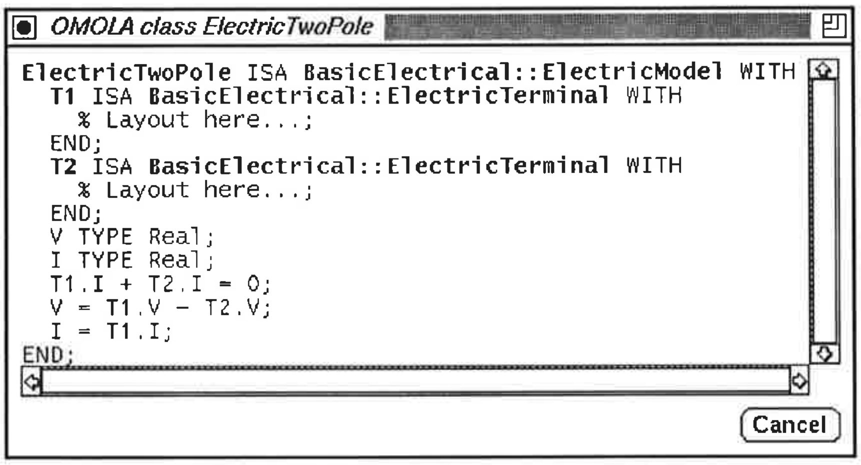
The result is the window shown in Figure 2.6.

The electric libraries

A set of libraries of electrical components and models have been developed to be used in this tutorial. We will start by giving a brief presentation of these libraries.

BasicElectrical is a library which defines a set of basic classes that are fundamental to our particular application domain. The library is printed in Appendix D. The definitions can be divided into three groups: terminals, base classes, and electric components.

The most important terminal in this library is called ElectricTerminal. It is a structured terminal with two components, V, which is a VoltageTerminal and I, which is a CurrentTerminal. An ElectricTerminal represents a single connection point of an electrical component, for example, one end of a resistor. It is has an electric potential, related to a system reference point, and an electric current flowing into the component. It is a convention used throughout this library that all terminals have a current with a positive value if the current is flowing into the component.



```
ElectricTwoPole ISA BasicElectrical::ElectricModel WITH
  T1 ISA BasicElectrical::ElectricTerminal WITH
    % Layout here...;
  END;
  T2 ISA BasicElectrical::ElectricTerminal WITH
    % Layout here...;
  END;
  V TYPE Real;
  I TYPE Real;
  T1.I + T2.I = 0;
  V = T1.V - T2.V;
  I = T1.I;
END;
```

Figure 2.6 Omola window showing definition of ElectricTwoPole.

Study the Omola definition of ElectricTerminal by selecting it in the Omola Class Browser and then choosing **Tools**→Omola. Position the window and enlarge it to an apropriate size.

An important base class is called ElectricTwoPole. It is used as a super class for many electric components that have two electric terminals. The terminals are called T1 and T2. A property common to all electric two-poles is that the currents through the terminals are equal but of different signs. This is expressed as an equation in ElectricTwoPole. ElectricTwoPole also defines two local variables, I and V, for convenience. V is the voltage across the two-pole and I is current floating from T1 to T2. The definition of ElectricTwoPole is shown in Figure 2.6.

Study the Omola definition of ElectricTwoPole in the same way as ElectricTerminal. Cancel the Omola Windows when you are done.

ElectricGenerators is a library which contains a set of voltage and current sources that are useful in models of electric circuits. CurrentSource and VoltageSource are constant sources where the voltage and current are defined by parameters.

Study the Omola definition of VoltageGenerator. How many equations does the VoltageGenerator define? How many variables and equations are defined for a VoltageGenerator when we also include inherited attributes?

CurrentGenerator and VoltageGenerator are controlled sources, where the current and voltage are controlled from an input terminal. The library also includes a two waveform generators that can be used to control the voltage and current generators.

The graphical model editor

We will now put together a simple structured model based on building blocks from the electric libraries. A model block diagram editor (Med) is an OmSim tool used for defining and viewing structured models graphically.

Select ≪**none**≫ in Class list of the Omola Class Browser. Open a model editor by selecting [Tools]→[Med] in the Omola Class Browser.

Med has one pulldown menu, called [Edit], and two editing buttons, called (Insert) and (Connect). It also has a graphical editing area showing the diagram of the edited model.

The Edit menu allows the user to load the editor with an existing model or with an empty, newly created, model. The model that is currently selected in the Omola Class Browser will be loaded for editing when Med is started or when [Edit]→[Existing] is selected. A new model can be created in Med, by selecting [Edit]→[New]. The currently selected model will then be used as the super class of the new model.

Select ElectricModel in the BasicElectrical library. Select [Edit]→[New] in Med.

A box with the name 'Unnamed' will appear in the model editor.

The model currently being edited in Med is called the *subject*. Every object visible in the editor, including the subject, has an associated pop-up menu. The menu is displayed by pushing a mouse button on the name or on an icon of the object. An option called [Info] in the object's menu, can be used to display important information about the object and to change its name.

We will start by giving the new model a proper name.

Press a mouse button on the string `Unnamed` and select [Info]. Use the keyboard to edit the Class name field. For example, change the name into `ElectricTest`. Confirm the change of name by pushing (Set name).

Submodels and terminals are added to the subject by the use of (Insert). An instance of the currently selected item in the Omola Class Browser is inserted as a new component of the subject every time (Insert) is pushed. The new component has to be placed on the diagram area by the mouse. New components are given automatically generated names. Positions and names of components can be changed at any time by the use of the component's pop-up menu.

It is a good practice to position terminals along the rectangle border and submodels inside the rectangle of the subject.

We will build a simple electric circuit based on a voltage generator, a resistor and a capacitor.

Select VoltageGenerator in library ElectricGenerators. Click on (Insert) in the Model block diagram editor. Move the mouse into the rectangle of the graphical area and position the icon by clicking a mouse button somewhere in the left part of the diagram.
Select FunctionGenerator from the same library and insert one in the diagram close to the voltage generator. Continue by inserting a Resistor and a Capacitor from library BasicElectrical. Finally, insert a GroundPoint, also defined in BasicElectrical.

(Connect) is used for defining connections between terminals of the subject and submodels. A connection is created in the following way.

