ISSN 0280-5316
ISRN LUTFD2/TFRT--5726--SE

Code Generation from JGrafchart to Modelica

Isolde Dressler

Department of Automatic Control
Lund Institute of Technology
March 2004

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

Document name
MASTER THESIS

Date of issue
March 2004

Document Number
ISRNLUTFD2/TFRT--5726--SE

Author(s)
Isolde Dressler

Supervisor
Karl-Erik Årzén LTH, Lund
Eckehard Steinbach Techn. Universität, München

Sponsoring organization

Title and subtitle
Code Generation from JGrafchart to Modelica (Kodgenerering från JGrafchart till Modelica)

Abstract
In this thesis the possibility to use the discrete-event modelling tool JGrafchart as a front end to the object-oriented modelling language Modelica has been studied. In this context, a code generator from JGrafchart to Modelica has been implemented. The code generator includes the basic JGrafchart objects like steps, transitions and variables, but also hierarchical elements like macrosteps and inlined procedure documents. The generated Modelica model imitates the JGrafchart execution model in an algorithm section. The code generator is finally tested on an example of a controller or at tank system.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
86

Recipient's notes

Security classification

Contents

Chapter 1

Introduction

The basis of this work consists of two different simulation languages, Modelica and JGrafchart. Whereas JGrafchart is designed to model sequential, procedural and state machine oriented control applications, Modelica is very powerful for simulating continuous multi-domain systems.

The name JGrafchart [1] refers to the graphical programming language as well as the programming editor. JGrafchart is based on Grafcet, sequential function charts and statecharts. It can be used both as a development tool and as a run-time system.

Modelica is an object-oriented modelling language. Its strong point is the simulation of continuous hybrid systems. Subsystems of different domains can easily be connected to a larger system, e.g. an industrial robot, where hydraulic, mechanical and electrical parts are aggregated. Dymola by Dynasim is a commonly used simulation environment for Modelica. It is also used in this thesis.

Remelhe [5] and Ferreira [6] examined the possibility of discrete-event modelling in Modelica. Whereas Remelhe implemented a modelling environment for statecharts and sequential function charts that generates a Modelica model, Ferreira developed a statecharts library for Modelica. This library is based on the Petri nets library [7], which provides models for places and transitions in order to build simple networks. However, Modelica has only limited support for sequential, state oriented applications. Those are often needed to model more complex control logic.

A known example for the combination of discrete and continuous modelling is Simulink with Stateflow. A finite state machine modelled with Stateflow can easily be integrated in a Simulink diagram. Compared to Simulink, Modelica is more suitable for multi-domain models and as its diagrams are not necessarily directional, Modelica models are more flexible. Statechart only implements finite state machines consisting of states and transitions

with the possibility of building hierarchies, whereas JGrafchart includes procedural programming and more advanced elements for building hierarchies.

The aim of this thesis is to investigate how JGrafchart can be used as a front end to Modelica. With the aid of JGrafchart sequential and state oriented control logic can then easily be modelled and used in connection with continuous Modelica models. In this way the advantages of both programs are combined.

The approach that is investigated is code generation from JGrafchart to Modelica. A model constructed in JGrafchart is translated into Modelica, so that the generated Modelica model can be used for simulations with Dymola.

Modelica and JGrafchart have been examined and the differences in semantics between the two languages studied. The behaviour of the JGrafchart model is analysed in order to find out the specifications for the Modelica model. By looking into Modelica the limitations of the code generation are examined and a subset of JGrafchart is identified that can be translated into Modelica code.

On this basis the code generator is then implemented. The model's behaviour should be the same in both environments. In that context it is also investigated under what conditions a non-working Modelica model is generated and in how that can be avoided. Furthermore, the possibility of integrating animation in the case of real-time simulation in Dymola is examined.

In addition the code generator and the resultant model are tested in an example application.

In the following chapter an overview over JGrafchart is given including the relevant elements and the execution model. In the third chapter the basic principle of Modelica is explained together with the structure of composed models. In Chapter 4 through 6 the code generator and the generated program are presented. After an overview of the generator and the program, the code generation for the different JGrafchart elements is explained in detail. Then the graphical appearance and animation of the Modelica model is discussed. In Chapter 7 an example application is presented and the resulting Modelica model discussed. In Chapter 8 the limitations and assumptions of the code generation are resumed and in Chapter 9 a summary of the project is given and the conclusions are presented.

Chapter 2

JGrafchart overview

JGrafchart is a graphical programming language for sequential, procedural and state-oriented applications [1]. It is based on Grafcet, sequential function charts and statecharts. JGrafchart handles sequences and hierarchical state machines. It is both object-oriented and procedural. The graphical object editor, that is also named JGrafchart, can be used as a modelling environment as well as a real-time execution system. It can have a connection to an external environment. JGrafchart is implemented in Java 2 and Swing together with some external software components.

2.1 Steps and transitions

The basic two elements of JGrafchart are steps and transitions. Steps represent the possible states a system can have. Transitions model the conditions on which the system can change its state.

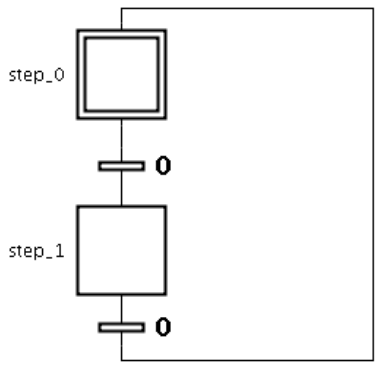

Figure 2.1: Example network with steps and transitions

By connecting these two elements alternately, a network is built that represents the state model of a system (see Figure 2.1). In JGrafchart the expression "document" is adopted for the compound of objects the model contains. During simulation this model is traversed. The execution of the simulation is based on an algorithm that is executed periodically with a specified scan cycle.

A JGrafchart network can only be traversed in one direction, each object has an input and an output port. A transition has an attached boolean condition. If this condition is true and the preceding step is active, the transition fires. Then the preceding step is deactivated and the following step activated. The active steps are marked with a token during simulation. In one scan cycle, a token cannot traverse more than one transition. A transition can be connected to exactly one step at each port, whereas a step can be linked to more than one transition per port. If more than one transition condition becomes true at the same time, all the corresponding transitions will fire and activate their following step. This behaviour is deterministic, but it can be avoided by making the transition conditions mutually exclusive.

As a transition can only be connected to one step at each port, special objects have to be used to connect a transition to two or more network branches. They are called parallel joins and parallel splits (see Figure 2.2). In the case of a parallel join both preceding steps have to be active in order to fire the transition and for a parallel split both succeeding steps are activated.

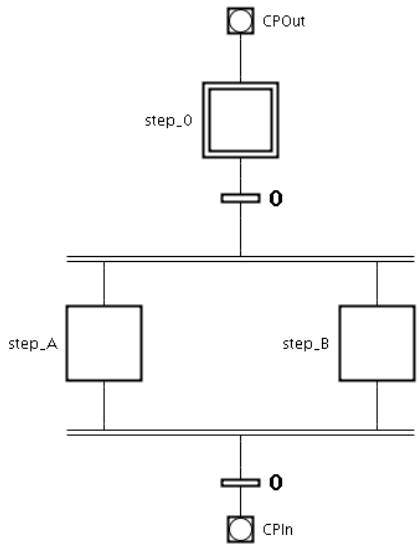

Figure 2.2: Network with parallel split and join and connection posts

To facilitate the overview of long extended networks, those can be disconnected and the corresponding ends connected to a pair of connection posts (see Figure 2.2). These objects create connections without direct graphical link.

There is also a special type of step called initial step. These steps are activated at the begin of a simulation. They are distinguished by a double frame.

A step can have actions. An action consists of an action qualifier that specifies when it is executed and the action itself. An action can contain an assignment or a method call.

Action qualifier	Description
S	Entry action: executed in the scan cycle the step becomes active
P	Periodic action: executed every scan cycle while the step is active
X	Exit action: executed in the scan cycle the step is deactivated
N	Normal action: the state of the step is assigned to a variable
A	Abort action: executed if the step is aborted

Table 2.1: The different action qualifiers

For example, if the variable `height` is to be initialized with the value 0 every time a specific step is entered, the action of this step would look like that:

```
S height = 0;
```

For every JGrafchart object a dialog window can be opened, where its properties can be specified, like the condition for transitions or the actions for steps.

For steps three different methods are available. Two of them give back the time a step has been active or 0 if it is deactivated, `step.s` in seconds and `step.t` in scan cycles. The method `step.x` gives back the state of the step.

Several other methods exist that can be used in actions or transition conditions. A major part of them sets or reads graphical or geometrical properties of the JGrafchart objects. Also mathematical functions like sine

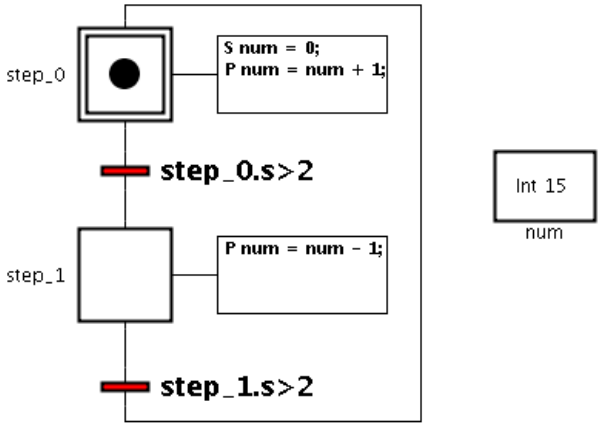

Figure 2.3: Example network

or square root are available. Other methods enable the access to objects, e.g. writing a value into a list.

In the example in Figure 2.3 you can see a simple network with two steps and two transitions. The upper step, `step_0` is an initial step that is currently active. When the condition of the following transition becomes true after two seconds, the token will enter `step_1`. The actions are visible in the action block. When `step_0` is entered, the variable `num` is set to 0 and then increases by 1 each scan cycle. When `step_1` has been activated, `num` decreases by 1 each scan cycle.

2.2 Variables, input and output

There are four different types of variables in JGrafchart: boolean, integer, real and string. A variable can have an initial value that it takes at the start of the simulation. Otherwise it keeps the last assigned value or has the standard value 0 for real and integer variables and false for booleans. If the attribute constant is set to true, it cannot be changed by any actions during simulation. Figure 2.4 shows a boolean variable with its object dialog.

These four types are also available as lists. The basic difference to an array known from e.g. the programming language C is that elements can be added or deleted during simulation, i.e. the size of a list is not fixed. There are different methods for lists to change, add, read or delete elements.

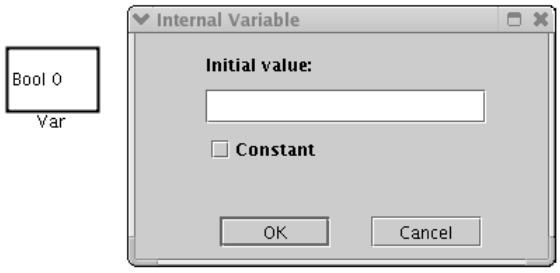

Figure 2.4: Boolean variable with object dialog

In order to communicate with a real process, there are different forms of input and output. Communication with input or output channels can be achieved with digital and analog input or output elements. Furthermore there are socket input and output elements for the basic four variable types. Figure 2.5 shows the boolean socket input and output elements. With the aid of these TCP messages can be sent to or received from a server. In addition objects to exchange xml-messages exist.

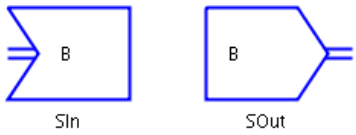

Figure 2.5: Boolean SocketIn and SocketOut

2.3 Workspaces and macrosteps

Workspaces and macrosteps enable the construction of hierarchies in JGrafchart.

A workspace (see Figure 2.6) is only a repository for a sublevel document. It has no state or actions itself. The workspace has no connection to the upper level network. The scan cycle of the sublevel model can be modified to an integer multiple of the top level scancyle. Additionally, the workspace can be disabled so that the transitions of the sublevel document will not fire and except the entry actions of initial steps no actions are executed.

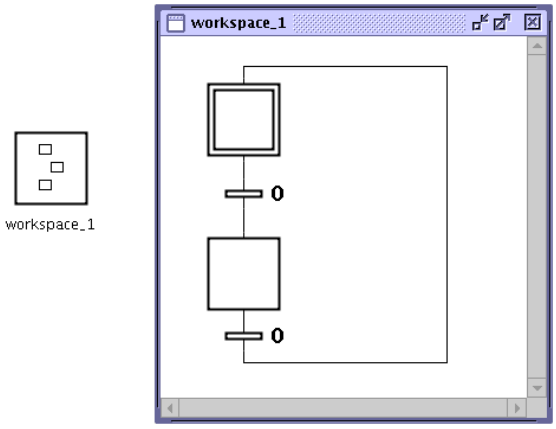

Figure 2.6: Workspace

A macrostep (see Figure 2.7) is another hierarchical element. It contains a model as well as it is a step that can have actions. In contrary to a workspace it is a part of the upper level network.

A macrostep can have several entries and exits. Every entry/exit is connected with an enter/exit step in the inner document. A macrostep like this can be entered by more than one port at the same time and have more than one active step inside. The macrostep itself is active as long as there is an active step inside. To enable the transition connected to a specific outport of a macrostep, the exit step corresponding to this output port has to be active.

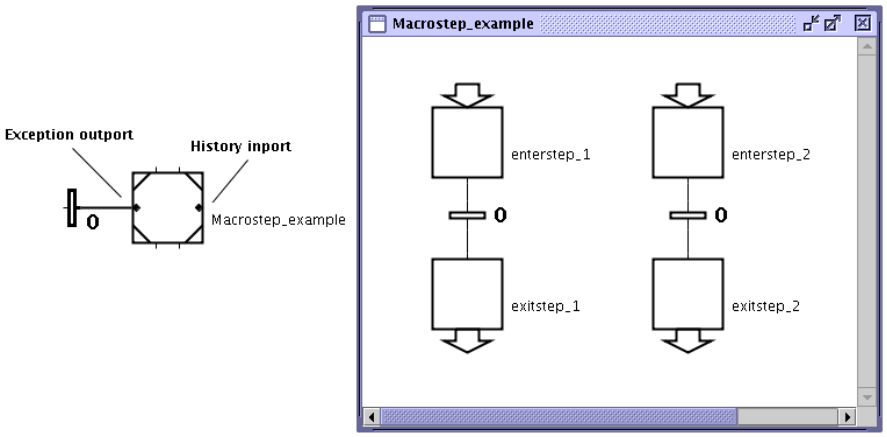

Figure 2.7: Example of a macrostep

When a macrostep is entered, the entry actions of the macrostep are executed before those of the enter step inside. When the macrostep is left, first the exit actions of the exit step inside are executed and then those of the macrostep. The periodic actions of the macrostep are executed after the inner steps' ones.

A macrostep always has two special ports. By leaving the macrostep through the exception outport, the whole macrostep including all inner steps is deactivated. Then the abort actions of the macrostep and the steps that have been active are executed. This port has to be connected to a special form of transition, an exception transition. The exception transition is not connected to a specific exit step inside the macrostep. It is sufficient that the macrostep is active to enable the firing. The exception transition has priority over the transitions inside the macrostep. If the exception transition fires, none of the inner transitions can fire in the same scan cycle.

If the macrostep once has been aborted, its state is stored and can be resumed by entering the macrostep through the history inport. Then the inner steps that have been active before the abortion are entered again and their enter actions are executed. This only works correctly if the macrostep has been aborted before.

2.4 Procedures

A procedure (see Figure 2.8) is an element containing a sublevel document with one enter and one exit step. It cannot directly be connected to transitions but can be entered by a procedure call in different ways. Procedures are reentrant, i.e. they can be entered through different ways at the same time. Then an independent copy of the procedure is entered in each case.

The most common way to call a procedure is from a process step or a procedure step (see Figure 2.9). However, you can also call a procedure directly via the procedure dialog or with the action method `spawn`.

A procedure can have parameters. They are represented by variables in the procedure document. During the procedure call, parameters can be handled as "called by value" (indicated by a "V") or "called by reference" (indicated by an "R"). The object dialog in Figure 2.9 shows an example of the two different parameter calls. In the first case, the parameter will take a specific value when the procedure is entered. The value is given by an expression that is evaluated at the time the procedure is called. The "call by reference" connects the parameter with a variable outside the procedure in such a way that a change on the inner variable affects the outer one.

Like macrosteps, procedure and process step also have their own actions.

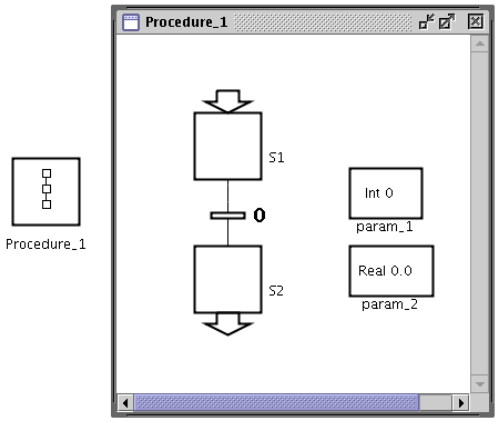

Figure 2.8: Procedure

However instead of containing a document, they call a procedure. In the dialog the procedure name or a string variable that specifies the name of the procedure can be indicated. By using a string variable a procedure or process step can call different procedures during the simulation run. You also specifiy the parameters of the procedure call and the way they are called, "by value" or "by reference".