1. Click on (Connect).
2. Select a terminal in the diagram.
3. Position any number of corners of the connection line.
4. Select the second terminal.

After the click on (Connect) until the second terminal has been selected, the editor is in connection mode. The mode is indicated by the mouse cursor appearing as a cross. Connection mode can be cancelled by pushing any key on the keyboard.

Selection of terminals can be done in different ways. One possibility is by clicking on a terminal icon. Terminal icons of submodels consists of small filled squares. Terminal icons of the subject model appear as larger, unfilled, squares with the names of the terminals. Another possibility to connect submodels is to select the terminal from the pop-up menu of the submodel icon. Next step will be to connect the submodels in our example model.

> Push (Connect). Then select the lower terminal of the voltage generator. Position a corners if desired. Finally select the terminal at the ground point icon. Repeat the procedure to connect the other end of the voltage generator to the resistor, the resistor to the capacitor, and finally the capacitor to the ground.

In order to connect the function generator to the voltage generator the terminals have to be selected from menus, since these particular terminals have no icons.

> Push (Connect). Push a mouse button on the function generator icon and select [x] from the pop-up menu. Then push a mouse button somewhere in the middle of the voltage generator icon and select [V0] from the menu.

The new model is now completed and the resulting diagram is shown in Figure 2.7.

Simulating the model A simulator for a model in Med can be created from the subject menu, that is the pop-up menu activated by pushing a mouse button on the subject's name in the upper left corner of the diagram.

> Push a mouse button on the name of the model you have created in Med and select [Simulate] from the pop-up menu. Position the simulator control panel.

Using the Model Access Tool A Model Access Tool is needed in order to get access to simulation variables. It is needed to specify variables to plot, to set initial values and to display variable values during simulation. The access tool obtained from [Access]→[Model Struct] of the simulator panel is useful to get access to all variables of a structured model. When it is created it shows the

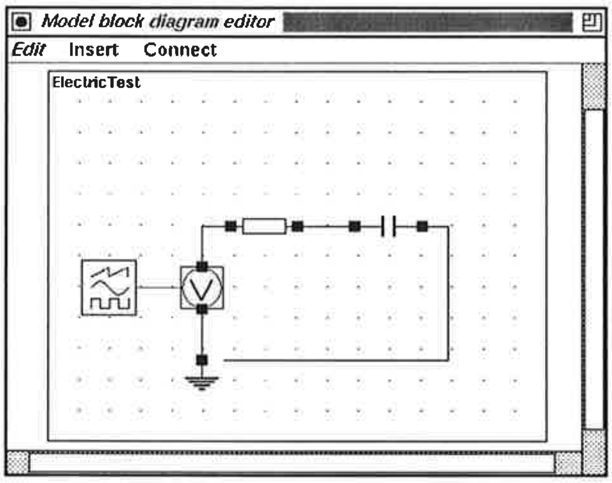


Figure 2.7 Med showing completed structured model.

names and the super classes of the model and all its immediate submodels. Each row displays an object which is either a component or simple variable. An object that is not a simple variable can be expanded in the display, so that its components become visible. A mouse click on a non-expanded component will make it expanded. A mouse click on an expanded component will compress it to its original one-line form.

Models that are defined by Med normally have automatically generated component names. For submodels with icons the names are normally not visible in the diagram. It is often necessary to know the name of a particular submodel to be able to select correct variables in an access tool. The name can be seen in the title of the pop-up menu of the submodel icon.

> Select [Access]→[Model struct] in the simulator panel. Position the access tool window and extend its size to make it about twice as high and a bit wider.
> Click on the line showing the capacitor.

The components of the capacitor submodel will be displayed. The parameter, C, and the the terminals, T1 and T2, can be expanded further.

> Create a plotter by selecting [In/Out]→[Plot] in the simulator window.

The capacitor has a variable, V, representing the voltage across the capacitor.

> Connect variable V of the capacitor to the plotter by pushing a mouse button on the variable name and selecting [Connect]→[Plotter1.1].

It is possible to make access tools which display selected variables. Access tools containing all parameters or all state variables of a model can be created from the menu. It is also possible to create customized access tools by copying single variables from other access tools. This is done by selecting [Copy] from the variable's pop-up menu.

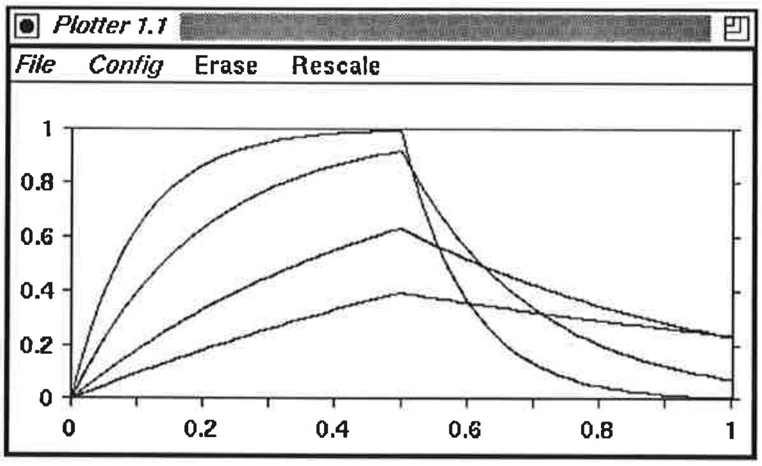


Figure 2.8 Graphs of ElectricTest simulation with square wave input and different capacitance settings.

In order to experiment with different parameter settings in our test model it is convenient have an access tool with only the parameters.

> Select Access→Parameters in the simulator panel.

Except for the resistance and the capacitance parameters, the function generator has a number of parameters for selecting waveform, frequency, amplitude, etc. The waveform is determined by a parameter, Func, with a symbolic value which may be `sawtooth`, `sine`, `square`, or `triangle`.

> Click in the value box of the Func parameter of the function generator model in the variable access tool for parameters. Use the keyboard to edit the value and replace the default value of `sine` with `square`.
> Simulate the model and try different settings for other parameters.

The resulting graphs from four simulations with square wave generator and different values of the capacitance is shown in Figure 2.8.

Saving the model A model created by Med can be saved in a file as Omola code. Submodels, connections and the graphical layout will be represented in Omola and saved.

A model can be associated with a file. To save a particular model it is necessary to save the file with which it is associated. This will also save all the other model associated with the same file. Menu selection File→Save file of the Omola Class Browser allows the user to select a model file and save it.

New models created by Med are always entered in a library called Scratch and associated with a file called '`scratch.om`'. The file name can be changed through the Info option of the model's pop-up menu in Med. If the file name is changed to an existing model file, the model will also be moved to the last library in that file. If the file name is changed to a non-existing file, then a new library with a similar name is created and the model is moved to that library.

Select Info from the pop-up menu of ElectricTest in Med. Change the file name to '`test.om`' and confirm the change by clicking in Set name. A new library called test appears in the class browser. Save the new model by selecting Save file in File of the class browser. Select `test.om` in the file selection box that appears and confirm the operation by clicking on Save.

2.5 Using an OCL script

The Omola Command Language (OCL) is a simple language for writing command scripts. A command script is a textual representation of a sequence of commands which are otherwise performed interactively using mouse and keyboard input. OmSim can read OCL scripts from a file and execute the commands. By using OCL it is possible to set up and execute complete simulation experiments. OCL is described in a separate manual.

We will use an OCL script to perform a parameter study of a simple second order model. The model is called `SecondOrder` and it is defined in the file '`tutorial.om`'.

The example OCL script first defines the model and calls it `m`. Then it creates a simulator called `sim` and a plotter called `Response`. Finally five simulations are performed for different settings of the model parameter `z`. The OCL script is defined in the file '`experiment.ocl`' and it looks like the following.

```
BEGIN
  Tutorial::SecondOrder m; % Create the model

  Simulator sim(m);         % Make a simulator for it
  sim.display(200, 200);    % Display the simulator
  sim.stoptime := 10;       % Set simulation stop time

  Plotter Response(m);      % Make a plotter
  Response.display(600, 200);
  Response.xrange(0, 10);   % Set the scales
  Response.yrange(0, 2);
  Response.y(x);            % Define the variable to plot

  % Simulate for different dampings:

  FOR z := [0.2, 0.3, 0.5, 0.7, 1.0] BEGIN
    m.z := z;               % Set damping parameter
    sim.start;              % Run simulation
  END;
END;
```

An OCL block starts with `BEGIN` and ends with `END;`. The commands are based on the object-oriented philosophy of creating objects and sending messages to them. Models in the model database and tools in OmSim are instantiated into objects and given names local to the particular block.

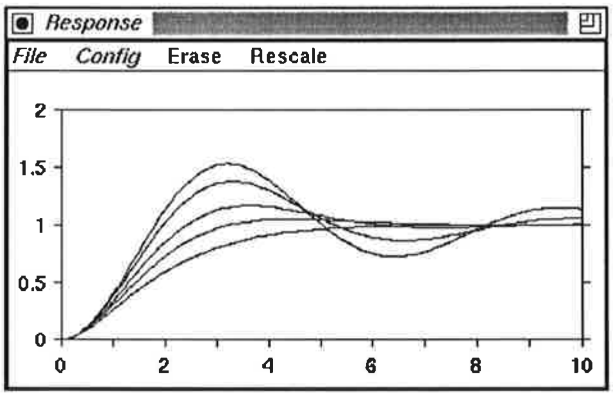


Figure 2.9 The resuling graphs from the example OCL script.

OCL scripts can be included in ordinary Omola model files and they are executed when the file is loaded. However, it is common practice to store each OCL script on a separate file.

We can try the OCL script by restarting OmSim and giving the script file as the last argument.

Restart OmSim by issuing the Unix command '`omsim tutorial.om experiment.ocl`'. Position the main OmSim window.

OmSim is started, the models are loaded and the OCL script is executed. The resulting plot is shown in Figure 2.9.

3. The Omola Modelling Language

This chapter gives an introduction to the Omola modelling language.

3.1 Class definitions

Omola models are based on class definitions. A class definition serves as a prototype that can be used for creating any number of model instances. Model instances are used during simulation.

Classes are used for representing model structures. A class defined on the top level in a file is called a *global* class. Classes can also be used as attributes, defined inside other classes. Such classes are called *components* or *local* classes.

A class has a name, a super class, and possibly a body of attribute definitions. A class definition has the structure:

```
<name> ISA <super class> WITH
  <class body>
END;
```

or if the class has no local attributes simply:

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

The class body can include definitions of other classes, variables, equations, connections and events.

As an example of a class definition, regard the following definition of a van der Pol model.

```
VanDerPol ISA Model WITH
  a, b ISA Parameter WITH default:=1.0; END;
  y TYPE REAL;

  y'' + a*(y*y-b)*y' + y = 0;
END;
```

The name of the defined class is '`VanDerPol`'. Its super class is `Model`, which is a previously defined class. The body defines four attributes: two component class definitions, one variable, and one equation.

3.2 Class attributes

A class definition may have a body of attribute definitions. Attributes can be

- components, which are nested class definitions representing submodels, terminals, parameters, etc.
- variables,
- equations,
- connections, and
- events.

Components, variables and equations are presented in this section. Connections are discussed in Section 3.6 and events are discussed in Section 3.7.

Components
A component is a class definition nested inside another class definition as an attribute.

A component is defined in the same way as a global class definition. A minor difference is that it is possible to give a list of names for defining a set of identical components. For example,

```
TankSystem ISA Model WITH
  Tank1, Tank2 ISA TankModel;
END;
```

defines a model, TankSystem, with two components, Tank1 and Tank2, which are subclasses of TankModel which must be previously defined.

Variables
The body of a class definition may include variable definitions. A variable has a name and a type and it may take different values during simulation of the model.

There are five basic data types supported in the current implementation. These are real, integer, string, symbol and enumeration. In addition, the structured type matrix of real, is supported. An enumeration typed variable takes symbolic values that are restricted to a given set of symbols.

A variable definition may also include a *binding* which binds the variable to an expression or a constant value. If a variable is bound to a constant value, this value will hold for all instantiations of the class at all times during simulation, i.e, the variable is actually a constant.

A variable binding must be type consistent with the variable. The types must be identical or it must be possible to coerce the binding to the type of the variable. For example, a real variable may be bound to an integer expression but not vice versa.