The order in which the entry and periodic actions of the procedure step and the inner steps of the procedure are executed corresponds to macrosteps. The difference between procedure and process steps consists in the way they are deactivated. A procedure step can only be left when the exit step of the procedure is active. The actions of the exit step are executed in the same way as in the case of a macrostep. A process step can be deactivated as soon as the transition condition is true, independent of the procedure's status. The procedure document is traversed until the exit step is entered. Then the entry actions of the exit step are executed and the procedure call is ended. The periodic or exit actions of the exit step are not executed. In contrast to procedure steps, a process step is consequently able to call a procedure anew before its last procedure call is ended.

A procedure step has, like a macrostep, an exception outport that may be connected to an exception transition, but no history inport.

The common characteristics of procedure and process steps are easy to explain looking at the Java classes. The class `ProcessStep` inherits the class `ProcedureStep`. `ProcedureStep` in turn inherits the class `MacroStep`.

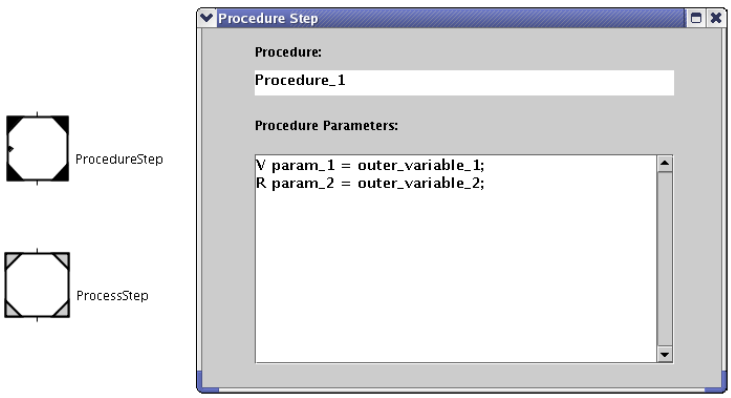

Figure 2.9: Procedure step with object dialog and process step

2.5 Object references in JGrafchart

In JGrafchart an object reference can be global or local. A global reference contains the whole path of an object, beginning at the top level workspace and naming every hierarchy level seperated by a dot. For example, the global reference to `variable` in Figure 2.10 is `J1.macrostep.variable`. A local reference contains only a part of the global path, mostly only the object name. JGrafchart looks for the object beginning in the actual workspace and going to upper levels until an object with the given name is found or the top level workspace is reached. Therefore the transition in `macrostep` can refer to `number` only with its name, whereas the transition in the top level workspace has to refer to `variable` inluding the name of the macrostep.

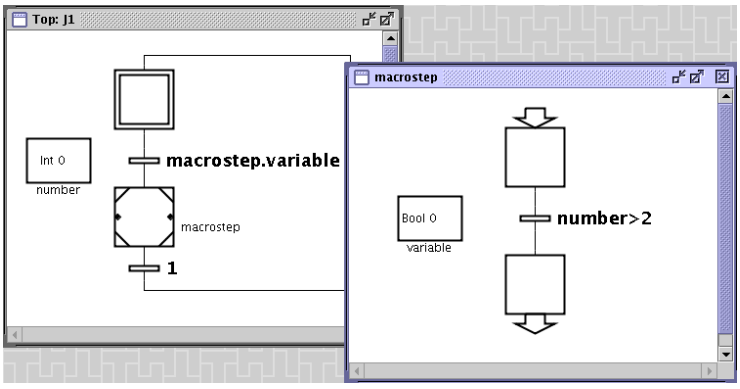

Figure 2.10: Illustration of references

2.6 Execution model

The execution of a JGrafchart model is based on a periodical algorithm. Every scan cycle the following operations are performed.

1. First the digital and analog input channels are read.
2. Then all the transitions are checked for firing. In the first pass those that should be fired are only marked. Only in the second pass the marked transitions are actually fired. For the neighbouring steps exit or entry actions are then executed and their state is updated. A step has two variables for its state, `x` and `newX`. At that time only `newX` is updated.
3. Then all the steps in the model are traversed. Now `newX` is assigned to `x` and the step timer is updated. Periodic actions are executed if the step has been active in the last scan cycle and is not deactivated in the actual one, normal actions are executed if the state of the step has changed.

In the first scan cycle, the initial steps are entered and their stored and normal actions are executed.

The following example illustrates the chronology of the different actions and how the timer of a step is updated. In Table 2.2 an overview for the example network in Figure 2.11 is given. For the timer of the steps the already updated value is specified.

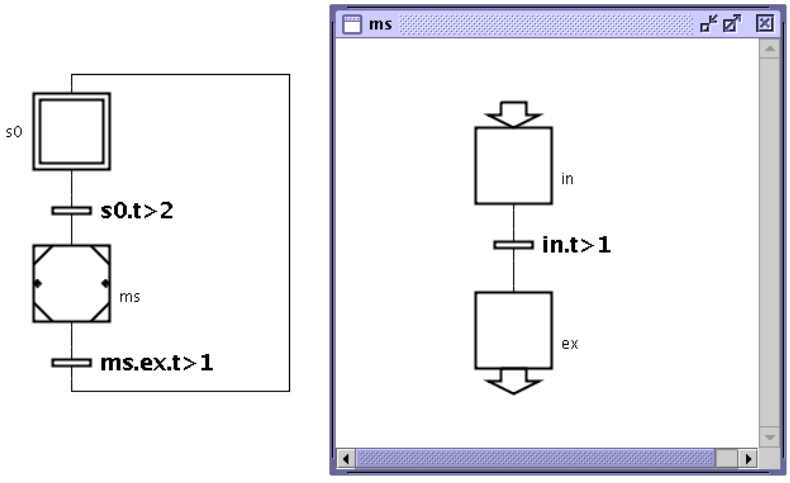

Figure 2.11: Example network

Scan cycle	Executed actions	Timer s0	Timer ms	Timer in	Timer ex
1	Entry action s0	0	0	0	0
2	Periodic action s0	1	0	0	0
3	Periodic action s0	2	0	0	0
4	Periodic action s0	3	0	0	0
5	Exit action s0				
	Entry action ms				
	Entry action in	0	0	0	0
6	Periodic action in				
	Periodic action ms	0	1	1	0
7	Periodic action in				
	Periodic action ms	0	2	2	0
8	Exit action in				
	Entry action ex				
	Periodic action ms	0	3	0	0
9	Periodic action ex				
	Periodic action ms	0	4	0	1
10	Periodic action ex				
	Periodic action ms	0	5	0	2
11	Exit action ex				
	Exit action ms				
	Entry action s0	0	0	0	0
	...				

Table 2.2: Execution of the example network

The example shows that it is only in the scan cycle after a step has been activated that periodic actions are executed and the timer increased. Periodic actions are executed one more time as one could think at first looking at the transition conditions. But that can be explained since the step is only left when its timer is greater than the indicated value.

Chapter 3

Modelica overview

Modelica [2] is an object-oriented modelling language. It has been and is developed by the Modelica Association [3], a non-profit organization founded for this purpose. Its aim is to facilitate simulations for complex multi-domain applications. Dymola [4] is a commercial implementation of Modelica, which is used in this thesis.

3.1 The basic principle

A Modelica model consists basically of a declaration part and an equation section. There can also be an algorithm section instead of, or in addition to, the equation section.

The characteristic of Modelica is the equation section. Every physical system can be described by a system of equations; differential, algebraical or discrete ones. Instead of converting the equation system into an algorithm, the equation system can be represented directly in Modelica. An advantage of this is that the unknowns are not fixed, so that the model is more flexible.

In contrast to the equation section, the algorithm section contains assignments. Through an assignment, a variable is assigned the value of an expression, whereas an equation has to be fulfilled, each variable can vary to achieve that. In the equation section it does not matter in which order the equations are written, whereas in the algorithm section the order of the assignments is important.

During the simulation run, the system of equations and assignments is solved for every simulation step. Unless an if clause restricts the validity of an equation or assignment, it has to be fulfilled during the whole simulation time. The algorithm section is executed every simulation step.

The model has to contain exactly as many equations or assignments as

variables, otherwise the equation system is under- or over-determined. Assignments to the same variable do not count several times. But an assignment is counted even if an if clause restricts it in such a way that it will not be executed during the simulation. While you can have unused variables in JGrafchart, Modelica is very strict in this point.

The following model specifies a simple LC-oscillator. A model always starts with the keyword `model` and its name, and ends with `end` and the name. In the first part the variables and parameters are declared. By using the keyword `parameter` or `constant` a variable cannot be modified during simulation. The equation section contains the two differential equations describing the oscillator.

```
model LC_oscillator
  SI.Voltage u;
  SI.Current i;
  parameter SI.Capacitance C = 1;
  parameter SI.Inductance L = 1;
equation
  L*der(i) = u;
  C*der(u) = -i;
end LC_oscillator;
```

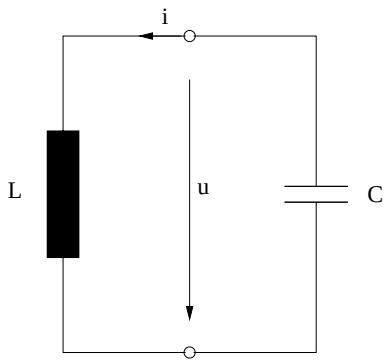

Figure 3.1: LC oscillator

3.2 Composed models

Modelica is an object-oriented language. Reuse of declared models is advantageous. Thus, in order to build a model, small reusable parts of the model are declared and then joined to a larger model.

According to this the oscillator model would look like:

```
model LC_oscillator
  capacitor C1;
  inductor L1;
equation
  connect(C1.p, L1.n);
  connect(L1.p, C1.n);
end LC_oscillator;
```

Here the oscillator is put together of a capacitor model and an inductor model. The two components are connected in the equation section. The function `connect()` generates equations that equate the currents and voltages at the components' pins. The differential equations of the former model have disappeared, they are now part of the capacitor and inductor model.

In order to allow the connections, the models of capacitor and inductor have to contain variables that refer to voltage and current at the pins. These are assembled in another element, a connector.

```
model capacitor
  SI.Voltage u;
  SI.Current i;
  parameter SI.Capacitance C = 1;
  pin p;
  pin n;
equation
  C*der(u) = i;
  u = p.u - n.u;
  0 = p.i + n.i;
  i = p.i;
end capacitor;

connector pin
  SI.Voltage u;
  flow SI.Current i;
end pin;
```

These elements are also available in the Modelica Standard Library. The library is to a large extent based on reusable elements, e.g. all components with two pins are based on a general one port component. A class can inherit another class by specifying the parent class with "`extends` *parent class*;" in

the child class. In addition to that the models contain code for graphical annotations to enable a graphical construction of systems.

A model can have an icon that represents this model in the diagram of an upper level model. In the diagram the relations between the sublevel parts of a model are shown through graphical connections between the different icons. In order to create a graphical connection between two icons, those must have terminals.

If the signal direction between the different parts of a model is to be considered, a `block` can be used instead of a `model` and the variables in the connectors specified with the keywords `input` and `output`. The resulting model is then similar to a model known from e.g. Simulink.

There are more classes available in Modelica. Different models can be assembled in a library with `package`. If several variables shall be assembled, a `record` can be used. A `function` can be declared for recurring calculations. New variable types can be declared with `type`, new classes with `class`.

3.3 Discrete-event modelling in Modelica

Discrete-event models can be implemented based on equations, like e.g. the Petri nets library [7], as well as based on algorithms and functions [5] [8]. The first case shows synchronous behaviour between the discrete-event and the continuous model parts, as all equations are evaluated at every time instant. In the second case asynchronous behaviour can be modelled. When a discrete event occurs, the integration of the equation part is halted until the algorithm part of the model is completely executed. The code generator implements the model with algorithms; in the following some Modelica elements that are useful in this context are explained.

The algorithm section of a model or a function contains assignments which are always evaluated in the same calculation order.

variable := *expression*;

If an assignment depends on a condition, several Modelica language elements can be used. In a conditional assignment statement, an assignment is executed according to a condition.

variable := `if` *condition* `then` *expression1* `else` *expression2*;

Another possibility to express the above statement is to use an if clause. The content of an if clause is only evaluated while its condition is true. The different parts of an if clause have to contain the same number of assignments on the same variables.

```
if condition then
  assignments
else
  assignments
end if;
```

The content of a when clause is evaluated at the moment the condition becomes true. Contrary to an if clause, the assignments are only executed once in a period in which the condition is continuously true.

```
when condition then
  assignments
end when;
```

The following example shows the difference between an if and a when clause. In the first case `variable1` always has the same value as `variable2`. In the second case `variable1` is assigned the value of `variable2` once at the simulation start. So even if `variable2` changes, `variable1` remains constant.

```
if time>=0 then
  variable1 := variable2;
end if;

when time>=0 then
  variable1 := variable2;
end when;
```

A when clause can always be transformed into an if clause with the aid of the `edge()` operator. It gives back the value "true" at the moment the expression in the brackets becomes true, else it gives back "false". The following if clause is equivalent to the when clause in the example above. Instead of `edge(time>=0)`, `initial()` is used here. It is true at the instant the simulation starts and false otherwise.

```
if initial() then
  variable1 := variable2;
end if;
```

In Modelica, variables can have different variabilities. Besides a `constant` or a `parameter`, a variable can also be `discrete`. Since boolean and integer variables have a piecewise continuous solutions in any case, the `discrete`

keyword is intended to be used for real variables. Real variables are considered as continuous unless they are declared as `discrete`. An assignment to a variable inside a when clause has the same effect as declaring the variable as `discrete`. Discrete variables may not be mixed with continuous expressions. To avoid problems arising from the mixture of continuous and discrete variables, when clauses are not used in the generated Modelica model.

When time-discrete behaviour is modelled, the sample operator is used in conjunction with an if or when clause. It produces a periodical boolean pulse beginning at a specified time instant.

`sample(`*starttime*`,` *sampleperiod*`)`

In this way, the statements inside the if or when clause are executed in regular intervals. With the sample operator, periodical algorithms can be implemented or, like in the following example, a continuous signal can be sampled.

```
when sample(0, 0.01) then
  sampled_signal := continuous_signal;
end when;
```

Chapter 4

Code generation overview

The code generator is implemented in the JGrafchart editor. The menu "Execute" contains the item "Modelica", which opens a dialog to enter the filename for the Modelica model. The code is generated by traversing the JGrafchart model and the generated Modelica model is saved under the given name.

The function `modelicaAction`, which is started by the menu item, is part of the class `Editor`. First a filename to store the Modelica model in is obtained through a dialog window. After the model has been compiled and checked by the function `checkDocument` for structures that would lead to a non-working Modelica model, the document is coded by calling the function `codedocument`.

When the JGrafchart model is compiled, a list of all elements is created. In the function `codedocument` this list is traversed and code is generated for every element according to its properties. For elements that contain themselves documents, like e.g. macrosteps, `codedocument` is called recursively. The generated code is then printed into the specified file.

There are several smaller functions that are called by `codedocument` or other functions. The most important is probably `drawmodel`. It builds a purely graphical model for hierarchical elements like macrosteps in such a way that the inner documents can be displayed in the Modelica diagram. Figure 4.1 gives an overview of the structure of the code generator. Recursive function calls are not marked. Table 4.1 gives a short description of the functions in Figure 4.1.

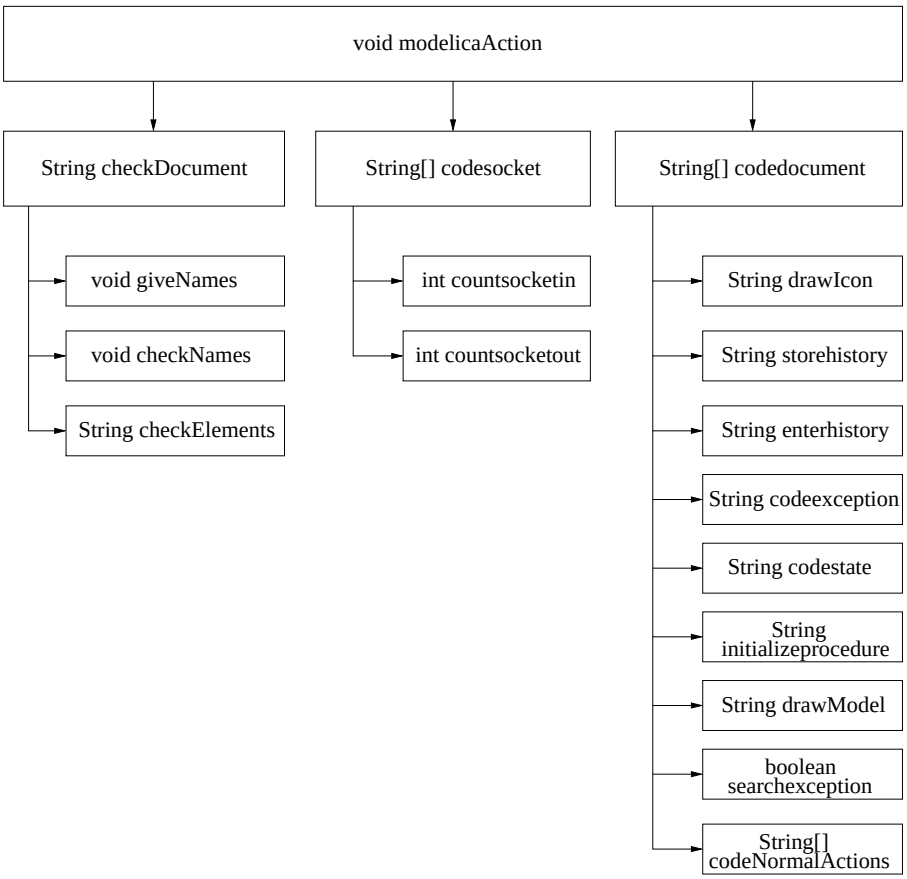

Figure 4.1: Function structure of the code generator

4.1 The generated Modelica program

As a state machine is no typical application for Modelica, the generated program differs from a typical Modelica model. First of all, the model does not contain an equation section, but an algorithm section. The algorithm imitates exactly the behaviour of the JGrafchart execution model.