A string constant is written in double quotes while a symbolic constant is preceded by a single quote mark. Here is an example with variables of the six different types, all with bindings to constant values.

```
M1 ISA Class WITH
  x TYPE REAL := 2.0;
  i TYPE INTEGER := 1;
  s TYPE STRING := "This is a string.";
  symb TYPE Symbol := 'OmSim;
  enum TYPE (Gas, Water, Steam) := 'Gas;
  matr TYPE [2, 2] := [1.1, 1.2; 2.1; 2.2];
END;
```

A variable without a binding is called a *free* variable and may be used as a state variable in a dynamic model. The following example shows a class with three variable attributes: x, which is a constant; y, which is a free variable; and z, which is bound to an expression that depends on x and y.

```
M2 ISA Class WITH
  x TYPE Integer := 1;
  y TYPE Real;
  z TYPE Real := x + y;
END;
```

A variable definition may include the keyword 'DISCRETE' to declare that the variable is not varying continuously in time. Discrete variables are discussed together with events in Section 3.7.

Equations
The body of a class may define equation attributes. Equation attributes can either be causal or non-causal. A non-causal equation attribute, normally just called *equation*, is written as

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

where the left and right side expressions have equal status. A causal equation attribute, usually called *assignment*, has the form

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

Assignments should, just as ordinary equations, be interpreted as equality relations. The difference is that an assignment defines which variable should be computed from the equation. An ordinary equation can basically be used for computing any of the involved variables.

Assignments are the same as variable bindings. The difference is only that the assignment does not appear together with the declaration. There can be at most one valid binding or assignment to a variable when the complete model is considered. Assignments in a class body override inherited assignments and bindings to the same variable. See also the following section about inheritance.

Equations include arithmetical or logical expressions based on functional operators and symbolic references to variables. The expressions must be type consistent. A reference to the time derivative of a variable is made using special operators. For first and second order time derivatives, can be indicated by succeeding the variable with one or two single quotes. For example, `x'` refers to the first order time derivative of `x` while `x''` refers to the second order time derivative. It is also possible to use the more general operator `dot(x,n)`, where `n` is an integer indicating the derivative order.

3.3 Class inheritance

A class inherits all attributes of its super class. If a class defines a local attribute with the same name as an inherited attribute, the local attribute will override the inherited one.

Except for the possibility of overriding, inherited attributes have the same status as locally defined ones. By definition, the set of attributes of a class is the set of local attributes in union with the set of not overridden attributes of the super class.

Connections and equations cannot be overridden since they have no names. However, assignments can be overridden since they can be uniquely identified by the assigned variable. For example, regard the following two model definitions where the second model is a subclass of the first one:

```
M ISA Class WITH
  x TYPE Real := 1.0;
  y TYPE Real := 2.0;
END;

MM ISA M WITH
  x := 3.0;
  y TYPE Integer;
END;
```

In class MM the variable x is inherited from class M but its binding is overridden by a local assignment to the constant 3.0. The variable y is redefined to an unbound integer in MM.

Class inheritance arranges all classes in an inheritance tree. The root of that tree is the predefined class `Class` which is the only class that does not have a super class. Figure 3.1 shows the inheritance tree for all predefined classes in Omola.

3.4 Scope rules and the resolution of identifiers

Components and variables are referred to by names. Names referring to attributes occur in class definitions (the super class name), in connections, in equations, and in binding expressions. When class definitions are manipulated or checked for consistency, names are resolved according to the scope rules of Omola. The scope rules are different for the super class name compared to the other uses of name references. The scope rules are described in the following.

Resolution of super classes
A class definition

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

resolves the super class name reference immediately when the class definition is parsed. The main rule for finding the super class is to search for a global (top level) class definition with the given name. Global class definitions are searched in the current library and possibly in other libraries according to the rules described in the Section 3.8. This default resolve rule applies whenever the super class is given as a single name.

It is possible to modify the super class search procedure by extending the super class name with a library qualification using a double colon like '`LibA::ModelX`'. In this case only the specified library is searched for a global class with the given name. If the library name is omitted and the super class name starts with the double colon like '`::ModelX`', only the current library is searched.

There is a set of special qualifier symbols that modifies the resolve rule to search in the local environment of components rather than in the global library.

`THIS::` When a super class name is preceded by this qualifier, like '`THIS::X`', the super class is searched in the local environment of the class being defined. In this case, `X` is supposed to be a component defined in the owner of the class being defined or in any of its super classes.

`SUPER::` This special qualifier works similar to `THIS::` but the search starts from the super class of the owner of the class being defined. This makes it possible to define class attributes like '`X ISA SUPER::X ... ;`' where the inherited component `X` is redefined and specialized.

`OUTER::` With this special qualifier the search for the super class starts in the owner of the owner of the class being defined. The search works outwards in the component hierarchy.

Here is a simple example using the `OUTER::` and the `THIS::` operators:

```
M ISA KindOfModel WITH
  X ISA Class;
  MM ISA Model WITH
    X ISA Class;
    Y ISAN OUTER::X;  % OUTER::X means M.X
    Z ISA  THIS::X;   % THIS::X means M.MM.X
  END;
END;
```

In this example the super class of Y is resolved into the X defined as an attribute of M. It would also work if X was not locally defined in M but inherited from KindOfModel.

Resolution of variables
All name references in expressions, equations and connections are resolved by searching the local environment and outwards in the component hierarchy. Since inheritance works as if attributes were defined locally, an inherited attribute will always be found before an attribute with the same name defined in the owner class. This default resolve rule can be changed by using the OUTER:: special qualifier before the variable name. In this case the search starts in the owner of the model in which the name reference appears.

Global variables or constants are referred to by preceding the variable name with a library name qualifier or just a double colon meaning global in the current library. A commonly used global variable is the simulated time which is predefined in the Base library (see Appendix B). This variable is referred to as Base::Time.

Dot-notation is used to refer downwards in component hierarchies. For example if model M has a class attribute called X which in its turn has a variable attribute called Y, this variable can be referred to as X.Y in the context of M.

Value semantics
If a class attribute has a variable attribute named value, that class can be used as a variable. When the name of such a class attribute is used in an expression where variables are expected, the value-attribute of the class is used the value. As an example, regard the predefined class Parameter:

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

A model can define parameters and refer to their value-attributes without using dot-notation. For example:

```
M ISA Model WITH
  P ISA Parameter;
  x TYPE Real := P + 2.0;
END;
```

where the name P in the binding expression of x actually refers P.value.