All the hierarchies in JGrafchart are translated into flat code, except for the graphical models. These have, however, no contribution to the functionality of the model. To avoid name conflicts, all elements receive in Modelica a name that contains the global reference in JGrafchart without the top level workspace. The different levels are seperated by an underscore instead of a dot. For example, the step `step1` inside the macrostep `macrostep1` will get the name `macrostep1_step1` in Modelica. Most of the functions of the code generator have a string with the name of the actual workspace as input

Function	Description
checkDocument	Calls several other functions in order to check the model for non-translatable elements
giveNames	Generates names for nameless steps
checkNames	Checks the model for name conflicts
checkElements	Checks the model for non-translatable elements
codesocket	Writes annotations for connector terminals and declares connectors
countsocketin	Gives back the number of SocketIn elements in the model
countsocketout	Gives back the number of SocketOut elements in the model
codedocument	Writes the main part of the Modelica program
drawIcon	Writes code to place an icon in the Modelica diagram
storehistory	Declares variables for the abortion of a macrostep
enterhistory	Writes code to enter a macrostep through its history port
codeexception	Writes code for the abortion of a macrostep
codestate	Writes code to update the state of a macrostep
initializeprocedure	Writes code to assign the initial values to variables inside a procedure if the procedure is left
drawModel	Generates a graphical model for hierarchical elements
searchexception	Checks if there is an exception transition connected to the exception outport of a macrostep
codeNormalActions	Declares variables and writes code for the execution of normal actions

Table 4.1: Function overview

value. Thus, the right names can be generated in recursive function calls by adding the element name to the workspace string.

Dymola does not implement string variables. Without the possibility to specify the procedure name by a string variable, a process or procedure step can only have one corresponding procedure. Procedure calls are inlined and procedure or process steps are directly associated with a model of their corresponding procedure.

In order to facilitate the communication between the generated model and another Modelica model, connectors can be generated. This is done

using SocketIn and SocketOut elements, which lose their original purpose in the code generation.

The generated Modelica program consists mainly of two parts. The declaration part contains general declarations of different records corresponding to the JGrafchart elements and declarations of the actual elements. In the algorithm section the JGrafchart execution sequence is periodically executed with the JGrafchart scan cycle.

The following lines give an overview over the Modelica program structure:

```
model example
protected
  declaration of element types (records)
  declaration of the actual elements the model contains
public
  declaration of scan cycle
  declaration of parameters and variables
algorithm
  if initial() then
    entry actions of initial steps
    normal actions of initial steps
  end if;
  if sample(scancycle, scancycle) then
    update transition conditions
    fire transitions
    update time functions for steps
    periodic actions
    normal actions
    update state of steps
  end if;
  graphical annotation for icon and diagram
end example;
```

The different element types and the actual elements are declared as protected, so that they can only be accessed from inside the model. In JGrafchart it is possible to read the timer of a step from outside a model. This possibility is abandoned in Modelica. Variables and parameters are public, so that they can be read and modified from outside the model like in JGrafchart.

Conventional Modelica models are continuous. To get a time-discrete model, the function `sample(starttime, period)` is used. It produces a periodical boolean pulse starting at a specified time. In conjunction with an if clause it encloses the whole algorithm.

In JGrafchart, the initial steps are entered and their stored and normal actions are executed in the first scan cycle. Since initial steps are already active at the simulation start in Modelica, the periodic algorithm may only begin in the second scan cycle in Modelica. Therefore the starting time of the `sample()` function is not "0" but "scan cycle", the second scan. In Modelica the entry and normal actions of the initial steps are encircled by an if clause. They are executed when `initial()` is true, at the instant the simulation starts.

Compared to the original JGrafchart algorithm, the state of the steps is updated later in Modelica. This is necessary because the execution of periodic and normal actions is depending of the old value of the state. How this works exactly will be explained in the context of code generation for steps. However, this does not affect the results of periodic and normal actions, as the new value is nevertheless available for them.

While the JGrafchart model is traversed during code generation, different pieces of code have to be added at specific locations. In order to maintain the program structure, the code is stored in a string array. The elements of the array correspond to the different parts of the Modelica program structure. The structure can like this easily be assured while traversing the object list and in recursive function calls. Together with some declaration parts, that every generated model contains, the elements of the string array are printed into the Modelica file. Table 4.2 shows the different string elements of the program.

4.2 Error prevention during code generation

Before the JGrafchart model is translated into Modelica, it is checked for constructs that would lead to a non-working Modelica model. For this purpose the function `checkDocument` calls several other functions that look for different sources of error.

In contrary to JGrafchart, Modelica requires a name for every element of a model. Therefore the function `giveNames` traverses the model looking for nameless elements. Objects without name receive an automatically generated name.

In the case that a name appears several times in the same document, JGrafchart only gives out a warning. In Modelica this would lead to an error. The function `checkNames` looks for names that appear several times or are reserved in Modelica (e.g. "time"). If the function detects a name conflict, it gives out a warning and the code generation will not be executed.

Finally, the function `checkElements` is called. It checks the model for

intro	Declaration of records, connectors, partial models for diagram
code[0]	Declarations of models for macrosteps, workspaces etc.; private declarations of elements that model contains
code[1]	Public declarations; scan cycle, variables, parameters
code[2]	Initial actions: entry actions of initial steps
code[3]	Initial actions: normal actions of initial steps
code[4]	Update of transition conditions
code[5]	If clauses for transitions
code[6]	Update of time functions for steps
code[7]	Periodic actions
code[8]	Normal actions
code[9]	Assign nextstate to state
code[10]	Annotation for diagram (links, parallel split/join)
conclusion	Annotation for icon, close model

Table 4.2: String composition of the Modelica model

non-translatable objects. It also checks specific translatable objects, e.g. the procedure in a process step may not be specified by a string variable. If a detected fault would lead to an error in Modelica, an error message appears and the code generation is stopped. In the case that a feature is found which cannot be translated into Modelica but will not lead to an error, only a warning is given out and the feature is ignored during code generation.

Chapter 5

Code generation for JGrafchart elements

5.1 Steps

In Modelica a step is represented by a record with three variables. There are two boolean variables for its state, one real variable for the timer methods (`step.s` etc.) and a graphical annotation for the icon. The graphical annotation is not mentioned in the representation of the records below. Because of the different start values and the different icon, initial steps have their own record.

```
record step
  Boolean state;
  Boolean nextstate;
  Real active;
end step;
```

```
record initstep
  Boolean state(start=true);
  Boolean nextstate(start=true);
  Real active;
end initstep;
```

The two variables for the state are necessary to decide if the state has been changed in the actual scan cycle or the step has been in that state for a longer time. This is necessary for periodic and normal actions. Therefore the connected transitions assign a new value to `nextstate` if they fire, and not until the end of the algorithm `state` is updated. Consequently periodic and normal actions can be executed and the time function of the step updated according to the two booleans.

The records are declared at the beginning of the Modelica model in the string `intro` that every generated file contains. Every step of the JGrafchart model is translated into an instance of these records.

In JGrafchart a syntax tree is attached to every step. The nodes of the tree contain the different actions. There are different classes of nodes corresponding to the possible action language. During code generation, this structure is traversed in order to code the different actions. The function `traverseActionTree` in the class `GCStep` calls the function `traverse` of the topmost node of the tree. This function is declared for every node. Each node calls `traverse` of its child nodes until the bottom nodes are reached. The function returns a string at each time. The return values of the child nodes are combined according to the properties of the actual node. Thus the Modelica code for an action is pieced together while the action tree is traversed.

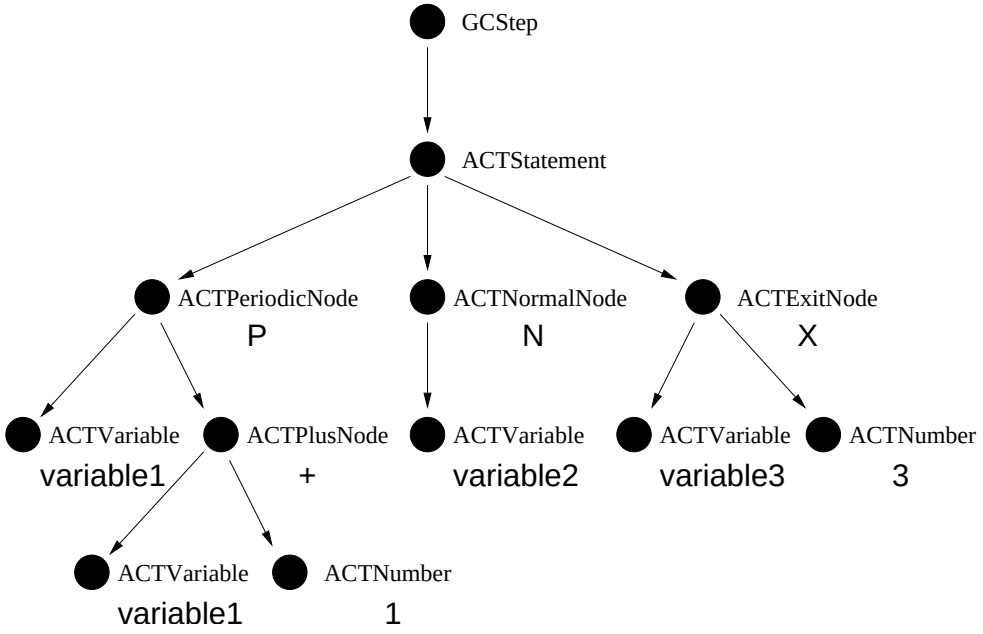

Figure 5.1: Action syntax tree

The tree above illustrates a step that has one periodic, one normal and one exit action. In the diagram the classes of the nodes and the corresponding action elements are indicated. In the action language the actions would look like this:

```
P variable1 = variable1 + 1;
N variable2;
X variable3 = 3;
```

When the function `traverseActionTree` is called, a string is passed on that specifies the kind of action that should be coded, e.g. the function

returns only code for the periodic actions, and disregards all other actions. In this way it is possible to write the actions at different places in the Modelica program according to their action qualifier. Enter, exit and abort actions are executed when the corresponding transition is fired. Periodic and normal actions are inserted in an if clause depending on the state of the step.

For periodic actions both `state` and `nextstate` have to be true since in JGrafchart periodic actions are not executed in the scan cycle the step becomes active or inactive. In that case the two variables would have different values.

```
if step1.state and step1.nextstate then
  variable1 := variable1 + 1;
end if;
```

Normal actions are executed in the scan cycle the state of a step changes. In the algorithm section they are executed before the variable `state` is updated, so that `state` and `nextstate` are unequal if the state has changed in the actual scan cycle. If several steps execute normal actions on the same variable in the same scan cycle, in JGrafchart an action which sets the variable to true will have priority over one that sets it to false. In Modelica, the last executed assignment decides the value of a variable. Since the calculation order of normal actions in the generated Modelica model cannot assure the priority of active steps, an auxiliary variable for every variable included in a normal action has to be implemented. If a variable is set to true, the variable `variable_setHigh` is also set to true. Normal actions on a variable are only executed, if `variable_setHigh` is false. After the execution of all normal actions, the auxiliary variables are set to false again. In this way, a normal action cannot set a variable to false which has been set to true in the same scan cycle by another normal action. The code for the normal actions is generated by traversing the syntax tree, the initialization and declaration of the auxiliary variables is generated by calling the function `codeNormalActions` in `codedocument` before returning the generated program to `modelicaAction`. During code generation, the variables included in a normal action have been marked, `codeNormalActions` only generates code for these variables.

```
if step1.state <> step1.nextstate then
  if not(variable2_setHigh) then
    variable2 := step1.nextstate;
    variable2_setHigh := step1.nextstate;
  end if;
```

```
end if;

variable2_setHigh := false;
```

In the context of the normal actions of initial steps in the first scan cycle, the auxiliary variables and if clauses are not necessary, the corresponding variables are just set to true.

Also the timer of the step is updated according to the two state variables. If the step is and has been active it is increased, else it is assigned "0".

```
step1.active := if (step1.state and step1.nextstate) then
  step1.active + scancycle else 0;
```

After the timer update and the actions `state` is finally updated.

```
step1.state := step1.nextstate;
```

Besides normal steps, also macrosteps, procedure and process steps have a state and a timer. They can as well have actions. These properties of macrosteps and procedure steps are coded in the same way as in the case of a normal step.

5.2 Transitions

Also transitions are represented by a record in Modelica. The record associates the boolean condition with the transition icon. The label is needed for the graphical annotation. In the declaration of the record it is an empty string, but when the instances of the individual transitions are declared, the JGrafchart labels are used. The transition condition in the label may differ from Modelica notation, e.g. instead of the boolean value "true" the JGrafchart notation "1" is used. As transitions have no name in JGrafchart but need one in Modelica, names are automatically generated. The name is composed of "transition" and the value of a counter.

```
record transition
  Boolean condition;
  parameter String label="";
end transition;
```

A transition in JGrafchart has a tree for its condition similar to what a step has for its actions. Also the different classes of the nodes correspond to

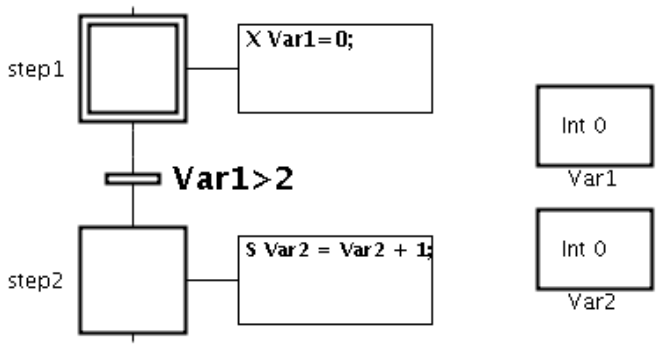

Figure 5.2: Example transition

those of the action tree. So the reading of the transition condition works in the same way as coding actions.

In the beginning of the Modelica algorithm the transition conditions are updated. The condition in Modelica does not exactly correspond to the condition in JGrafchart. In Modelica the states of the preceding steps are also considered in the condition. The following example shows the generated code for the transition in Figure 5.2.

```
transition1.condition := (Var1 > 2) and step1.state;
```

The firing of the transitions is implemented in an if clause. Inside the if clause `nextstate` is updated for the preceding and succeeding steps and the entry and exit actions of these steps are executed.

```
if transition1.condition then
  step1.nextstate := false;
  Var1 := 0; \\exit action step1
  step2.nextstate := true;
  Var2 := Var2 + 1; \\entry action step2
end if;
```

A JGrafchart transition has two attached lists, one for the preceding steps and one for the succeeding steps. During code generation the two lists are read and for every step in the list code is written. For a preceding step, `state` is added to the transition condition, `nextstate` is set to false and its exit actions are traversed. For a succeeding step `nextstate` is set to true and the entry actions are traversed.

A transition can only have one direct connection at each port. Nevertheless the lists can contain more than one item. If parallel splits or joins are used, all the steps connected over the split(s) or join(s) are written into the list when the model is compiled. Thus it is not necessary to generate code for parallel splits and joins, that is done automatically when the transitions are coded. For parallel splits and joins only a graphical annotation in the Modelica diagram is generated.

Also in the case of connection posts the code generation works without implementing them due to the lists.

Transitions connected to macrosteps, procedure steps or process steps need special handling. This will be discussed in the context of these elements.

5.3 Variables

JGrafchart and Modelica have the same four basic variable types: integer, real, boolean and string. In both languages these types have the same default start values. So the variables can be translated directly. Variables are declared public in Modelica to maintain the public structure of JGrafchart models.

A JGrafchart variable has an attribute `initialValue`. During code generation this attribute is checked and if necessary a start value is specified in the Modelica model. Otherwise the default start values are used. The two simulation environments handle initialization at the simulation start in different ways. In JGrafchart a variable without initial value keeps the last assigned value even in a new simulation run, in Modelica it is always initialized at the beginning of a simulation.

Modelica needs an exactly determined system of equations and assignments. Therefore every declared variable has to be assigned a value at least once in the algorithm section. That can be avoided by declaring the variable as parameter. In JGrafchart the difference between modifiable variables and parameters is not important. The attribute `isConstant()` of a variable can be set true, so that the value cannot be modified any more during simulation. But it is not necessary to do that, a JGrafchart model will also be able to execute with unused variables. When code for variables is generated, those for which `isConstant()` is true are declared as parameters. Hence it is very important that the programmer pays attention to the difference between variables and parameters already in JGrafchart, otherwise the Modelica model will not be executable.