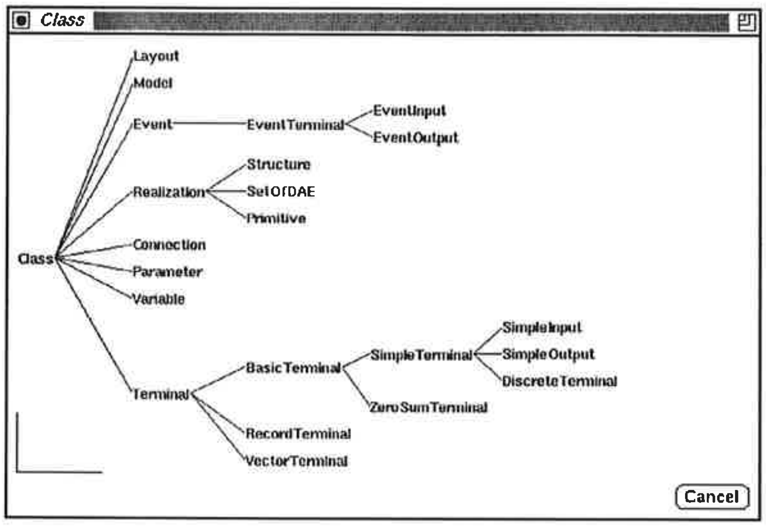


Figure 3.1 Inheritance tree for the Base library.

Variable bindings

The rules for resolving a variable reference make it possible for any expression in a binding or equation to refer to variables downwards as well as upwards in the component hierarchy. However, for the left side of an assignment (the assigned variable) the rules are more strict. Only variables below the assignment in the component hierarchy may be bound. The first element in the dot-notation of the assigned variable must be the name of an attribute present in the same class as the assignment (but it may be inherited). In other words, a model can bind variables in its own submodels and other components but a submodel or a component cannot bind variables defined in their owners.

3.5 The Base library

Omola includes a library called Base, which defines some basic classes. These classes are always assumed to be predefined and they have special meanings to the system and affects the way classes are interpreted as simulation models.

The Base library is shown as an inheritance hierarchy in Figure 3.1. The complete Omola definition of Base is given in Appendix B.

The meaning and usage of some of the classes in Base is described in the following.

Terminal

A terminal is a model component that is a subclass (directly or indirectly) of the predefined class `Terminal`. It is used as a part of the model's interface definition. In a structured model, interaction between submodels is represented by *connections* between terminals. Connections are discussed in the next section.

There are a number of predefined subclasses of `Terminal` that can be used as super classes of model components. `Terminal` itself cannot be used directly as a super class of a model component, it is an empty class meant as a classification for all terminals. Some terminal classes are presented in the following.

BasicTerminal is a subclass of `Terminal`. It is used as a super class for all terminals representing a single interface variable. It defines the following attributes.

`value` is the terminal's value.

`quantity` is a string defining the physical quantity of the terminal. The string must be a valid name of quantity according to the list in Appendix C.

`unit` is a string defining the unit of measure. The string must be a valid SI unit in agreement with the quantity according to Appendix C.

`variability` specifies if the terminal represent an ordinary time varying variable or of if is a parameter which is constant during simulation.

`default` is used for giving a value to an unconnected terminal. If this attribute is bound, and if the terminal is not connected on the outside, the binding expression is used to form an equation with the terminal.

SimpleTerminal is a specialization of `BasicTerminal` with one additional attribute: `causality`. The class is used for all terminals resulting in equalities when they are connected. The `causality` attribute can be used to give the terminal a defined causality either as an input or as an output terminal. Normally the causality is undefined.

ZeroSumTerminal is a specialization of `BasicTerminal` with one additional attribute: `direction`. The class is used for all terminals resulting in zero-sum equations when they are connected. Zero-sum terminals are usually used for representing flows, currents and forces.
The `direction` attribute can be `In` or `Out` defining the direction of positive "flow".

RecordTerminal is a class for structured, multi-variable, terminals. Specializations of this class are supposed to add a set of terminal components which may be basic terminals or other structured terminals.

Other important classes in Base
In addition to `Terminal` discussed above and `Event` discussed below, there are a number of classes in Base which have special meaning to OmSim. These special classes are presented here. There are also a few classes, e.g., `Realization` and its subclasses and `VectorTerminal`, defined in Base but not used in current implementation. These classes are reserved for future use and they are not discussed further in this document.

Model should be used as a super class for all class definitions representing a complete top level model, or classes that are designed to be specialized into complete models or submodels. The class has only one attribute, `Graphic`, which is used by the system for graphical representation of models.

Parameter should be used as a super class for all user parameters, i.e., variables that are not varying continuously but can be accessed and changed by the user of the model. The class defines a `value` attribute and another attribute called `default`, used as value if nothing else is supplied by the user.

Parameters get special treatment by the simulation tools. If the model contains equations and assignments that only involve parameters, these relations will be used to propagate parameter values when they are changed by the user.

Variable should be used as a super class to components that are used instead of ordinary variables. A variable component has an attribute called `initial` which may be bound to any desired initial value.

Layout is used for the graphical representation of models. A model component of class `Layout` is used and manipulated by the graphical editor but ignored by the simulator. For more information, see the comments in of definition of tt Layout in Appendix B and in the Med manual pages.

Connection is used by the graphical representation of connections. Model components of this class are defined and manipulated by the graphical editor.

3.6 Connections

A connection is a class attribute that defines a relation between two terminals. Terminals are components that are defined as subclasses of the predefined class `Terminal`. Terminals are used for defining the interface of a model, i.e., the ports through which it interacts with other models in the environment.

A connection is written as

```
<terminal name> AT <terminal name> ;
```

where that left and right sides have the same status.

Connections represent interaction between models. A connection will be translated into one or several equations between terminal variables. How this is done depends on the properties of the terminals. For example, if two terminals descending from `SimpleTerminal` are connected it will result in an equality.

Connection consistency
Connections are checked for consistency before a model is instantiated. The consistency rules that applies are the following.

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

When simple terminals with defined causality are connected to zero-sum terminals according to rule 5c, the simple terminals will not be included in the zero-sum equation. An input terminal works as a measurement of the connected zero-sum terminal while an output terminal binds the value of the connected zero-sum terminal.

3.7 Discrete events

In Omola it is possible to define models with discrete event behaviour in combination with continuous time behaviour. An *event* is a discontinuity that occurs in a model during simulation. This section will describe how discrete events are defined in Omola.

An event occurs due to some logical condition on variables or due to some other event. The event may cause discontinuities in the continuous time behaviour of the model, i.e., one or several variables may change value abruptly. An event may also cause other events to be scheduled to occur at some time in the future.

Event definitions

An event definition in Omola may define a condition for the event to occur, called the *firing condition*, and the result of the occurrence called the *event action*.

Events are defined as class attributes. An event attribute has the following form:

```
ONEVENT <condition> <action> ;
```

where the keyword "ONEVENT" should be read "on event".

A common way to define an event is to give the firing condition as a logical condition on some state dependent variables and the event action as a set of action statements. This is written as:

```
ONEVENT <logical expression> DO <action statements> END;
```

The 'DO...END' section defines any number of actions, typically state assignments, that will be executed in some order when the event occurs. An event declared in such a way has no name. Named events are sometimes useful and may be declared as components that are subclasses of one of the predefined classes Event or EventTerminal. For named events the firing condition and the action can be defined in separate clauses. For example, the definition of a named event with one firing condition and one action body will look like:

```
E ISAN Event;
ONEVENT <condition> CAUSE E;
ONEVENT E DO <action statements> END;
```

The 'ONEVENT...CAUSE' construction is used to propagate an event condition to one or many named events. The CAUSE keyword may be followed by a list of event names. The advantages of using named events are that one event may have many firing conditions and that one firing condition may cause many events. Firing conditions and actions may also be distributed in different submodels. Named events serves as the basis for event propagation and synchronization discussed below.

Event conditions

Events may be caused by conditions on continuous time variables. Assuming x is variable we may write:

```
ONEVENT x < 0 CAUSE E1;
```

This will cause the event E1 to fire when x goes from positive to negative value. In order for the event to fire a second time the logical condition has to become false and then true again.

In general, the firing condition can be any OR-expression combining logic expressions with named events. For example:

```
ONEVENT x<0 OR y>1 OR Ea OR Eb CAUSE E1;
```

will fire the event E1 if any of the events Ea or Eb occurs or if any of the two inequality expressions become true.

Event actions
A set of actions in a DO...END action body, are not executed in sequence but are sorted and executed in a appropriate order depending on the variables used.

A typical action is an assignment of a new value to a state variable. It is also possible to have other types of event actions like printing a message or scheduling a future event. If there is more than one action body associated with an event, the actions will be sorted together.

In order to define relations between the value of state variables before and after an event, the variable operator 'new' is introduced. The expression 'new(x)' in an action body refers to the value of x immediately after the event. For *discrete* variables, which are discussed below, the new operator may be interpreted as the forward shift operator. Only new-values may be assigned in an event action. That means the normal form of an event action statement is:

```
new(x) := <expression> ;
```

where x is any variable. The right side expression may refer to variables with or without the new operator. This means that it is possible to refer to the value of a variable as it was immediately before the event or as it will become immediately after the event.

All assignments associated with an event will be sorted such that an assignment to a particular variable will be executed before any other assignment referring to new(x) in the right hand side expression. It is an error of the model if such a sorting is not possible.

Event propagation
Events may be propagated to cause other events. An event clause defining event propagation looks like:

```
ONEVENT <or-expression> CAUSE <list of event names> ;
```

where the or-expression is one or many event names separated by OR. The effect of event propagation is that all members in the propagation chain will be treated as one single event, this means that their associated actions will be sorted in proper order.

Events can also propagate between submodels over terminal connections. There is a predefined Omola class called EventTerminal that has the combined functionality of a terminal and an event. A connection between two event terminals will propagate events in both directions. For example, if Et1 and Et2 are event terminals, the connection

```
Et1 AT Et2;
```

will be equivalent to the two event clauses:

```
ONEVENT Et1 CAUSE Et2;
ONEVENT Et2 CAUSE Et1;
```

There are also predefined event terminal classes called EventInput and EventOutput that restrict event propagation to one direction.

Discrete time variables

In a combined continuous time and discrete event model, some variables are varying continuously in time while other variables change step-wise as a result of events. The latter kind of variables are called *discrete time* variables (or simply *discrete* variables), while the former kind is called *continuous time* variables (or simply *continuous* variables).

In Omola, all variables are considered to be continuous time variables unless they are explicitly declared to be discrete. A discrete variable is declared with keyword 'DISCRETE' in the variable definition. For example:

```
X TYPE DISCRETE Real;
```

A discrete variable is considered to have a known value by the continuous time simulation system. This means that the equation manipulation system that sets up the simulation system needs not to find a continuous equation that can be solved for that variable.

Discrete assignments and equations

An equation or an assignment that involves only discrete variables is called a *discrete equation* or a *discrete assignment*.

Discrete assignments may be defined in event action bodies or anywhere in a model. A discrete assignment or equation, not defined in an event action definition, is interpreted as an equality valid at all times. During simulation it has to be evaluated only at those event instances that affect any of the variables in the equation. A discrete equation or assignment can be ignored by the continuous time simulation system.

A discrete assignment defined in an event action body is evaluated only as a result of that particular event. As mentioned before, the assigned variable must always be referred to by the new-operator. For example, with the following declarations:

```
x, y ,z, TYPE DISCRETE Real;
E1, E2, E3 ISAN Event;
z := x+y;
ONEVENT E1 DO new(x):=0.0; END;
ONEVENT E2 DO new(x):= x + 1.0; END;
ONEVENT E3 DO new(y):=2*x; END;
```

it will always be true that $z = x + y$. The relation $y = 2x$ will hold immediately after the event E3 (until some other event changes the value of x). The action of event E2 represents the difference equation $x_{k+1} = x_k + 1$.

Scheduled events

The events considered so far have been triggered by some logic condition on time varying variables. However, it is very common that the exact time when an event will fire is known or can be computed in advance. This is, for example, the case for sampled systems where the time for the next sample event is known. Such an event can be *scheduled*, i.e, it can be put in queue of time ordered events. The system can then be simulated more efficiently.