As in Dymola string variables are not implemented, only string parameters are translated into Modelica code. If string variables are used in the

JGrafchart model, an error message will appear during the code generation when the model is checked for unallowed elements.

In the action or condition tree variables are represented by the class `ACTVariable` resp. `TRVar`. Through the attribute `out` the actual variable can be accessed and the name generated by reading the complete path of the variable.

5.4 Example

The code generation explained in the previous sections is illustrated with the example network shown in Figure 5.3. The declarations of the element records have been left out in the Modelica text. Besides steps and transitions, the example implements the different kinds of actions and parallel splits (see `transition1`) and joins (see `transition3`).

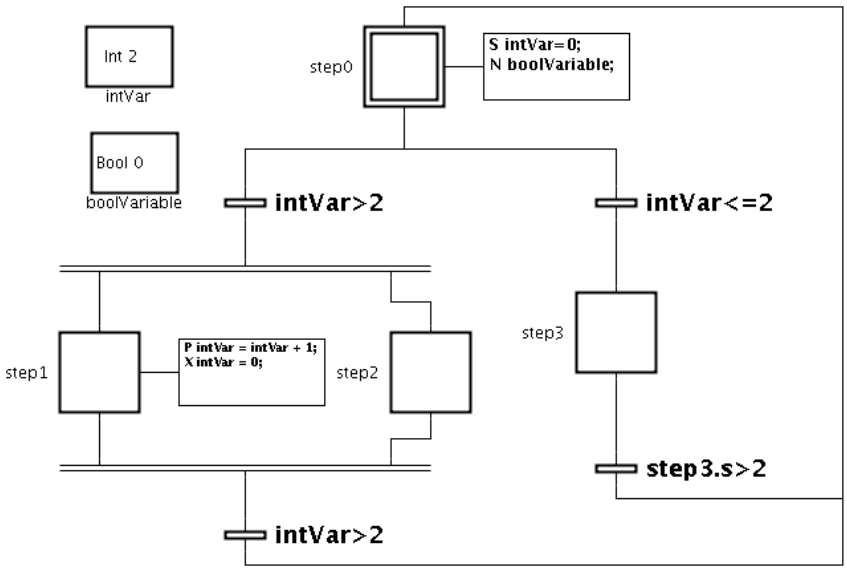

Figure 5.3: Example network in JGrafchart

```
model Example1
protected
  initstep step0;
  transition transition1(label="intVar>2");
  transition transition2(label="intVar<=2");
```

```
  step step3;
  step step1;
  step step2;
  transition transition3(label="intVar>2");
  transition transition4(label="step3.s>2");
  Boolean boolVariable_setHigh;

public
  parameter Real scancycle=0.04;
  Boolean boolVariable;
  Integer intVar(start=2);

algorithm

  if initial() then
    intVar := 0;
    boolVariable := true;
  end if;

  if sample(scancycle, scancycle) then

    transition1.condition = (intVar > 2) and step0.state;
    transition2.condition = (intVar <= 2) and step0.state;
    transition3.condition = (intVar > 2) and step1.state and
      step2.state;
    transition4.condition = (step3.active > 2) and step3.state;

    if transition1.condition then
      step0.nextstate := false;
      step1.nextstate := true;
      step2.nextstate := true;
    end if;
    if transition2.condition then
      step0.nextstate := false;
      step3.nextstate := true;
    end if;
    if transition3.condition then
      intVar := 0;
      step1.nextstate := false;
      step2.nextstate := false;
      intVar := 0;
```

```
      step0.nextstate := true;
    end if;
    if transition4.condition then
      step3.nextstate := false;
      intVar := 0;
      step0.nextstate := true;
    end if;

    step0.active := if (step0.state and step0.nextstate) then
      step0.active + scancycle else 0;
    step3.active := if (step3.state and step3.nextstate) then
      step3.active + scancycle else 0;
    step1.active := if (step1.state and step1.nextstate) then
      step1.active + scancycle else 0;
    step2.active := if (step2.state and step2.nextstate) then
      step2.active + scancycle else 0;

    if step1.state and step1.nextstate then
      intVar := intVar + 1;
    end if;

    if step0.state <> step0.nextstate then
      if not(boolVariable_setHigh) then
        boolVariable := step0.nextstate;
        boolVariable_setHigh := step0.nextstate;
      end if;
    end if;
    boolVariable_setHigh := false;

    step0.state := step0.nextstate;
    step3.state := step3.nextstate;
    step1.state := step1.nextstate;
    step2.state := step2.nextstate;

  end if;

end Example1;
```

5.5 Workspaces

A workspace is only a repository for a sublevel model, the element itself does not interfere with the rest of the model. The Modelica model does not follow the hierarchies, therefore workspace objects are only present in a purely graphical form in Modelica. For workspace objects the function `codedocument` is called recursively in order to code the sublevel document. As the names in Modelica contain the whole path, the workspace name is present in the names of the sublevel elements.

In JGrafchart it is possible to set a specific scan cycle for a workspace or to disable the firing of transitions. These features are not implemented in the Modelica program.

5.6 Macrosteps

A macrostep is both a step and a hierarchical element that contains a sub-workspace. In Modelica this hierarchy is not maintained, the Modelica macrostep is similar to a normal step and linked to the sublevel elements via the transition if clauses.

Like a normal step a macrostep is represented by a record. Only the name and the graphical annotation (which is not shown in the following representation) are different.

```
record macrostep
  Boolean state;
  Boolean nextstate;
  Real active;
end transition;
```

The periodical and normal actions are coded in the same way as those of steps. Also the updating of the timer and the variable `state` corresponds to normal steps.

Since a macrostep can have several entries and exits, its sublevel document can contain more than one active step at a time. So, if a macrostep is left through one exit, all the inner steps have to be checked to decide if `nextstate` has to be set true or false. Instead of controlling this for every transition connected to an exit, `nextstate` is assigned after all transition if clauses. The following example illustrates this for the macrostep of Figure 5.4.

```
mstep.nextstate := mstep_enterstep_1.nextstate or
  mstep_enterstep_2.nextstate or mstep_exitstep_1.nextstate or
  mstep_exitstep_2.nextstate;
```

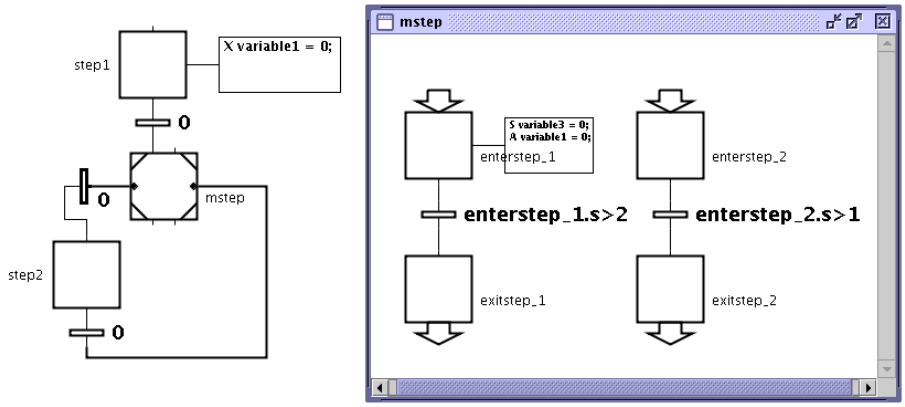

Figure 5.4: Example macrostep

The inner document is coded in the same way as the top level document, i.e., `codedocument` is called recursively. In the Modelica model the inner steps are at the same level as the macrostep and all other elements, only their name indicates the hierarchy in JGrafchart.

The link between the inner elements and the macrostep is made in the transition if clauses and the transition conditions. For every enter or exit step code is written in the if clause of the transition connected to the corresponding port. The enter or exit actions of the inner steps are added as well as their `nextstate` is updated. The order in which the actions are executed is the same as in JGrafchart. The entry actions of the macrostep are executed before those of the enter step, and the exit actions of the exit step before those of the macrostep. The state of the corresponding exit step is added to the transition condition. In the example below `enterstep_1` in Figure 5.4 is entered.

```
if transition1.condition then
  variable1 := 0; \\exit action of preceding step
  step1.nextstate := false;
  variable2 := 2; \\entry action of macrostep
  mstep.nextstate := true;
  variable3 := 0; \\entry action of enter step
  mstep_enterstep_1.nextstate := true;
end if;
```

When the JGrafchart model is compiled, the enter and exit steps are also written in the lists of succeeding and preceding steps of the transitions. Thus they can easily be accessed and coded.

Also the exception outport and the history port are implemented by adding code to the transition if clause.

The name generated for the exception transition does not conform with that of normal transitions, it is composed of the name of the macrostep and "_exception".

The exception transition has priority over all inner transitions. In the Modelica program this is achieved by updating the inner transition conditions only if the exception transition condition is false. Normally new code is added in the end of a string, but the code for the update of the exception transition is added in the beginning of `code[4]`, so that it is updated before all other conditions of the same document. The conditions of the inner transitions are enclosed by an if clause, so that they can only be set true if the exception transition will not fire in this scan cycle.

```
if not (mstep_exception.condition) then
  mstep_transition1.condition := (mstep_enterstep_1.active >2)
    and mstep_enterstep_1.state;
  mstep_transition2.condition := (mstep_enterstep_2.active >1)
    and mstep_enterstep_2.state;
end if;
```

In the context of the abortion of a macrostep some additional variables have to be declared. In order to recall the state of a macrostep through the historyport after an abortion, a boolean variable is declared for each inner step. In these the actual states of the steps are stored when the macrostep is aborted. This is done during the code generation for the macrostep. If the function `searchexception` finds an exception transition connected to the macrostep and returns the value "true", the function `storehistory` is called. This function generates code to declare a private boolean variable for every inner step. The generated name is composed of the complete name of the step and "_storedstate".

A macrostep can only be recalled after it has been aborted. Therefore another boolean variable is needed that is set true after the first abortion. Its name is composed of the name of the macrostep and "_aborted".

The abortion is performed in the if clause of the exception transition. This if clause contains an if clause for each inner step. Depending on the actual state of the inner step different operations are executed. In both cases the actual state is stored and `nextstate` set to false. If the step is active, its abort actions are executed.

After the if clauses for the inner steps the macrostep itself is aborted. The abort actions are executed, `nextstate` is set to false and `macrostep_aborted` to true.

Then the succeeding step of the exception transition is entered. For the list of succeeding steps of an exception transition the same code is generated as for a normal transition. The following example shows the if clause of the exception transition in Figure 5.4.

```
if mstep_exception.condition then
  if mstep_enterstep_1.state then
    mstep_enterstep_1_storedstate := true;
    variable1 := 0; \\abort action of enterstep_1
    mstep_enterstep_1.nextstate := false;
  else
    mstep_enterstep_1_storedstate := false;
    mstep_enterstep_1.nextstate := false;
  end if;
  if mstep_exitstep_1.state then
    ...
  if mstep_enterstep_2.state then
    ...
  if mstep_exitstep_2.state then
    ...
  variable2 := true; \\abort action of macrostep
  mstep.nextstate := false;
  mstep_aborted := true;
  step2.nextstate := true;
end if;
```

In contrary to the exception outport, the history port is connected to a normal transition. Its if clause contains the operations that are executed when a macrostep is recalled. For each inner step `nextstate` is updated according to the stored state and for the steps that become active the entry actions are executed. If the macrostep has not been aborted before, it will not be activated. The preceding step is left, but no other step is activated subsequently. In the following example the macrostep of Figure 5.4 is resumed.

```
if transition2.condition then
  step2.nextstate := false; \\preceding step of transition is
    left
  if mstep_aborted then
    variable1 := 1; \\entry action of macrostep
    mstep.nextstate := true;
```

```
    mstep_enterstep_1.nextstate :=
      mstep_enterstep_1_storedstate;
    if mstep_enterstep_1.nextstate then
      variable2 := 0; \\enter action enterstep_1
    end if;
    mstep_exitstep_1.nextstate :=
      mstep_exitstep_1_storedstate;
    ...
    mstep_enterstep_2.nextstate :=
      mstep_enterstep_2_storedstate;
    mstep_exitstep_2.nextstate :=
      mstep_exitstep_2_storedstate;
    ...
  end if;
end if;
```

If a transition is connected to the history port of a macrostep, its list of succeeding steps contains the history outport and the macrostep. The history port is the first element in this list. The boolean variable `historyport` is set "true" if a history port is found in the list. Thus it can be detected when a macrostep is found in the list if the transition is connected to its history port. Then the function `enterhistory` writes the code for the recall of the macrostep and after that `historyport` is reset.

The following example network `Example2` implements a macrostep with an exception transition and a transition connected to its history inport. The corresponding JGrafchart model is shown in Figure 5.5. The record declarations are not included in the Modelica text.

```
model Example2
protected
  initstep step0;
  macrostep mstep;
  Boolean mstep_aborted;
  Boolean mstep_exit_storedstate;
  Boolean mstep_enter_storedstate;
  step mstep_exit;
  step mstep_enter;
  transition mstep_transition1(label="enter.s>1");
  transition transition1(label="step0.s>1");
  transition transition2(label=''mstep.exit.s>1");
  exceptiontransition mstep_exception(label="Var1>10");
```

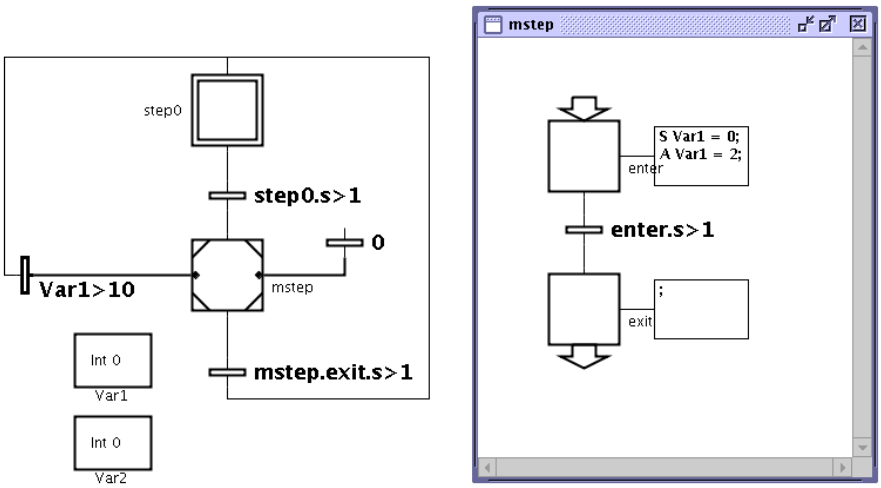

Figure 5.5: Example network in JGrafchart

```
  transition transition3(label="0");

public
  parameter Real scancycle=0.04;
  Integer Var1;
  Integer Var2;

algorithm

  if sample(scancycle, scancycle) then

    mstep_exception.condition = (Var1 > 10) and mstep.state;
    if not (mstep_exception.condition) then
      mstep_transition1.condition = (mstep_enter.active > 1)
        and mstep_enter.state;
    end if;
    transition1.condition = (step0.active > 1) and step0.state;
    transition2.condition = (mstep_exit.active > 1) and
      mstep_exit.state and mstep.state;
    transition3.condition = false;

    if mstep_transition1.condition then
      mstep_enter.nextstate := false;
```

```
  mstep_exit.nextstate := true;
end if;
if transition1.condition then
  step0.nextstate := false;
  Var2 := Var2 + 1;
  mstep.nextstate := true;
  Var1 := 0;
  mstep_enter.nextstate := true;
end if;transition2.condition then
  mstep_exit.nextstate := false;
  step0.nextstate := true;
end if;
if mstep_exception.condition then
  if mstep_exit.state then
    mstep_exit_storedstate := true;
    mstep_exit.nextstate := false;
  else
    mstep_exit_storedstate := false;
    mstep_exit.nextstate := false;
  end if;
  if mstep_enter.state then
    mstep_enter_storedstate := true;
    Var1 := 2;
    mstep_enter.nextstate := false;
  else
    mstep_enter_storedstate := false;
    mstep_enter.nextstate := false;
  end if;
  Var2 := 0; \\abort action mstep
  mstep.nextstate := false;
  mstep_aborted = true;
  step0.nextstate := true;
end if;
if transition3.condition then
  if mstep_aborted then
    Var2 := Var2 + 1;
    mstep.nextstate := true;
    mstep_exit.nextstate := mstep_exit_storedstate;
    mstep_enter.nextstate := mstep_enter_storedstate;
    if mstep_enter.nextstate then
      Var1 := 0;
```

```
        end if;
      end if;
    end if;

    mstep.nextstate := mstep_exit.nextstate or
      mstep_enter.nextstate;

    step0.active := if (step0.state and step0.nextstate) then
      step0.active + scancycle else 0;
    mstep.active := if (mstep.state and mstep.nextstate) then
      mstep.active + scancycle else 0;
    mstep_exit.active := if (mstep_exit.state and
      mstep_exit.nextstate) then
      mstep_exit.active + scancycle else 0;
    mstep_enter.active := if (mstep_enter.state and
      mstep_enter.nextstate) then
      mstep_enter.active + scancycle else 0;

    step0.state := step0.nextstate;
    mstep.state := mstep.nextstate;
    mstep_exit.state := mstep_exit.nextstate;
    mstep_enter.state := mstep_enter.nextstate;

  end if;

end Example2;
```

5.7 Procedures

In JGrafchart, procedure and process steps are able to call different procedures during a simulation run. This is possible if the procedure is not indicated directly but through a string variable. As Dymola does not support string variables and procedure calls are rather complicated to implement in Modelica , procedure calls are inlined. The resulting structure in Modelica is then similar to that of a macrostep.

A procedure has no directly corresponding object in Modelica. For every procedure or process step its corresponding procedure is identified. The model inside this procedure is then translated into Modelica and linked to the procedure or process step. So the model of a procedure can appear more than once in the Modelica model – or not at all if the procedure is not called.

For this reason the principle of naming objects with the full JGrafchart path cannot be maintained here. It could lead to name conflicts and confusion, as the inlined elements would not include the name of the procedure step but that of the procedure. So the procedure name in the path is exchanged with the name of the procedure or process step.

To most of the functions of the code generator a string with the actual path is handed over when they are called. With the aid of this string the object names for Modelica are generated. When a function is called recursively, the name of the actual object (e.g a macrostep) is added to the path. In the case of procedure or process steps, their name is added when their corresponding procedure is coded. In this way the right path is always available.

Yet another case has to be considered. When the action or condition tree is traversed and coded, the actual path does not always correspond to the path of a variable or other object included in the expression. With the method `getFullName()`, that gives back the complete path of an object, a correct name for Modelica can only be obtained if the object is not inside a procedure. Though one cannot refer from the outside to variables inside a procedure, procedures can contain hierarchical elements. Thus it is possible that a procedure contains e.g. a macrostep and that a transition condition inside the macrostep refers to a variable outside the macrostep, but inside the procedure. Figure 5.6 illustrates this case.

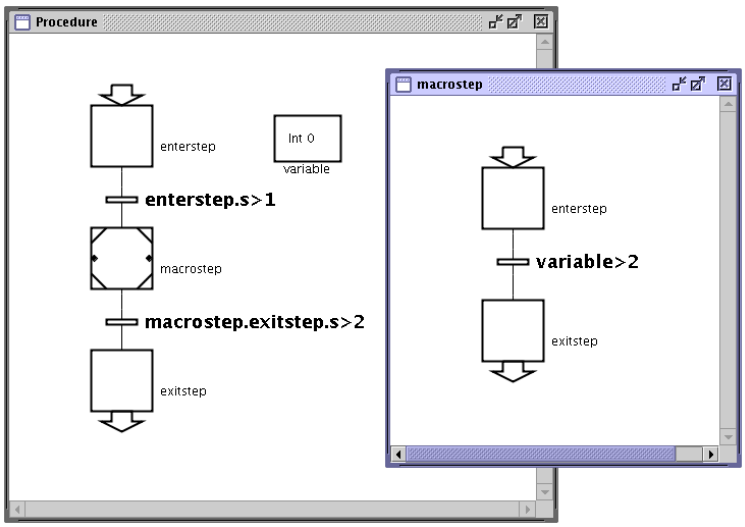

Figure 5.6: Illustration of path problem

Consequently a new function `getFullName()` is needed, that exchanges the procedure name against the name of the actual procedure step in the path. The method `getFullName()` traverses starting from the considered object up to the toplevel workspace the whole path and writes the names of the different hierarchical objects into a string. This function can easily be adopted in checking all traversed objects on being a procedure. In that case the name of the actual procedure step is added to the string instead of the procedure name. Instead of declaring a new method for each JGrafchart object, the code is added to the method `traverse` in the classes that represent objects in the action or condition tree. This is the case for `ACTVariable`, `ACTFunction`, `TRVar` and `TRFunction`. In the class `GrafcetProcedure` a new string variable `currentpstep` is added. When a procedure step is coded, its name is assigned to `currentstep` of its corresponding procedure. Thus, the correct object name can be accessed through the procedure.

Like steps for actions, procedure and process steps have an attached syntax tree for the procedure call. The procedure step has two nodes, `procnode` and `paramnode`. To the first the name of the procedure is attached, to the second the parameters and the way they are called. Before the procedure and the parameters can be read, the procedure or process step has to be compiled.

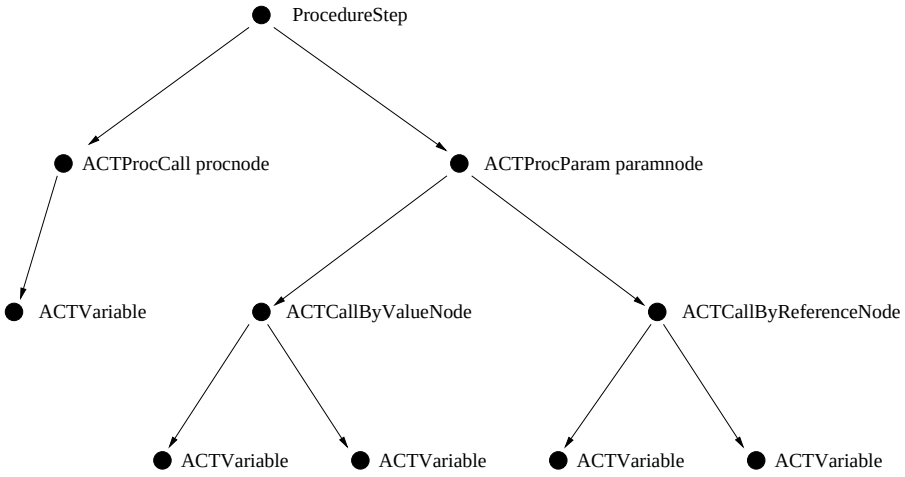

Figure 5.7: Nodes of a procedure step

Then the procedure can be accessed and its document coded in a recursive call of `codedocument` in the same way as in the case of the inner document of a macrostep. As procedure and process steps also belong to the class `MacroStep`, their code generation is implemented in that of macrosteps. The

common features like actions, timer and exception outport are translated in the same way. As procedures cannot have several entries or exits, `nextstate` of a procedure or process step can be assigned in the corresponding transition if clause instead of appointing it after the transition if clauses.

Apart from the procedure parameters, a procedure step is translated in the same way as a macrostep. In the way a process step is deactivated, it differs from macrosteps and procedure steps. Therefore some differences appear in the transitions inside the procedure and connected to the process step. The condition of the succeeding transition of the process step does only depend on the state of the process step, it does not depend of the state of the exit state of the procedure document. In the if clause of the transition connected to the exit step, the exit step is not activated, only its exit actions are executed.

When a procedure or process step with a "CallByReferenceNode" is compiled, a link between two variables is created. A JGrafchart variable has an attribute `redirect`. During compilation, the outer variable is assigned to `redirect` of the inner variable. In the Modelica program, the inner variable does not exist, it is replaced with the outer variable in all actions and conditions. Therefore, when code is generated for variables, `redirect` of the corresponding variable is checked first. If it is not "null", no declaration is generated for the variable and it is replaced with the outer variable when generating code for actions or transition conditions.

Each time a copy of a procedure is entered, the inner variables take their inital values if they are not parameters called by value or by reference. A procedure has no memory of past procedure calls, whereas a Modelica variable keeps the last assigned value. Therefore the inlined procedure call in Modelica has to be initialized before it is entered again. As the call by value has priority over the assignment of the initial value, all procedure variables are assigned their initial values or the standard initial value when the procedure step is left. When the procedure step is entered later, the call by value assignments are made. This is done in the transition if clause of the transition preceding the procedure or process step. The assignments are coded in the same way as actions, by traversing the syntax tree.

The code for the initialization of the procedure document is generated in calling the function `initializeprocedure`. It generates an assignment for every variable that is not redirected because of a call by reference. For the initialization the difference between procedure and process steps has to be taken into account. In the case of a procedure step, the procedure call is definitely finished when the procedure step is left. Thus the initialization can take place in the transition if clause of the transition succeeding the procedure step. For a process step this cannot be done, as the proce-

dure might still be active when the process step is left. So the initialization is made after the exit actions of the procedure's exit step. Consequently `initializeprocedure` is called at different locations depending on the type of step. For procedure steps it is called if a procedure step is found in the list of preceding steps of a transition, for process steps if an exit step is found in the list of succeeding steps and the boolean variable `processstep` indicates that the actual document will be inlined in a process step.

The following example (see Figure 5.8) shows a network with a procedure step. Both "call by value" and "call by reference" are implemented (see object dialog in Figure 5.9). The declaration of the element records are left out in the Modelica text.

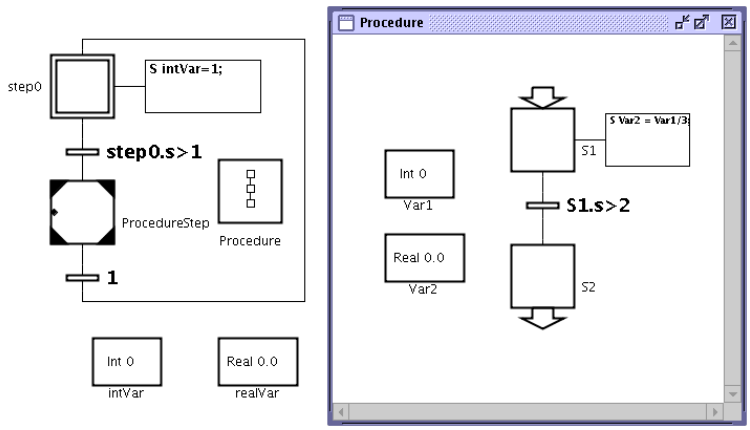

Figure 5.8: Example network in JGrafchart

```
model Example3
protected
  initstep step0;
  transition transition1(label="step0.s>1");
  macrostep ProcedureStep;
  step ProcedureStep_S2;
  step ProcedureStep_S1;
  transition ProcedureStep_transition1(label="S1.s>2");
  transition transition2(label="1");

public
  parameter Real scancycle=0.04;
  Integer ProcedureStep_Var1;
```

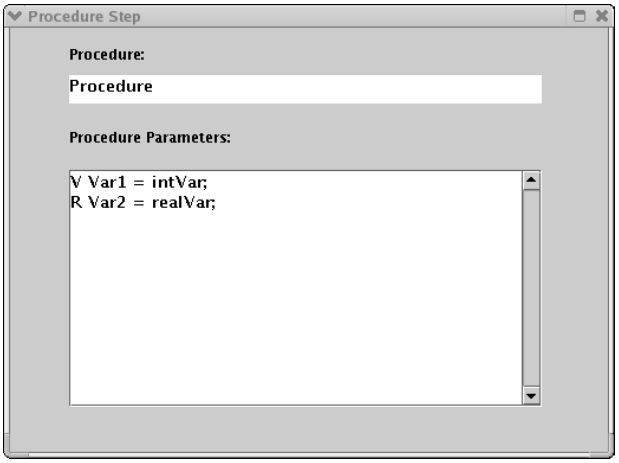

Figure 5.9: Part of the procedure step's object dialog

```
  Integer intVar;
  Real realVar;

algorithm

  if initial() then
    intVar := 1;
  end if;

  if sample(scancycle, scancycle) then

    transition1.condition = (step0.active > 1) and step0.state;
    ProcedureStep_transition1.condition =
      (ProcedureStep_S1.active > 2) and ProcedureStep_S1.state;
    transition2.condition = true and ProcedureStep_S2.state
      and ProcedureStep.state;

    if transition1.condition then
      step0.nextstate := false;
      ProcedureStep_Var1 := intVar;
      ProcedureStep.nextstate := true;
      realVar := (ProcedureStep_Var1)/(3);
      ProcedureStep_S1.nextstate := true;
    end if;
    if ProcedureStep_transition1.condition then
```

```
      ProcedureStep_S1.nextstate := false;
      ProcedureStep_S2.nextstate := true;
    end if;
    if transition2.condition then
      ProcedureStep_S2.nextstate := false;
      ProcedureStep_Var1 := 0;
      ProcedureStep.nextstate := false;
      intVar := 1;
      step0.nextstate := true;
    end if;

    step0.active := if (step0.state and step0.nextstate) then
      step0.active + scancycle else 0;
    ProcedureStep.active := if (ProcedureStep.state and
      ProcedureStep.nextstate) then
      ProcedureStep.active + scancycle else 0;
    ProcedureStep_S2.active := if (ProcedureStep_S2.state and
      ProcedureStep_S2.nextstate) then
      ProcedureStep_S2.active + scancycle else 0;
    ProcedureStep_S1.active := if (ProcedureStep_S1.state and
      ProcedureStep_S1.nextstate) then
      ProcedureStep_S1.active + scancycle else 0;

    step0.state := step0.nextstate;
    ProcedureStep.state := ProcedureStep.nextstate;
    ProcedureStep_S2.state := ProcedureStep_S2.nextstate;
    ProcedureStep_S1.state := ProcedureStep_S1.nextstate;

  end if;

end Example3;
```

5.8 Connectors

In Modelica models with connectors can easily be connected to other models without detailed knowledge of the internals of the models. Connectors are also implemented in the code generator, so that the generated model can be used in Modelica without knowledge of the generated model structure.

As in JGrafchart no similar elements are available, SocketIn and SocketOut elements are used to generate connectors. They cannot be translated

into Modelica according to their original JGrafchart purpose and TCP communication may therefore not be used in a model that will be translated. Furthermore, a SocketIn and a SocketOut element are available for each variable type.

Hence, the connectors in Modelica are declared in analogy with the different SocketIn and SocketOut elements. For each variable type except string variables an output and an input connector are defined. The following lines show the declaration of the boolean output connector. Additionally, every connector contains a graphical icon annotation.

```
connector SocketBoolOut
  ouput Boolean signal;
end SocketBoolOut;
```

The string `intro` in the beginning of the Modelica program contains the six possible connector classes. The function `codesocket` declares the connectors the model contains. They are declared as public, as they have to be accessible from other models.

In actions and transition conditions SocketIn and SocketOut elements are treated like ordinary variables. In JGrafchart, a reference to a SocketIn or SocketOut element consists of the element name. In Modelica, this name refers to the connector, but not to the signal inside the connector. Therefore, when translating actions and conditions to Modelica, "`.signal`" has to be added to the element name.

Chapter 6

Graphics and animation in Modelica

6.1 The icon and the diagram

Every Modelica model can have an icon and a diagram. The diagram shows the sublevel composition of the model. The icon represents the model in upper level models. For both, a common coordinate system is declared. To avoid coordinate conversions, the dimensions of the top level workspace in JGrafchart are used for the Modelica coordinate system. Nevertheless the JGrafchart coodinates cannot be used directly, as the direction of the vertical axis is different in the programs. In order to create a quadratic icon, the greater of the workspace dimensions is used. The model icon consists of a rectangle with the name of the model inside. Figure 6.1 shows an example of an icon as it appears in an upper level diagram. The following graphical annotation shows the declaration of the coordinate system and the icon.

```
annotation (
  Coordsys(extent=[0, -806; 806, 0]),
  Icon(Rectangle(extent=[20, -786; 786, -20], style(color=0,
    thickness=4)),
      Text(
        extent=[50, -756; 756, -50],
        style(color=0, thickness=3),
        string = "%name")));
```

The coordinate system and the icon are generated in `modelicaAction` and written into the strings `code[10]` and `conclusion` respectively.

To enable graphical connections between the generated model and other Modelica models, the connectors have an icon. An icon is placed in the diagram by specifying its extent coordinates in a graphical annotation. If this is done for a connector, the icon will also appear in the model icon. A graphical connection can be made by clicking with the mouse on the connector icon in the model icon and drawing the mouse to another connector icon. The complete declaration of the boolean output connector is:

```
connector SocketBoolOut
  output Boolean signal;
  annotation (
    Coordsys(extent=[-100, -100; 100, 100]),
    Icon(Polygon(points=[60, 0; -60, -80; -60, 80],
      style(color=0, fillColor=0)), Text(
        extent=[-100, 200; 100, 100],
        string="%name",
        style(color=0, thickness=3))))
end SocketBoolOut;
```

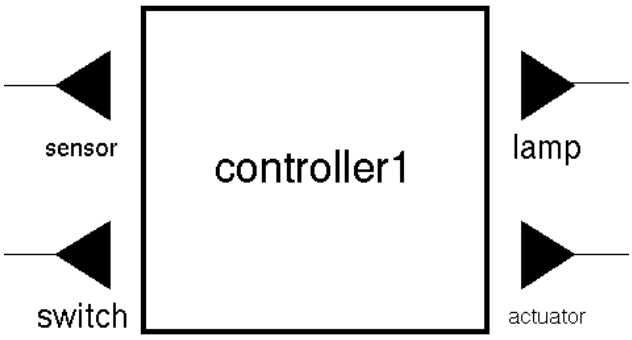

Figure 6.1: Model icon with connectors

The graphical annotations for connectors are generated by the function `codesocket`. The input connectors are placed at the left side of the model icon, the output connectors at the right one. In order to place the connectors in regular intervals, they are counted by the functions `countsocketin` and `countsocketout`. Thus, a convenient distance between the connector icons can be fixed. Then the document is traversed and code for the connector declarations and graphical annotations is generated. The connector icons are placed one by one along the sides of the model icon from top to bottom.