A special procedure call: '`schedule(E,delay)`', where `E` is a name of an event, may be used in event action bodies. The effect is that the event `E` will be scheduled to occur in `delay` time units from current time. If `delay` is zero the new event will be handled immediately after the current event.

As an example of using scheduled events, a sampled data model can be represented by the following Omola code. The event `Init` is special since it will always be called by the simulator at the start of a new simulation.

```
DiscreteSyst ISA Model WITH
  h TYPE Real := 1.0; %Sample interval
  Sample, Init ISAN Event;
  OnEvent Sample OR Init DO
    % ... variable assignments
    schedule(Sample, h);
  END;
END;
```

3.8 Omola files and model libraries

An omola file may contain any number of class definitions. A class defined at file level, i.e., not encapsulated in any other class definition, is called a global class. Every global class belongs to a *library*. An Omola file may contain a library statement, declaring to which library the following definitions belong. A library statement looks like:

```
LIBRARY <library name> ;
```

The library name can be any valid name string. The statement sets the current library that will be valid until the next library statement or until the end of the file. A library called `Scratch` will be used as current library until the first library statement appears in the file.

A file may contain several library statements but it is recommended to use a one-to-one correspondence between files and libraries and to have a library statement first in every Omola file.

A library is a separate name-space for global definitions. Normally when a class definition is parsed, if the super class is given as an unqualified name, only the current library is searched. However, it is possible to give a search path of libraries that are to be searched for unqualified name references. Such a statement is given in an Omola file as:

```
USES <lib1>, <lib2>, ... ;
```

and it is usually put directly after the library statement. Global names will be searched starting with current library and then in the libraries given in the `USES` statement, beginning from the last library in the list. A `USES` statement is valid until the next statement or until end of file.

Global constants

Variables bound to constant values may be defined globally in an Omola file. The definition is similar to a variable attribute definition with a binding. It has the form:

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

where the expression must evaluate to a constant value. A reference to a global constant in a class definition must include a double colon '::'. For example, a global constant named `Pi`, must be referred to as '`::Pi`'.

A. Omola Syntax

```
omola-definitions ::=
        /* empty */
        omola-definitions id class-def ;
        omola-definitions id type-declaration ;
        omola-definitions LIBRARY id ;
        omola-definitions USES name-list ;
        omola-definitions block ;
        omola-definitions COMMENT
        omola-definitions error ;

class-def ::=
        super-class-def
        super-class-def WITH class-body END

super-class-def ::=
        ISA reference

class-body ::=
        body tag-body

body ::=
        /* empty */
        body body-item ;
        body COMMENT

body-item ::=
        name-list class-def
        name-list type-declaration
        reference := expr
        expr = expr
        reference AT reference
        event-handler
        error

event-handler ::=
        ONEVENT event-body
        WHEN event-body

event-body ::=
        expr DO actionbody END
        expr CAUSE expr-list

actionbody ::=
        /* empty */
        actionbody function-designator := expr ;
        actionbody function-designator ;
        actionbody COMMENT

tag-body ::=
        /* empty */
        tag-body tag body

tag ::=
        TAG

type-declaration ::=
        TYPE var-kind type-designator
        TYPE var-kind type-designator  := expr

var-kind ::=
        /* empty */
        DISCRETE

type-designator ::=
        MATRIX [ expr , expr ]
        ROW [ expr ]
        COLUMN [ expr ]
        REAL
        INTEGER
        STRING
        ( name-list )
        POLYNOMIAL
        REFERENCE
        SYMBOL
        id

expr-list ::=
        expr
        expr-list , expr

expr ::=
        IF expr THEN expr ELSE expr
        expr AND expr
        expr OR expr
        NOT expr
        expr REL-OP expr
        expr .. expr
        expr ADD-OP expr
        expr MUL-OP expr
        expr HAT expr
        expr DOTHAT expr
        ADD-OP expr
        expr QUOTE
        primary
        error

primary ::=
        reference
        QUOTE id
        matrix
        polynomial
        REAL
        INTEGER
        STRING
        ( expr )
        function-designator

matrix ::=
        [ rows ]

rows ::=
        columns
        rows ; columns

columns ::=
        expr
        expr : expr : expr
        columns , expr
```

```
polynomial ::=
        LCBRACK poly-item RCBRACK

poly-item ::=
        c-poly
        r-poly

r-poly ::=
        expr : expr-list

c-poly ::=
        expr-list

function-designator ::=
        id ( expr-list )
        id ( )

name-list ::=
        id
        name-list , id

indexed ::=
        id
        id [ expr-list ]

indexed-list ::=
        indexed
        indexed-list . indexed

reference ::=
        indexed-list
        : : indexed-list
        id : : indexed-list

block ::=
        begin statement-list END

begin ::=
        BEGIN

statement-list ::=
        /* empty */
        statement-list statement ;

statement ::=
        reference id
        reference id ( reference )
        reference ocl-arg-list
        reference := expr
        FOR reference := expr block
        block
        error

ocl-arg-list ::=
        /* empty */
        ( )
        ( expr-list )
```