The code generator uses two variables to store the actual vertical position of the last declared connector on the left and right side. After a connector has been generated, the position is updated and increased by the connector distance. In order to maintain the positions up-to-date in recursive function calls, the positions have to be handed over to and given back by `codesocket`. Therefore, `codesocket` gives back a string array with the generated code as well as the positions of the last declared connectors. An example for a model icon with connectors can be seen in Figure 6.1.

In order to illustrate the generated program for the Modelica user, a diagram is generated in Modelica that shows the network from JGrafchart. The network elements are represented by icons. The layout of the Modelica icons is based on the JGrafchart icons. In the case of non-hierarchical elements like steps and transitions, the icon is defined in the declaration of the object classes at the beginning of the Modelica model. When the instances of these elements are declared, the icons are placed in the diagram with a graphcial annotation that specifies the coordinate extent of the icon. The following lines show the declaration text of a step including the object icon and the declaration of an instance of a step in Modelica.

```
record step
  Boolean state;
  Boolean nextstate;
  Real active;
  annotation (Icon(
      Rectangle(extent=[-80, 80; 80, -80], style(
          color=0,
          thickness=2,
          fillColor=7)),
      Line(points=[0, 80; 0, 115], style(color=0)),
      Line(points=[0, -80; 0, -87], style(color=0)),
      Text(
          extent=[-300, 20; -100, -20],
          string="%name",
          style(color=0))));
end step;
...
step step2 annotation(extent=[45, -600; 115, -530]);
```

The icons of hierarchical elements are declared and placed in a similar way. The graphical representation of hierarchical elements is explained more in detail in Section 6.2.

The graphical annotations to place step instances in the diagram are generated when steps and initial steps are translated in `codedocument`. Then the function `drawIcon` is called. This function indentifies the coordinates of the object in the diagram and the size of the icon and generates the code for the graphical annotation. The coordinates of the JGrafchart diagram are converted by changing the sign of the vertical axis. In some cases, e.g. for steps, it also has to be taken into account that the coordinates do not refer to the center of the object icon. Therefore the coordinates have to be adjusted. With the aid of `getHeight()` or `getWidth()` the dimensions of the JGrafchart icon can be identified. When translating the icon size into Modelica, it has to be considered that different object classes have a different relation between the invisible bounding rectangle of an icon given by `getHeigth()` and `getWidth()` and the actual visible icon. For example, without considering this different relation, enter and exit steps would appear larger in the Modelica diagram as ordinary steps.

For transitions and exception transitions the function `drawIcon` is not called, though the procedure is the same. As the object classes for transitions and exception transitions do not inherit the class `Referencable`, which the function uses, the code for the graphical annotation is generated directly in `codedocument`.

The parts of the JGrafchart network that have no corresponding Modelica element, like the links between steps and transitions or parallel splits and joins, are modelled by drawing lines in the diagram. The code for the lines is integrated in the diagram part of the graphical annotation at the end of the model that also contains the model icon and the coordinate system. The generated code is written in the string `code[10]`.

```
annotation (
  Coordsys( ...),
  Diagram(
    Line( ...),
    ...),
  Icon( ...));
```

The connections between the different objects in the JGrafchart diagram are represented by the object class `GCLink`. An attached list of points defines the graphical connection. If such an object is found in the object list of the top level document, the points are read and their coordinates translated to Modelica. The following line shows an example of a graphical annotation for a link. As one can see, the line is defined by more points than are actually necessary. The JGrafchart link includes points in regular distances and during code generation all points are translated.

```
Line(points=[160, -335; 160, -340; 160, -345], style(color=0))
```

Also parallel splits and joins lack corresponding objects in Modelica. For them two parallel lines are generated that imitate their JGrafchart icon. If a parallel split or join is found in the object list of the top level document, its coordinates and extension are read and according to them the distance, the length and the coordinates of the lines are generated. The following example shows a graphical annotation for a parallel split or join.

```
Line(points=[180, -217; 460, -217],
   style(color=0, thickness=2)),
Line(points=[180, -225; 460, -225],
   style(color=0, thickness=2))
```

6.2 The model for hierarchical objects

The JGrafchart model is translated into flat Modelica code. Without any further code generation, the JGrafchart hierarchies would only be visible in the object names that contain their complete path in JGrafchart. The Modelica diagram would only show the top level document although it is possible to show the diagrams of sublevel models in Modelica. In order to show also the lower level documents, a purely graphical model is generated for every hierarchical element. This is also done for procedure or process steps, though they do not have a sublevel document in JGrafchart. The graphical model shows then the procedure document.

This model exists in parallel with the functional objects that are generated when a hierarchical object is translated into flat code. The top level model contains the declarations of the graphical model class as well as the declaration of an instance of this model with a graphical annotation that specifies its coordinates in the top level diagram. The graphical model is generated by calling the function `drawModel` when a hierarchical element is translated and the code for it stored in `code[0]`. The name of the model class is composed of the complete path of the object beginning in the top level document and "_model", whereas the instance of this model is named with "_g" attached to the complete object name. The graphical annotation that places the model icon in the top level diagram is generated with the function `drawIcon`.

As for the top level document, an icon and a coordinate system are declared for the graphical model. In the function `drawModel` the dimensions of the inner document are read and according to them a coordinate system is

created in the same way as for the top level document. Then an icon is generated corresponding to the object icon in JGrafchart. The graphical model includes also a string parameter `label` which contains the name without the path of the object. This string is integrated in the model icon.

The steps and transitions the graphical model contains are represented by records. The difference compared to the functional records of these elements is that the graphical ones contain no variables, they only aggregate an icon and a string parameter for the name or transition condition in a class. In addition to the record `step_dummy` shown in the example below, similar records for initial steps, transitions and exception transitions are declared in the general declaration part in the beginning of the top level model.

```
record step_dummy
  paramter String label="";
  annotation (Icon(
      Rectangle(extent=[-80, 80; 80, -80], style(
          color=0,
          thickness=2,
          fillColor=7)),
      Line(points=[0, 80; 0, 115], style(color=0)),
      Line(points=[0, -80; 0, -87], style(color=0)),
      Text(extent=[-300, 20; -100, -20],
          string="%label",
          style(color=0))));
end step_dummy;
```

In the function `drawModel` the object list of the sublevel document is traversed and for every step or transition an instance of the corresponding graphical element is declared. A graphical annotation for the icon extent in the diagram is generated by calling the function `drawIcon`.

Parallel splits and joins and links between the objects are handled in the same way as in the top level document. They are modelled by lines that are added to the graphical annotation of the model diagram.

If a hierarchical object contains another hierarchical object, the function `drawModel` is called recursively and the lower level model class is inserted in the upper level model as well as an instance of the declared model class. Consequently, the function `codedocument` calls `drawModel` only for hierarchical objects in the top level document. The graphical models of lower level hierarchical elements are generated through recursion of `drawModel`.

The following example illustrates the generated graphical model. Figure 6.2 shows the example macrostep containing an enter and an exit step and a transition.

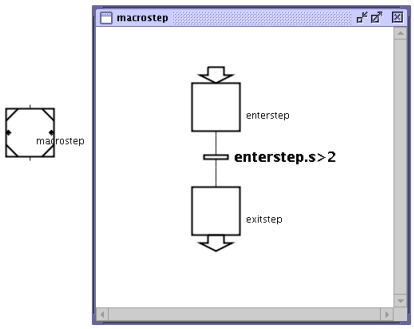

Figure 6.2: Example macrostep

```
model macrostep_model
  parameter String label="macrostep";
  annotation (Coordsys(extent=[0, -400; 400, 0]), Icon(
      Rectangle(extent=[20, -380; 380, -20], style(
          color=0,
          thickness=2)),
      Line(points=[110, -20; 20, -110],
          style(color=0, thickness=2)),
      Line(points=[290, -380; 380, -290],
          style(color=0, thickness=2)),
      Line(points=[290, -20; 380, -110],
          style(color=0, thickness=2)),
      Line(points=[20, -290; 110, -380],
          style(color=0, thickness=2)),
      Ellipse(extent=[20, -190; 40, -210], style(
          color=0,
          thickness=4,
          fillColor=0)),
      Ellipse(extent=[360, -190; 380, -210], style(
          color=0,
          thickness=4,
          fillColor=0)),
      Text(extent=[400, -200; 800, -290],
          string="%label",
          style(color=0))));
  step_dummy macrostep_exitstep_m(label="exitstep")
    annotation (extent=[116, -269; 184, -201]);
```

```
  step_dummy macrostep_enterstep_m(label="enterstep")
    annotation (extent=[116, -139; 184, -71]);
  transition_dummy macrostep_t1(label="enterstep.s>2")
    annotation (extent=[125, -185; 175, -135]);
  annotation (Diagram(line(points=[150, -1135; 150, -145;
    -150, -145; 150, -140; 150, -140; 150, -150],
    style(color=0)), Line(points=[150, -175; 150, -185;
    150, -185; 150, -185; 150, -185; 150, -195],
    style(color=0))));
end macrostep_model;
```

6.3 Animation

Modelica diagrams can be animated during realtime simulation [9]. This possibility is also implemented in the generated Modelica model.

To enable animation, a graphical annotation can contain dynamical parts. For example, the fill color of a rectangle or the coordinates of a line can depend on a variable. In order to make a part of a graphical annotation dynamical, the function `DynamicSelect` is needed. It gives back the value of an attribute according to a dynamic expression. Also a static value has to be specified.

```
attribute = DynamicSelect(static value, dynamic expression)
```

This feature can be used to animate the generated network in such a way that the active steps show a token like in JGrafchart. To the icon of steps and other objects that have a state an ellipse that represents the token is added. The color of the ellipse depends of the state of the corresponding object. If the element is deactivated, the token is white (color code 7) and thus invisible on the white background. Transparency corresponds to the lack of the fill color attribute and can therefore not be used to make the token invisible. If the step is activated, the token changes its color to black (color code 0) and gets visible. The static value for the token is white. The following example shows the declaration of the token as it is integrated in the record icon.

```
Ellipse(extent=[-20, -20; 20, 20], style(color=7,
  fillColor=DynamicSelect(7, if state > 0.5 then 0 else 7)))
```

In the same way the color of transitions can be changed between red (color code 1) and green (color code 2) according to their condition.

For steps and transitions their state or condition can easily be obtained as icon and variable belong to the same record. For hierarchical elements that is not the case. The in the diagram represented graphical element is not the same as the functional element that contains the variable `state`. As animation did not work with a reference from the annotation in the graphical model to the variable of the functional model, the token of hierarchical elements is included in the functional record. The record declaration including the icon is the following, the icon only consists of the token.

```
record macrostep
  Boolean state;
  Boolean nextstate;
  Real active;
  annotation(Icon(
    Ellipse(extent=[-20, -20; 20, 20],
      style(color=7, fillColor=
      DynamicSelect(7, if state > 0.5 then 0 else 7)))));
end macrostep;
```

The icons of functional and graphical elements are placed one upon the other with `drawIcon`. When the object is activated, the token is visible through the transparent icon of the graphical object. This is only possible for hierarchical elements in the top level document, because the corresponding objects have to be at the same hierarchy level in order to appear in the same diagram.

For this reason, it is not possible to animate the graphical sublevel steps and transitions in this way. Joining the parallel existing objects to one object and generating a functional hierarchical model is not compatible with the idea to generate one Modelica model, because it is not possible to declare a hierarchical model containing variables inside another Modelica model. Therefore, animation is at the moment only implemented for the top level document.

Chapter 7

Example: A controlled tank system

7.1 The tank system

In this example the code generator is used to generate a controller for a tank system. The tank system itself is modelled in Modelica. It consists of two tanks that are connected in series (see Figure 7.1). The upper tank can be filled from a reservoir and emptied into the lower one, the lower one can be emptied through a drain. The filling and emptying can be controlled by three on-off valves.

With the tank system a simple process is executed. The upper tank is filled from the reservoir up to a specified limit. Then the reservoir is shut and the valve between the two tanks is opened. When the liquid has completely flowed into the second tank, it is emptied through the drain. After a tank has been filled, the process pauses for a short time.

The discrete-event controller implements this process in an infinite loop. In order to start and stop the process, the tank system includes also three buttons (see Figure 7.1). With "start", the process is started. With "stop", the process can be interrupted. After a process has been halted, it can be resumed with "start" or the plant can be shutdown with "shut". At most one button is active at the same time, clicking on one button forces the other ones to deactivate.

The three valves are opened or shut by the controller. The tanks also include level sensors, that send back their signals to the controller.

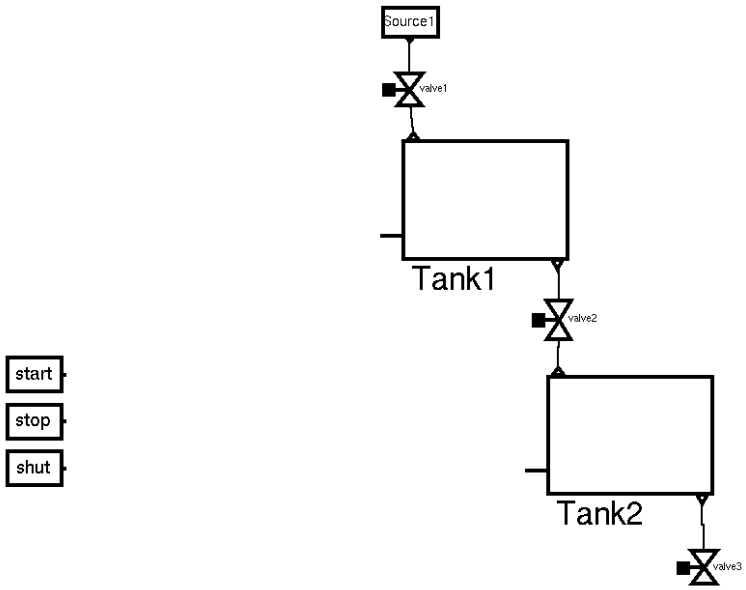

Figure 7.1: Tank system in Modelica

7.2 The JGrafchart controller

Figure 7.2 shows the top level workspace of the JGrafchart controller. The starting point of the process is the initial step `s_1`. All the valves are closed and the process is waiting for the "Start" signal. If "Start" is set to true, the macrostep `MakeProduct` is activated and one cycle of the process is executed. When the process cycle is ended, the token reenters `s_1` and a new process cycle begins.

The infinite loop can be interrupted by setting "Stop" to true. Then the exception transition of `MakeProduct` is fired and the process is aborted. The abort actions of `MakeProduct` are executed, i.e. the three valves are closed. In `s_2`, the controller waits for a new signal. Either the process can be resumed by setting "Start" to true or the plant can be shutdown by setting "Shutdown" to true. In the case the process is resumed, `MakeProduct` is entered through the history port and the process is continued. If the plant is shutdown, the tanks are emptied and after that the initial step `s_1` is entered. As "Start" has been deactivated by setting "stop" to true, the controller waits for a new signal in `s_1`.

In order to communicate with the process, the controller needs different input and output variables. The tank level signals are received by two SocketRealIn objects and the three button signals by three SocketBoolIn objects.

The valves are connected to three SocketBoolOut objects.

Figure 7.3 shows the macrostep `MakeProduct` which contains the actual process. First the upper tank is filled. For that valve1 is opened and after a certain level, `limit`, has been reached, it is closed again. After a short waiting time, valve2 is opened and the lower tank is filled until the upper tank is empty. Then valve2 is shut again. It is assumed that the lower tank is not smaller than the upper one. After another waiting time, the lower tank is emptied by opening valve3. The macrostep is left when the lower tank is empty.

7.3 The generated Modelica controller

The complete code of the generated Modelica model is shown in Appendix A. In the first part of the model, the different object classes are defined (Page 72). Then the graphical model for the macrostep follows (Page 76). The declaration of the object instances (Page 78) completes the declaration part. On Page 79, the algorithm section begins with the execution of the initial steps' entry actions. After that the if clause with the sampled algorithm follows. First the transition conditions are updated. The transition conditions inside the macrostep are only updated if the exception transition condition is false. Then the if clauses to fire the transitions follow. This section contains notably the if clause for the exception transition (Page 80) and for `transition2`, which is connected to the history port of the macrostep (Page 81). After the assignment for the macrostep's `nextstate`, the timer and `state` of each step is updated (Page 83). The model contains no periodic or normal actions. The graphical annotation for the diagram and icon is the last part of the model.

Figure 7.4 the tank system including the controller, Figure 7.5 shows the generated Modelica diagram.

7.4 Results

Figure 7.6 shows the simulation results of the tank system combined with the generated controller. In the beginning of the simulation, all valves are shut and the tanks are empty. After one second, "Start" is activated. The process starts and one cycle is completely executed. When tank 1 is emptied into tank 2 in the second cycle, "Stop" is activated. After some seconds, the process is resumed with "Start". When the process is stopped a second time, the plant is shutdown after some seconds and both tanks are emptied. The generated controller works in the intended way.

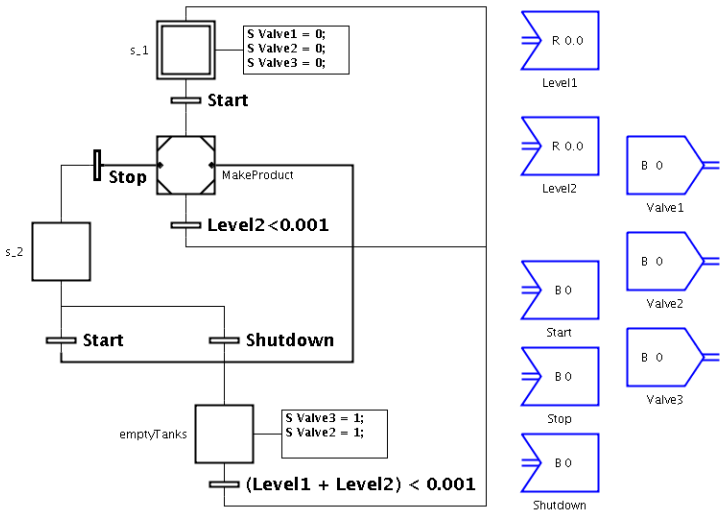

Figure 7.2: The tank controller in JGrafchart

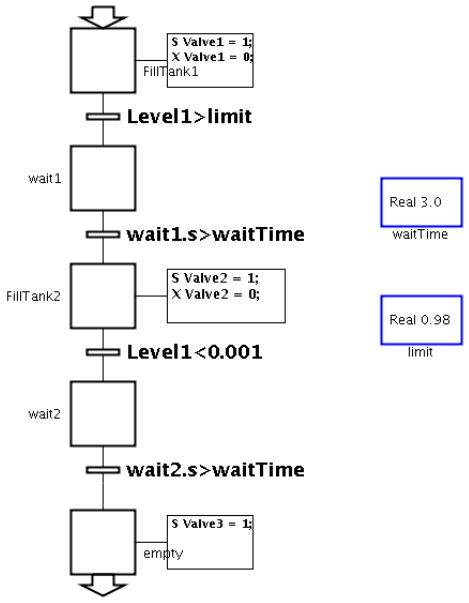

Figure 7.3: The fill and empty process

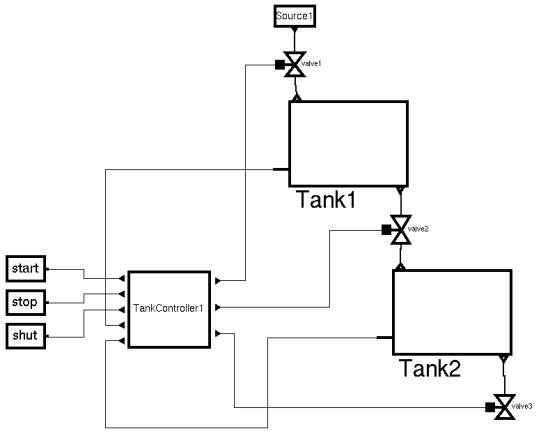

Figure 7.4: System with tanks and controller

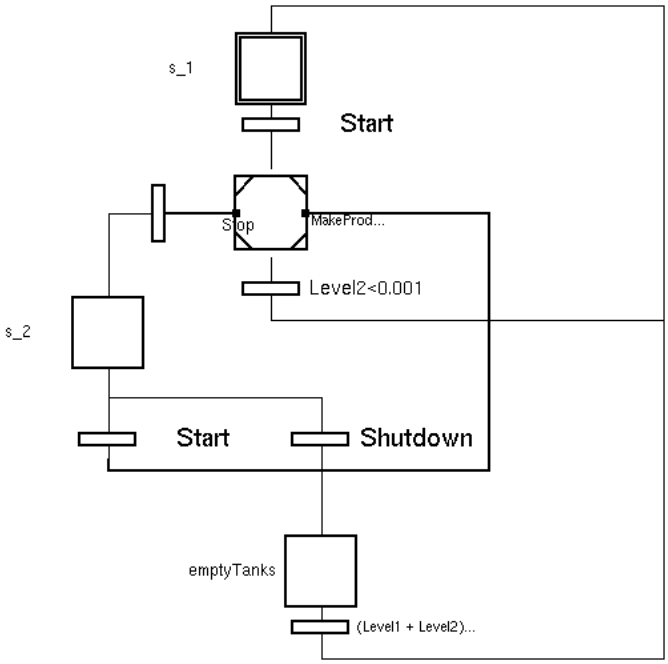

Figure 7.5: Diagram of the generated Modelica controller

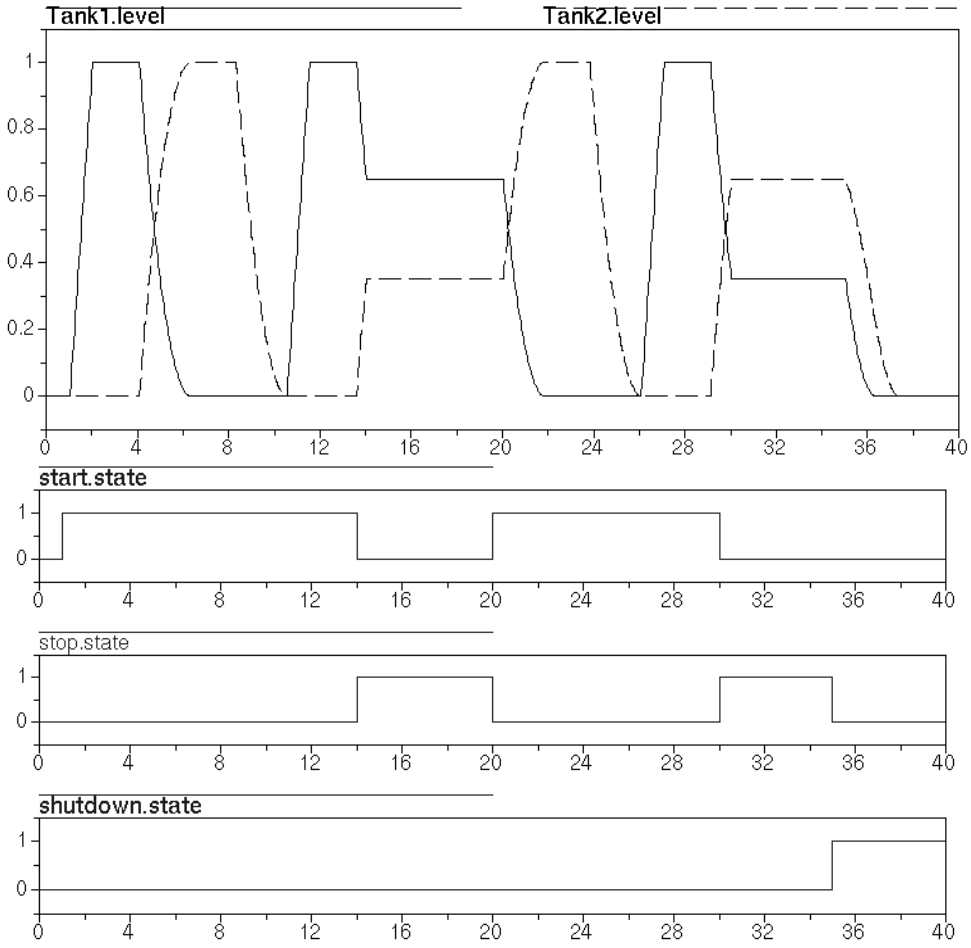

Figure 7.6: Simulation results

Chapter 8

Limitations and assumptions

Although the basic elements of JGrafchart are implemented in the code generator, some objects cannot be translated. Dymola has not the same possibilities to communicate with real processes as JGrafchart. Therefore, the communication part of JGrafchart cannot be translated into Modelica. This part includes digital or analog input or output channels and TCP and XML messages respectively.

Another part that has not been implemented in the code generator is the graphical user interface. Although some of its elements and methods are translatable into Modelica with the aid of dynamic or interactive graphical annotations, they do not play a major role in modelling discrete-event controllers and have therefore been left out.

JGrafchart lists are because of their flexible size impossible to translate. Whereas in JGrafchart elements can be added to or removed from a list and thus its size changed during simulation, in Modelica the size of an array is fixed and each element has to be assigned a value.

In the case a not implemented element is part of a network that is to be translated into Modelica, the code generator detects this when the function `checkElements` is called. Since the number of elements available in the JGrafchart Editor is limited, every non-translatable element can be taken into account in `checkElements` and errors in Modelica resulting from those can be avoided.

Besides complete elements that cannot be translated into Modelica, several elements include features which are not included in the code generator. For example, the scan cycle of a subworkspace cannot be changed, as this would be difficult to implement in the Modelica program structure. Likewise, the procedure may not be given by a string variable for a procedure or process step, as string variables are not supported by Dymola. The concerned elements are also checked by the function `checkElements` and non-working

Modelica models can thus be avoided.

However, non-working Modelica programs cannot be avoided on all accounts. During the design of the code generator, not all possible combinations of elements and their resulting Modelica model may have been considered, especially when hierarchical and more sophisticated elements are implemented. Therefore it may be possible that an unconsidered combination of elements leads to a non-working Modelica program.

Even though the JGrafchart model is checked by `checkDocument` before the actual code generation, not all known sources of error can be found in this way. JGrafchart handles variables more flexibly. A difference between parameters and variables is not necessary, though a variable can be assigned constant. It is also possible to mix different variable types in assignments like real and integer variables. On the other hand, Modelica is very strict concerning the treatment of variables. The different variable types cannot be mixed and for every variable which is not a parameter or a constant, at least one assignment in the algorithm section is required. Therefore every variable in JGrafchart which is not specified by an action has to be a constant. Also the different variable types may not be mixed in a model that is to be translated into Modelica. Since the code generator does not check variables in conjunction with the actions, the success of the code generation regarding these matters relies completely on the JGrafchart user.

Chapter 9

Conclusion

In this thesis, a code generator to translate JGrafchart models into Modelica language has been implemented. With the aid of the code generator, discrete-event controllers can be modelled in JGrafchart an then combined with a continuous Modelica model. The code generator includes the relevant JGrafchart objects for discrete-event modelling. The basic elements like steps, transitions and variables are translated as well as hierarchical and procedural constructions including e.g. macrosteps or procedures. During the code generation, the JGrafchart model is checked for non-translatable constructs in order to avoid a non-working Modelica model. In doing so, several sources of error can be detected, but a working program cannot be assured at any rate. For example, the distinction between variables and parameters, which is not important in JGrafchart but necessary in Modelica, relies completely on the JGrafchart user.

The Modelica program is based on an algorithm section in which the JGrafchart execution model is implemented. After the transition conditions are updated and the transitions fired that are to be fired, the actions are executed and the timer and the state of the steps updated. The algorithm is executed with the JGrafchart scan cycle and an identical model behaviour in both simulation environments can be assured. A negligible but inevitable difference may result from the fact that Modelica in contrast to JGrafchart considers that calculations do not take any time. In Modelica the results of the algorithm section are available at the same simulation time instant the execution of the algorithm starts, whereas JGrafchart is executed in real-time. The objects are represented by records which contain variables for the state and timer or the transition condition respectively. Hierarchical constructs are translated into flat code, dependencies between the hierarchical element and inner objects are implemented in the algorithm part of the corresponding transitions. Procedure calls are inlined in procedure or pro-

cess steps. In addition to the functional program part a diagram including realtime simulation animation is generated.

Further development of the code generator could include the improvement of the error prevention. For example, a function could be implemented that checks if every JGrafchart variable which is not constant is assigned a value by at least one action. Furthermore, possible combinations of hierarchical elements and their resulting Modelica model could be examined more extensively in order to identify and remove potential sources of error. In that way the success of the code generation could be ensured.

Another way to ensure the success of the code generation could be to have a special Modelica version of JGrafchart where the elements and features which are not translatable are not available. In that way errors are already prevented when the model is constructed.

Another point that could be implemented is animation of sublevel diagrams. Although the approach that has been tried out in this thesis did not work, other approaches could be studied. For example, the possiblity to add dynamic graphical annotations to the sublevel diagram that are not part of the graphical object icon could be investigated. Another way to animate sublevel models could be to generate a library that includes the Modelica object classes and a hierarchical model of the JGrafchart network. In the case of a hierarchical Modelica model the problem of the reference between functional and graphical objects does not occur and thus animation can be made possible.

A comparison between a flat and a hierarchical Modelica model and a Modelica library for JGrafchart elements could also be interesting in other contexts.

Furthermore, the possibilities of on-line communication between JGrafchart and Dymola could be studied. By implementing on-line communication, elements that are not translatable into Modelica might nevertheless be used if a part of the simulation is executed in JGrafchart.

Bibliography

[1] JGrafchart, On-Line Help of the JGrafchart Editor.

[2] Tiller M.M.: *Introduction to Physical Modeling with Modelica*, Boston, 2001.

[3] Modelica Homepage: http://www.Modelica.org/.

[4] Dymola Homepage: http://www.Dynasim.se/.

[5] Remelhe M.A.P.: *Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment*, 2nd International Modelica Conference, Proceedings, pp.203-207, 2002.

[6] Ferreira J.A., Estima de Oliveira J.P.: *Modelling Hybrid Systems using Statecharts and Modelica*, Proc. 7th IEEE International Conference on Emerging Technologies and Factory Automation, Barcelona, Spain, 1999.

[7] Mosterman P.J., Otter M., Elmqvist H.: *Modeling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica*, Proceedings of the Summer Computer Simulation Conference -98, Reno, USA, 1998.

[8] Remelhe M.A.P., Engell S.: *Structuring Discrete-Event Models in Modelica.*

[9] Elmqvist H., Olson H.: *Modelica Objects for User Interaction*, Realsim Deliverable Report P3, Lund, Sweden, 2002.