B. The Base Library

```
LIBRARY Base;

Time TYPE Real;

Layout ISA Class WITH
  %% Graphical layout of a class

  x_pos TYPE Real;
  y_pos TYPE Real;
  %% Position of center, relative lower-left corner of
  %% enclosing block, in screen coordinates.

  x_size TYPE Real := 400;
  y_size TYPE Real := 300;
  %% Size of block in screen coordinates.

  invisible TYPE Integer := 0;
  %% If true, the enclosing model should be invisible in a block
  %% diagram; may currently be used for terminals on components.

  %% A layout may also have these attributes, which are
  %% undefined by default.
  %%
  %% bitmap Name of a bitmap file; "sum" denotes
  %% bitmap file "sum.bm".
  %%
  %% label Used to label a model.
END;

Model ISA Class WITH
attributes:
    Graphic ISA Layout;
    % Primary_Realization TYPE String := "Any realization name";
END;

Event ISA Class;

EventTerminal ISAN Event;

EventInput ISAN EventTerminal WITH
  causality TYPE Symbol := 'input;
END;

EventOutput ISAN EventTerminal WITH
  causality TYPE Symbol := 'output;
END;

Realization ISA Class;

Structure ISA Realization;

SetOfDAE ISA Realization;

Primitive ISA Realization; % same as SetOfDAE

Connection ISA Class; % Connection with breakpoints.

Parameter ISA Class WITH
```

```
attributes:
    value TYPE Real;
    default TYPE Real := 0.0;
END;

Variable ISA Class WITH
attributes:
    value TYPE Real;
    initial TYPE Real := 0.0;
END;

Terminal ISA Class WITH
    Graphic ISA Layout;
END;

BasicTerminal ISA Terminal WITH
attributes:
  value TYPE Real;
  quantity TYPE String := "number";
  unit TYPE String := "1";

  variability TYPE (TimeVarying, Parameter) := 'TimeVarying;
  default TYPE Real;
END;

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

SimpleInput ISA SimpleTerminal WITH
  % Corresponds to an input in Simnon.
  causality := 'Input;
END;

SimpleOutput ISA SimpleTerminal WITH
  % Corresponds to an output in Simnon.
  causality := 'Output;
END;

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

DiscreteTerminal ISA SimpleTerminal WITH
  value TYPE Discrete Real;
END;

RecordTerminal ISA Terminal WITH
components:
END;

VectorTerminal ISA Terminal WITH
attributes:
    Length TYPE Integer;
END;
```

C. Quantity Database

Name of quantity	**Unit**
number	1
Space and time quantities:	
angular.acceleration	rad/s2
angular.velocity	rad/s
angle	rad
acceleration	m/s2
velocity	m/s
length	m
breadth	m
height	m
thickness	m
radius	m
diameter	m
length.of.path	m
area	m2
volume	m3
time	s
Periodic quantities:	
period	s
time.constant.of.an.exponentially.varying.quantity	s
frequency	Hz
rotational.frequency	Hz

Name of quantity	Unit
Mechanics quantities:	
mass.flow.rate	kg/s
mass	kg
density	kg/m3
specific.volume	m3/kg
momentum	kg.m/s
impulse	N.s
angular.momentum	kg.m2/s
angular.impulse	N.m.s
moment.of.inertia	kg.m2
force	N
weight	N
moment.of.force	N.m
bending.moment	N.m
torque	N.m
pressure	Pa
normal.stress	Pa
shear.stress	Pa
viscosity	m2/s
power	W
work	J
energy	J
potential.energy	J
kinetic.energy	J
volume.flow.rate	m3/s
Heat quantities:	
thermodynamic.temperature	K
celsius.temperature	degC
heat.flow.rate	W
heat	J
density.of.heat.flow.rate	W/m2
thermal.conductivity	W/(m.K)
coefficient.of.heat.transfer	W/(m3.K)
heat.capacity	J/K
specific.heat.capacity	J/(kg.K)
entropy	J/K
internal.energy	J
enthalpy	J
helmholtz.free.energy	J
gibbs.free.energy	J
specific.entropy	J/K/kg
specific.enthalpy	J/kg
specific.internal.energy	J/kg

Name of quantity	**Unit**
Electricity and magnetism quantities:	
electric.current	A
electric.charge	C
electric.field.strength	V/m
electric.potential	V
electric.voltage	V
electric.flux	C
capacitance	F
permittivity	F/m
magnetic.field.strength	A/m
magnetic.flux	Wb
self.inductance	H
permeability	H/m
resistivity	ohm.m
conductivity	S/m
reluctance	H-1
impedance	ohm
reactance	ohm
resistance	ohm
power	W
active.power	W
apparent.power	W
reactive.power	W

D. The BasicElectrical library

This is the definition the basic electrical library used in the tutorial. All graphical information have been stripped from the models in order to save space and to make the models easier to read.

```
LIBRARY BasicElectrical;
%%
%% Basic library of general electric components. Developed to be used
%% as an example in "OmSim and Omola Tutorial and User's Manual,
%% Version 3", M. Andersson, 1993.
%%

VoltageTerminal ISA SimpleTerminal WITH
  unit := "V";
  quantity := "electric.potential";
END;

CurrentTerminal ISA ZeroSumTerminal WITH
  unit := "A";
  quantity := "electric.current";
END;

ElectricTerminal ISA RecordTerminal WITH
  V ISA VoltageTerminal;
  I ISA CurrentTerminal;
END;

ElectricModel ISA Model;

ElectricNode ISA ElectricModel WITH
  %% Connection point for electric terminals.

  T ISA ElectricTerminal;

  T.I = 0.0;
END;

ElectricTwoPole ISA ElectricModel WITH
  %% Electrical component with two ports.

  T1, T2 ISA ElectricTerminal;

  V, I TYPE Real;
  %% I is current through component, into T1.
  %% V is voltage drop over component; T1 is positive.

  T1.I + T2.I = 0;
  V = T1.V - T2.V;
  I = T1.I;
END;

ElectricTwoPoleV ISA ElectricTwoPole WITH
  %% Electric two pole with vertical teminal layout.
END;

Resistor ISA ElectricTwoPole WITH
  %% Resistor with parameter resistance.

  R ISA Parameter WITH default := 1.0; END;
  V = I * R;
END;

Capacitor ISA ElectricTwoPole WITH
  %% Capacitor with parameter capacitance.
  %% Model may be refined to have time varying capacitance
  %% by replacing C.

  C ISA Parameter WITH default := 1.0; END;

  dot(C*V) = I;
END;

Inductor ISA ElectricTwoPole WITH
  %% Inductor with constant inductance.

  L ISA Parameter WITH default := 1.0; END;

  V = L*I';
END;


GroundPoint ISA ElectricModel WITH
  %% Defines the electric potential reference.
  %% Every complete model of an electric circuit should
  %% have exactly one of these.

  T ISA ElectricTerminal;

  T.V = 0.0;
END;

VarCapacitor ISA Capacitor WITH
  %% A capacitor with variable capacitance.

  C ISA SimpleTerminal;
END;

Switch ISA ElectricTwoPole WITH
  %% Electric switch controlled by external events.

  Open, Close ISAN Event;

  is_closed TYPE DISCRETE Integer;

  0 = IF is_closed THEN V ELSE I;

  OnEvent Open DO
    new(is_closed) := 0;
  END;

  OnEvent Close DO
    new(is_closed) := 1;
  END;
END;
```