Appendix A

Code of the example controller

```
model TankController
protected
  record step
    Boolean state;
    Boolean nextstate;
    Real active;
    annotation (Icon(
      Rectangle(extent=[-80, 80; 80, -80],
        style(color=0, thickness=2, fillColor=7)),
      Line(points=[0, 80; 0, 115], style(color=0)),
      Line(points=[0, -80; 0, -87], style(color=0)),
      Ellipse(extent=[-20, -20; 20, 20],
        style(color=7, fillColor=
        DynamicSelect(7, if state > 0.5 then 0 else 7))),
      Text(extent=[-300, 20; -100, -20], string="%name",
        style(color=0))));
  end step;

  record initstep
    Boolean state(start=true);
    Boolean nextstate(start=true);
    Real active;
    annotation (Icon(
      Rectangle(extent=[-80, 80; 80, -80],
        style(color=0, thickness=2, fillColor=7)),
      Rectangle(extent=[-70, 70; 70, -70],
        style(color=0, thickness=2, fillColor=7)),
      Line(points=[0, 80; 0, 115], style(color=0)),
```

```
    Line(points=[0, -80; 0, -87], style(color=0)),
    Ellipse(extent=[-20, -20; 20, 20],
      style(color=7, fillColor=
      DynamicSelect(7, if state > 0.5 then 0 else 7))),
    Text(extent=[-300, 20; -100, -20], string="%name",
      style(color=0))));
end initstep;

record step_dummy
  parameter String label = "";
  annotation (Icon(
    Rectangle(extent=[-80, 80; 80, -80],
      style(color=0, thickness=2, fillColor=7)),
    Line(points=[0, 80; 0, 115], style(color=0)),
    Line(points=[0, -80; 0, -87], style(color=0)),
    Text(extent=[-300, 20; -100, -20], string="%label",
      style(color=0))));
end step_dummy;

record initstep_dummy
  parameter String label = "";
  annotation (Icon(
    Rectangle(extent=[-80, 80; 80, -80],
      style(color=0, thickness=2, fillColor=7)),
    Rectangle(extent=[-70, 70; 70, -70],
      style(color=0, thickness=2, fillColor=7)),
    Line(points=[0, 80; 0, 115], style(color=0)),
    Line(points=[0, -80; 0, -87], style(color=0)),
    Text(extent=[-300, 20; -100, -20], string="%label",
      style(color=0))));
end initstep_dummy;

record macrostep
  Boolean state;
  Boolean nextstate;
  Real active;
  annotation (Icon(Ellipse(extent=[-20, -20; 20, 20],
    style(color=7, fillColor=
    DynamicSelect(7, if state>0.5 then 0 else 7)))));
end macrostep;
```

```
record transition
  Boolean condition;
  parameter String label = "";
  annotation (Icon(
    Rectangle(extent=[-90, -20; 90, 20],
      style(color=0, thickness=2, gradient=0,
      fillColor=7, fillPattern=1)),
    Line(points=[0, 20; 0, 40], style(color=0)),
    Line(points=[0, -20; 0, -60], style(color=0)),
    Text(extent=[500, 30; 120, -30], string="%label",
      style(color=0))));
end transition;

record exceptiontransition
  Boolean condition;
  parameter String label = "";
  annotation (Icon(
    Rectangle(extent=[-20, -90; 20, 90],
      style(color=0, thickness=2)),
    Line(points=[-20, 0; -60, 0], style(color=0)),
    Line(points=[20, 0; 40, 0],
      style(color=0, thickness=2)),
    Text(extent=[20, -10; 500, -70], string = "%label",
      style(color=0))));
end exceptiontransition;

model transition_dummy
  parameter String label = "";
  annotation (Icon(
    Rectangle(extent=[-90, -20; 90, 20],
      style(color=0, thickness=2, gradient=0,
      fillColor=7, fillPattern=1)),
    Line(points=[0, 20; 0, 40], style(color=0)),
    Line(points=[0, -20; 0, -60], style(color=0)),
    Text(extent=[500, 30; 120, -30], string="%label",
      style(color=0))));
end transition_dummy;

model exception_dummy
  parameter String label = "";
  annotation (Icon(
```

```
    Rectangle(extent=[-20, -90; 20, 90],
      style(color=0, thickness=2)),
    Line(points=[-20, 0; -60, 0], style(color=0)),
    Line(points=[20, 0; 40, 0],
      style(color=0, thickness=2)),
    Text(extent=[20, -10; 500, -70], string = "%label",
      style(color=0))));
end exception_dummy;

connector SocketBoolIn
  input Boolean signal;
  annotation (Coordsys(extent=[-100, -100; 100, 100]),
    Icon(
      Polygon(points=[-60, 0; 60, -80; 60, 80],
        style(color=0, fillColor=0)),
      Text(extent=[-100, 200; 100, 100], string="%name",
        style(color=0, thickness=3))));
end SocketBoolIn;

Connector SocketIntIn
  input Integer signal;
  annotation (Coordsys(extent=[-100, -100; 100, 100]),
    Icon(
      Polygon(points=[-60, 0; 60, -80; 60, 80],
        style(color=0, fillColor=0)),
      Text(extent=[-100, 200; 100, 100], string="%name",
        style(color=0, thickness=3))));
end SocketIntIn;

connector SocketRealIn
  input Real signal;
  annotation (Coordsys(extent=[-100, -100; 100, 100]),
    Icon(
      Polygon(points=[-60, 0; 60, -80; 60, 80],
        style(color=0, fillColor=0)),
      Text(extent=[-100, 200; 100, 100], string="%name",
        style(color=0, thickness=3))));
end SocketRealIn;

connector SocketBoolOut
  output Boolean signal;
```

```
  annotation (Coordsys(extent=[-100, -100; 100, 100]),
    Icon(
      Polygon(points=[60, 0; -60, -80; -60, 80],
        style(color=0, fillColor=0)),
      Text(extent=[-100, 200; 100, 100], string="%name",
        style(color=0, thickness=3))));
end SocketBoolOut;

connector SocketIntOut
  output Integer signal;
  annotation (Coordsys(extent=[-100, -100; 100, 100]),
    Icon(
      Polygon(points=[60, 0; -60, -80; -60, 80],
        style(color=0, fillColor=0)),
      Text(extent=[-100, 200; 100, 100], string="%name",
        style(color=0, thickness=3))));
end SocketIntOut;

connector SocketRealOut
  output Real signal;
  annotation (Coordsys(extent=[-100, -100; 100, 100]),
    Icon(
      Polygon(points=[60, 0; -60, -80; -60, 80],
        style(color=0, fillColor=0)),
      Text(extent=[-100, 200; 100, 100], string="%name",
        style(color=0, thickness=3))));
end SocketRealOut;


model MakeProduct_model
  parameter String label = "MakeProduct";
  annotation (Coordsys(extent=[0, -671; 671, 0]),
    Icon(
      Rectangle(extent=[20, -651; 651, -20], style(color=0,
        thickness=2)),
      Line(points = [177, -20; 20, -177],
        style(color=0, thickness=2)),
      Line(points = [493, -651; 651, -493],
        style(color=0, thickness=2)),
      Line(points = [493, -20; 651, -177],
        style(color=0, thickness=2)),
```

```
    Line(points = [20, -493; 177, -651],
      style(color=0, thickness=2)),
    Ellipse(extent=[20, -318; 53, -352],
      style(color=0, thickness=4, fillColor=0)),
    Ellipse(extent=[617, -318; 651, -352],
      style(color=0, thickness=4, fillColor=0)),
    Text(extent=[671, -335; 1342, -493], string="%label",
      style(color=0))));
step_dummy MakeProduct_empty_m(label="empty")
  annotation (extent=[76, -549; 144, -481]);
step_dummy MakeProduct_FillTank1_m(label="FillTank1")
  annotation (extent=[76, -99; 144, -31]);
transition_dummy MakeProduct_t1(label="Level1>limit")
  annotation (extent=[85, -135; 135, -85]);
transition_dummy MakeProduct_t2(label="wait2.s>waitTime")
  annotation (extent=[85, -465; 135, -415]);
step_dummy MakeProduct_wait1_m(label="wait1")
  annotation (extent=[75, -210; 145, -140]);
transition_dummy MakeProduct_t3(label="wait1.s>waitTime")
  annotation (extent=[85, -245; 135, -195]);
transition_dummy MakeProduct_t4(label="Level1<0.001")
  annotation (extent=[85, -355; 135, -305]);
step_dummy MakeProduct_wait2_m(label="wait2")
  annotation (extent=[75, -430; 145, -360]);
step_dummy MakeProduct_FillTank2_m(label="FillTank2")
  annotation (extent=[75, -320; 145, -250]);
annotation(Diagram(
  Line(points=[110, -315; 110, -325; 110, -325; 110, -310;
    110, -310; 110, -320], style(color=0)),
  Line(points=[110, -425; 110, -435; 110, -435; 110, -420;
    110, -420; 110, -430], style(color=0)),
  Line(points=[110, -345; 110, -355; 110, -355; 110, -345;
    110, -345; 110, -355], style(color=0)),
  Line(points=[110, -205; 110, -215; 110, -215; 110, -200;
    110, -200; 110, -210], style(color=0)),
  Line(points=[110, -125; 110, -135; 110, -135; 110, -125;
    110, -125; 110, -135], style(color=0)),
  Line(points=[110, -455; 110, -465; 110, -465; 110, -465;
    110, -465; 110, -475], style(color=0)),
  Line(points=[110, -95; 110, -105; 110, -105; 110, -90;
    110, -90; 110, -100], style(color=0)),
```

```
      Line(points=[110, -235; 110, -245; 110, -245; 110, -235;
        110, -235; 110, -245], style(color=0))));
  end MakeProduct_model;

  initstep s_1 annotation (extent=[155, -160; 225, -90]);
  macrostep MakeProduct;
  Boolean MakeProduct_aborted;
  Boolean MakeProduct_empty_storedstate;
  Boolean MakeProduct_FillTank1_storedstate;
  Boolean MakeProduct_wait1_storedstate;
  Boolean MakeProduct_wait2_storedstate;
  Boolean MakeProduct_FillTank2_storedstate;
  MakeProduct_model MakeProduct_g
    annotation (extent=[159, -271; 221, -209]);
  step MakeProduct_empty;
  step MakeProduct_FillTank1;
  transition MakeProduct_transition1(label="Level1>limit");
  transition MakeProduct_transition2(label="wait2.s>waitTime");
  step MakeProduct_wait1;
  transition MakeProduct_transition3(label="wait1.s>waitTime");
  transition MakeProduct_transition4(label="Level1<0.001");
  step MakeProduct_wait2;
  step MakeProduct_FillTank2;
  transition transition1(label="Start")
    annotation (extent=[165, -195; 215, -145]);
  exceptiontransition MakeProduct_exception(label="Stop")
    annotation (extent=[75, -265; 125, -215]);
  step s_2 annotation (extent=[25, -370; 95, -300]);
  transition transition2(label="Start")
    annotation (extent=[35, -445; 85, -395]);
  transition transition3(label="Shutdown")
    annotation (extent=[205, -445; 255, -395]);
  step emptyTanks annotation (extent=[195, -520; 265, -450]);
  transition transition4(label="(Level1 + Level2) <0.001")
    annotation (extent=[205, -565; 255, -515]);
  transition transition5(label="Level2<0.001")
    annotation (extent=[165, -325; 215, -275]);

public
  parameter Real scancycle=0.04;
```

```
  parameter Real MakeProduct_waitTime(start=3);
  parameter Real MakeProduct_limit(start=0.98);
  SocketBoolIn Start
    annotation (extent=[-107, -43; -20, -130]);
  SocketBoolIn Stop
    annotation (extent=[-107, -218; -20, -305]);
  SocketBoolIn Shutdown
    annotation (extent=[-107, -393; -20, -480]);
  SocketRealIn Level1
    annotation (extent=[-107, -568; -20, -655]);
  SocketRealIn Level2
    annotation (extent=[-107, -743; -20, -830]);
  SocketBoolOut Valve1
    annotation (extent=[899, -73; 986, -160]);
  SocketBoolOut Valve2
    annotation (extent=[899, -366; 986, -453]);
  SocketBoolOut Valve3
    annotation (extent=[899, -659; 986, -746]);

algorithm

  when time >= 0 then
    Valve1.signal := false;
    Valve2.signal := false;
    Valve3.signal := false;
  end when;

  if sample(scancycle, scancycle) then

    MakeProduct_exception.condition = Stop.signal and
      MakeProduct.state;
    if not (MakeProduct_exception.condition) then
      MakeProduct_transition1.condition =(Level1.signal>
        MakeProduct_limit) and MakeProduct_FillTank1.state;
      MakeProduct_transition2.condition =
        (MakeProduct_wait2.active > MakeProduct_waitTime)
        and MakeProduct_wait2.state;
      MakeProduct_transition3.condition =
        (MakeProduct_wait1.active > MakeProduct_waitTime)
        and MakeProduct_wait1.state;
      MakeProduct_transition4.condition = (Level1.signal
```

```
    < 0.001) and MakeProduct_FillTank2.state;
end if;
transition1.condition = Start.signal and s_1.state;
transition2.condition = Start.signal and s_2.state;
transition3.condition = Shutdown.signal and s_2.state;
transition4.condition = ((Level1.signal + Level2.signal)
  < 0.001) and emptyTanks.state;
transition5.condition = (Level2.signal < 0.001) and
  MakeProduct_empty.state and MakeProduct.state;

if MakeProduct_transition1.condition then
  Valve1.signal := false;
  MakeProduct_FillTank1.nextstate := false;
  MakeProduct_wait1.nextstate := true;
end if;
if MakeProduct_transition2.condition then
  MakeProduct_wait2.nextstate := false;
  Valve3.signal := true;
  MakeProduct_empty.nextstate := true;
end if;
if MakeProduct_transition3.condition then
  MakeProduct_wait1.nextstate := false;
  Valve2.signal := true;
  MakeProduct_FillTank2.nextstate := true;
end if;MakeProduct_transition4.condition then
  Valve2.signal := false;
  MakeProduct_FillTank2.nextstate := false;
  MakeProduct_wait2.nextstate := true;
end if;
if transition1.condition then
  s_1.nextstate := false;
  MakeProduct.nextstate := true;
  Valve1.signal := true;
  MakeProduct_FillTank1.nextstate := true;
end if;
if MakeProduct_exception.condition then
  if MakeProduct_empty.state then
    MakeProduct_empty_storedstate := true;
    MakeProduct_empty.nextstate := false;
  else
    MakeProduct_empty_storedstate := false;
```

```
    MakeProduct_empty.nextstate := false;
  end if;
  if MakeProduct_FillTank1.state then
    MakeProduct_FillTank1_storedstate := true;
    MakeProduct_FillTank1.nextstate := false;
  else
    MakeProduct_FillTank1_storedstate := false;
    MakeProduct_FillTank1.nextstate := false;
  end if;
  if MakeProduct_wait1.state then
    MakeProduct_wait1_storedstate := true;
    MakeProduct_wait1.nextstate := false;
  else
    MakeProduct_wait1_storedstate := false;
    MakeProduct_wait1.nextstate := false;
  end if;
  if MakeProduct_wait2.state then
    MakeProduct_wait2_storedstate := true;
    MakeProduct_wait2.nextstate := false;
  else
    MakeProduct_wait2_storedstate := false;
    MakeProduct_wait2.nextstate := false;
  end if;
  if MakeProduct_FillTank2.state then
    MakeProduct_FillTank2_storedstate := true;
    MakeProduct_FillTank2.nextstate := false;
  else
    MakeProduct_FillTank2_storedstate := false;
    MakeProduct_FillTank2.nextstate := false;
  end if;
  Valve1.signal := false;
  Valve2.signal := false;
  Valve3.signal := false;
  MakeProduct.nextstate := false;
  MakeProduct_aborted = true;
  s_2.nextstate := true;
end if;
if transition2.condition then
  s_2.nextstate := false;
  if MakeProduct_aborted then
    MakeProduct.nextstate := true;
```

```
    MakeProduct_empty.nextstate :=
      MakeProduct_empty_storedstate;
    if MakeProduct_empty.nextstate then
      Valve3.signal := true;
    end if;
    MakeProduct_FillTank1.nextstate :=
      MakeProduct_FillTank1_storedstate;
    if MakeProduct_FillTank1.nextstate then
      Valve1.signal := true;
    end if;
    MakeProduct_wait1.nextstate :=
      MakeProduct_wait1_storedstate;
    MakeProduct_wait2.nextstate :=
      MakeProduct_wait2_storedstate;
    MakeProduct_FillTank2.nextstate :=
      MakeProduct_FillTank2_storedstate;
    if MakeProduct_FillTank2.nextstate then
      Valve2.signal := true;
    end if;
  end if;
end if;
if transition3.condition then
  s_2.nextstate := false;
  Valve3.signal := true;
  Valve2.signal := true;
  emptyTanks.nextstate := true;
end if;
if transition4.condition then
  emptyTanks.nextstate := false;
  Valve1.signal := false;
  Valve2.signal := false;
  Valve3.signal := false;
  s_1.nextstate := true;
end if;
if transition5.condition then
  MakeProduct_empty.nextstate := false;
  Valve1.signal := false;
  Valve2.signal := false;
  Valve3.signal := false;
  s_1.nextstate := true;
end if;
```

```
MakeProduct.nextstate := MakeProduct_empty.nextstate or
  MakeProduct_FillTank1.nextstate or
  MakeProduct_wait1.nextstate or
  MakeProduct_wait2.nextstate or
  MakeProduct_FillTank2.nextstate;

s_1.active := if (s_1.state and s_1.nextstate) then
  s_1.active + scancycle else 0;
MakeProduct.active := if (MakeProduct.state and
  MakeProduct.nextstate) then
  MakeProduct.active + scancycle else 0;
MakeProduct_empty.active := if (MakeProduct_empty.state and
  MakeProduct_empty.nextstate) then
  MakeProduct_empty.active + scancycle else 0;
MakeProduct_FillTank1.active := if
  (MakeProduct_FillTank1.state and
  MakeProduct_FillTank1.nextstate) then
  MakeProduct_FillTank1.active + scancycle else 0;
MakeProduct_wait1.active := if (MakeProduct_wait1.state and
  MakeProduct_wait1.nextstate) then
  MakeProduct_wait1.active + scancycle else 0;
MakeProduct_wait2.active := if (MakeProduct_wait2.state and
  MakeProduct_wait2.nextstate) then
  MakeProduct_wait2.active + scancycle else 0;
MakeProduct_FillTank2.active := if
  (MakeProduct_FillTank2.state and
  MakeProduct_FillTank2.nextstate) then
  MakeProduct_FillTank2.active + scancycle else 0;
s_2.active := if (s_2.state and s_2.nextstate) then
  s_2.active + scancycle else 0;
emptyTanks.active := if (emptyTanks.state and
  emptyTanks.nextstate) then emptyTanks.active + scancycle
  else 0;

s_1.state := s_1.nextstate;
MakeProduct.state := MakeProduct.nextstate;
MakeProduct_empty.state := MakeProduct_empty.nextstate;
MakeProduct_FillTank1.state :=
  MakeProduct_FillTank1.nextstate;
MakeProduct_wait1.state := MakeProduct_wait1.nextstate;
```

```
    MakeProduct_wait2.state := MakeProduct_wait2.nextstate;
    MakeProduct_FillTank2.state :=
      MakeProduct_FillTank2.nextstate;
    s_2.state := s_2.nextstate;
    emptyTanks.state := emptyTanks.nextstate;

  end if;

  annotation (Coordsys(extent=[0, -879; 879, 0]),
    Diagram(
      Line(points=[190, -315; 190, -375; 407, -375; 407, -75;
        190, -75; 190, -85], style(color=0)),
      Line(points=[190, -275; 190, -285; 190, -285; 190, -280;
        190, -280; 190, -290], style(color=0)),
      Line(points=[230, -555; 230, -565; 407, -565; 407, -75;
        190, -75; 190, -85], style(color=0)),
      Line(points=[230, -515; 230, -525; 230, -525; 230, -520;
        230, -520; 230, -530], style(color=0)),
      Line(points=[60, -435; 60, -445; 364, -445; 364, -240;
        231, -240; 221, -240], style(color=0, thickness=2)),
      Line(points=[230, -435; 230, -445; 230, -445; 230, -435;
        230, -435; 230, -445], style(color=0)),
      Line(points=[60, -365; 60, -375; 60, -387; 230, -387;
        230, -400; 230, -410], style(color=0)),
      Line(points=[60, -365; 60, -375; 60, -387; 60, -387;
        60, -400; 60, -410], style(color=0)),
      Line(points=[90, -240; 80, -240; 60, -240; 60, -262;
        60, -285; 60, -295], style(color=0)),
      Line(points=[159, -240; 149, -240; 132, -240; 132, -240;
        115, -240; 105, -240], style(color=0, thickness=2)),
      Line(points=[190, -185; 190, -195; 190, -195; 190, -195;
        190, -195; 190, -205], style(color=0)),
      Line(points=[190, -155; 190, -165; 190, -165; 190, -150;
        190, -150; 190, -160], style(color=0))),
    Icon(
      Rectangle(extent=[20, -859; 859, -20],
        style(color=0, thickness=4)),
      Text(extent=[50, -829; 829, -50], style(color=0,
        thickness=3),string="%name")));
end TankController;
```